ĐẠI HỌC QUỐC GIA TP.HCM
TRƯỜNG ĐẠI HỌC BÁCH KHOA
KHOA KHOA HỌC & KỸ THUẬT MÁY TÍNH

oOo

BÁO CÁO MÔN HỌC CƠ SỞ TOÁN CHO KHMT

NGHIÊN CỨU SO SÁNH CÁC THUẬT TOÁN TỐI ƯU HÓA ADADELTA VÀ ADAM

(Comparative Study of Optimization Algorithms: AdaDelta and Adam)

GIẢNG VIÊN HƯỚNG DẪN
TS. Nguyễn An Khương
TS. Trần Tuấn Anh
PGS. TS. Lê Hồng Trang

HỌC VIÊN THỰC HIỆN
2480859 - Nguyễn Tấn Phú
2570431 - Nguyễn Đăng Khoa
2570315 - Phạm Hoàng Sơn
2570460 - Nguyễn Hoàng Nam
2252906 - Nguyễn Trần Huy Việt

TP. HỒ CHÍ MINH – 2025

LỜI CẢM ƠN

Trong suốt quá trình học môn Cơ Sở Toán cho Khoa học Máy tính, nhóm chúng em đã nhận được sự hướng dẫn tận tâm và những kiến thức quý báu từ Thầy TS. Nguyễn An Khương, Thầy TS. Trần Tuấn Anh và Thầy PGS. TS. Lê Hồng Trang. Những bài giảng và chỉ dẫn của Quý Thầy đã giúp chúng em xây dựng nền tảng vững chắc trong việc hiểu, phân biệt và ứng dụng các cơ sở toán học vào lĩnh vực khoa học máy tính. Nhóm chúng em xin bày tỏ lòng biết ơn chân thành và sâu sắc tới Quý Thầy vì sự tận tụy trong giảng dạy và những định hướng hữu ích dành cho chúng em trong suốt quá trình học tập. Những kiến thức và lời khuyên của Quý Thầy không chỉ hỗ trợ chúng em trong học tập và nghiên cứu, mà còn có ý nghĩa lâu dài đối với con đường nghề nghiệp và cuộc sống sau này.

Cuối cùng, nhóm chúng em kính chúc Quý Thầy luôn mạnh khỏe, an vui và tràn đầy nhiệt huyết để tiếp tục đóng góp cho sự nghiệp giáo dục và truyền đạt tri thức cho các thế hệ sinh viên tiếp theo.

TP. Hồ Chí Minh, Ngày 01 Tháng 12 Năm 2025

Nhóm học viên thực hiện

Đại diện nhóm báo cáo

Học viên. Nguyễn Tấn Phú

Thành viên nhóm và phân công công việc:

Đường dẫn tới: GitHub

Họ và tên	Công việc thực hiện (100%)
Nguyễn Tấn Phú	1 Phần Lý thuyết (.ipynb) – Trình bày phần tổng quan (chương 1) và kiến thức cơ sở (chương 2) về các thuật toán tối ưu và động cơ xuất hiện của AdaDelta và Adam. – Phân tích các hạn chế của Adam trong bài toán lồi và lý do Yogi được đề xuất. – Làm rõ sự khác biệt trong cập nhật v_t giữa Adam và Yogi, và cơ chế giúp Yogi tránh việc "phóng đại" phương sai. 2 Phần Thực nghiệm (.ipynb) – Cài đặt thuật toán Yogi (NumPy/PyTorch). – Thử nghiệm trên dataset MNIST. – So sánh đường loss của Yogi, Adam, AdaDelta, SGD, SGD + Momentum, SGD + Nesterov Momentum, Adagrad và RMSProp với mô hình Multilayer Perceptron (MLP), Convolutional Neural Networks (CNN) đơn giản, VGG16, ResNet-18. – Tổng hợp bảng so sánh: tốc độ hội tụ, độ ổn định, độ dao động loss, accuracy. – Đưa ra gợi ý tình huống phù hợp để sử dụng các thuật toán. 3 Phần Tổng hợp & Báo cáo (.pdf)
Nguyễn Đăng Khoa	1 Phần Lý thuyết (.ipynb) – Trình bày tổng quan về nguyên lý hoạt động của thuật toán AdaDelta.

Thành viên nhóm và phân công công việc:

Họ và tên	Công việc thực hiện (100%)
Nguyễn Đăng Khoa	1 Phần Lý thuyết (.ipynb) – Viết và phân tích đầy đủ các công thức cập nhật của AdaDelta, bao gồm: ước lượng trung bình động bình phương gradient $E[g^2]_t$, ước lượng trung bình động bước cập nhật $E[\Delta x^2]_t$, công thức tính Δx_t và cập nhật tham số. – Giải thích ý nghĩa các đại lượng quan trọng: ρ, ϵ, $E[g^2]_t$, $E[\Delta x^2]_t$, RMS$[g]$, RMS$[\Delta x]$. – Trình bày lập luận và chứng minh trực quan cho thấy AdaDelta khắc phục hạn chế của AdaGrad trong việc làm giảm learning rate quá nhanh. – Hoàn thiện nội dung lý thuyết nhằm làm cơ sở cho phần so sánh và thực nghiệm ở các mục sau.
Phạm Hoàng Sơn	1 Phần Thực nghiệm kết hợp với phần lý thuyết của (Nguyễn Đăng Khoa) (.ipynb) – Cài đặt thuật toán AdaDelta bằng Python (NumPy/PyTorch). – Xây dựng mô hình CNN đơn giản cho bài toán phân loại MNIST. – Huấn luyện mô hình sử dụng AdaDelta và ghi nhận quá trình tối ưu. – Tạo biểu đồ hội tụ (loss theo iteration/epoch). – Tính toán và báo cáo accuracy trên tập kiểm thử. – Chuẩn bị file mã nguồn minh họa và các hình ảnh kết quả để đính kèm báo cáo.

Thành viên nhóm và phân công công việc:

Họ và tên	Công việc thực hiện (100%)
Nguyễn Hoàng Nam	1 Phần Lý thuyết (.ipynb) – Trình bày lý thuyết nền tảng của thuật toán Adam. – Diễn giải đầy đủ công thức toán học và các thành phần trong thuật toán. – Trình bày chi tiết quy tắc cập nhật (update rule) theo từng bước. – Phân tích và giải thích lý do Adam hội tụ nhanh và ổn định, so sánh với các phương pháp tối ưu truyền thống.
Nguyễn Trần Huy Việt	1 Phần Thực nghiệm kết hợp với phần lý thuyết của (Nguyễn Hoàng Nam) (.ipynb) – Cài đặt thuật toán Adam bằng Python (NumPy/PyTorch). – Xây dựng mô hình VGG16 đơn giản cho bài toán phân loại MNIST. – Huấn luyện mô hình sử dụng Adam và ghi nhận quá trình tối ưu. – Thực hiện chạy song song với các thuật toán khác để so sánh đánh giá quá trình hội tụ. – Tạo biểu đồ hội tụ (loss theo iteration/epoch). – Tính toán và báo cáo accuracy trên tập kiểm thử. – Chuẩn bị file mã nguồn minh họa và các hình ảnh kết quả để đính kèm báo cáo.

MỤC LỤC

DANH MỤC TỪ VIẾT TẮT

GD	Gradient Descent
SGD	Stochastic Gradient Descent
NAG	Nesterov Accelerated Gradient
AdaGrad	Adaptive Gradient Algorithm
RMSProp	Root Mean Square Propagation
AdaDelta	Adaptive Learning Rate Method
Adam	Adaptive Moment Estimation Method

DANH MỤC CÁC HÌNH

DANH MỤC CÁC BẢNG

Chương 1.

TỔNG QUAN

1.1 Bối cảnh và động lực nghiên cứu

Trong lĩnh vực học sâu hiện đại, quá trình tối ưu hóa giữ vai trò quyết định đối với hiệu năng của mô hình. Các thuật toán tối ưu hóa truyền thống như Gradient Descent (GD), Stochastic Gradient Descent (SGD), Momentum hay Nesterov Accelerated Gradient (NAG) [1] tuy đã được chứng minh hiệu quả trong nhiều thiết lập, nhưng vẫn tồn tại những hạn chế cố hữu. Cụ thể, chúng yêu cầu lựa chọn tốc độ học tối ưu một cách thủ công, dễ bị ảnh hưởng bởi sự thay đổi trong phân phối gradient, và thường gặp khó khăn khi xử lý các mô hình có không gian tham số lớn hoặc hàm mất mát không lồi.

Sự phát triển của các thuật toán tối ưu hóa thích nghi như AdaGrad [2], RMSProp [3], AdaDelta [4] và Adam [5, 6] đã tạo ra bước tiến đáng kể khi các phương pháp này tự điều chỉnh tốc độ học theo đặc trưng của từng tham số. Trong nhóm này, AdaDelta và Adam nổi bật nhờ khả năng hội tụ nhanh và giảm đáng kể sự phụ thuộc vào việc tinh chỉnh thủ công siêu tham số. Tuy nhiên, chính sự khác biệt trong cơ chế hoạt động, bao gồm cách ước lượng gradient bậc nhất/bậc hai, cách tích lũy và điều chỉnh động tốc độ học, dẫn đến những hành vi hội tụ khác nhau, đôi khi gây tranh luận trong cộng đồng nghiên cứu về tính ổn định và khả năng tổng quát hóa.

Mặc dù cả AdaDelta và Adam đều được triển khai rộng rãi, vẫn tồn tại những khoảng trống trong việc hiểu sâu về điều kiện mà mỗi thuật toán hoạt động hiệu quả, các kịch bản mà một thuật toán vượt trội hơn thuật toán còn lại, và mức độ nhạy cảm của chúng đối với siêu tham số. Thực tế cho thấy Adam thường hội tụ nhanh nhưng có thể gặp vấn đề về tổng quát hóa, trong khi AdaDelta ổn định hơn nhưng đôi khi chậm thích nghi trong các giai đoạn đầu huấn luyện. Những vấn đề này chưa được phân tích một cách hệ thống trong nhiều nghiên cứu ứng dụng.

Chính vì vậy, việc tiến hành đánh giá và so sánh hai thuật toán này dưới cả góc độ lý thuyết và thực nghiệm là cần thiết nhằm: (1) làm rõ đặc tính hội tụ, độ ổn định và hành vi tối ưu hóa; (2) xác định ưu–nhược điểm trong những bối cảnh huấn luyện khác nhau; và (3) cung cấp cơ sở khoa học cho việc lựa chọn thuật toán tối ưu trong thực tiễn. Những động lực này định hướng cho nghiên

cứu hiện tại, đồng thời góp phần bổ sung hiểu biết về cách các thuật toán tối ưu hóa thích nghi vận hành trong các mô hình học sâu quy mô lớn.

1.2 Mục tiêu nghiên cứu

Mục tiêu của nghiên cứu này là phân tích và đánh giá một cách hệ thống hai thuật toán tối ưu hóa thích nghi phổ biến trong học sâu, bao gồm AdaDelta và Adam. Nghiên cứu tập trung làm rõ các đặc tính lý thuyết và hành vi thực nghiệm của từng thuật toán, từ đó chỉ ra những điểm tương đồng, khác biệt và điều kiện ứng dụng hiệu quả. Cụ thể, nghiên cứu hướng tới các mục tiêu sau:

- Phân tích lý thuyết cơ chế cập nhật tham số, giả định nền tảng và các đặc điểm quan trọng chi phối tốc độ hội tụ của AdaDelta và Adam.
- Đánh giá thực nghiệm thông qua việc triển khai và thử nghiệm hai thuật toán trong các mô hình học sâu tiêu biểu, nhằm quan sát sự khác biệt về tốc độ hội tụ, độ ổn định và khả năng tổng quát hóa.
- So sánh hiệu năng dựa trên các chỉ số định lượng như giá trị hàm mất mát, độ chính xác, và mức độ nhạy cảm với siêu tham số.
- Xác định ưu điểm và hạn chế của từng thuật toán, từ đó đưa ra nhận định về phạm vi ứng dụng phù hợp trong các bối cảnh huấn luyện khác nhau.
- Định hướng lựa chọn thuật toán tối ưu, góp phần hỗ trợ người dùng và các nhà nghiên cứu lựa chọn phương pháp tối ưu hóa thích hợp cho các mô hình học sâu hiện đại.

Thông qua các mục tiêu trên, nghiên cứu kỳ vọng cung cấp cái nhìn toàn diện và có giá trị thực tiễn về hai thuật toán AdaDelta và Adam, đồng thời đóng góp vào việc nâng cao hiệu quả của quá trình huấn luyện mô hình trong lĩnh vực học sâu.

1.3 Phạm vi và giới hạn

Nghiên cứu này giới hạn trong việc phân tích và so sánh hai thuật toán tối ưu hóa thích nghi AdaDelta và Adam, phù hợp với mục tiêu làm rõ đặc tính hoạt động và hiệu năng của chúng đã nêu ở phần trước. Các đánh giá thực nghiệm được thực hiện trên một số mô hình học sâu cơ bản và các bộ dữ liệu chuẩn ở quy mô nhỏ, nhằm quan sát hành vi hội tụ và độ ổn định trong những điều kiện minh họa.

Do chỉ tập trung vào hai thuật toán chính và phạm vi mô hình hạn chế, nghiên cứu không hướng đến việc khảo sát toàn bộ các thuật toán tối ưu hóa thích nghi khác hoặc đánh giá trên các kiến trúc mạng và tập dữ liệu phức tạp

hơn. Vì vậy, kết quả mang tính mô tả và định hướng, phù hợp với phạm vi nghiên cứu nhưng không đại diện đầy đủ cho mọi bối cảnh ứng dụng trong học sâu.

1.4 Cấu trúc báo cáo

Báo cáo được tổ chức thành 5 chương như sau:

- **Chương 1:** Tổng quan
- **Chương 2:** Kiến thức cơ sở
- **Chương 3:** Thuật toán AdaDelta và Adam
- **Chương 4:** Thực nghiệm và đánh giá
- **Chương 5:** Kết luận và hướng phát triển.

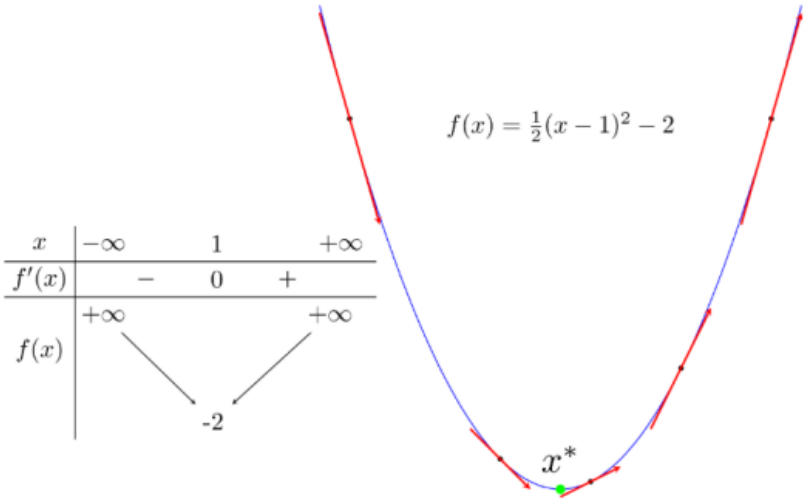
Chương 2.

KIẾN THỨC CƠ SỞ

2.1 Gradient Descent

2.1.1 Ý tưởng thuật toán

Xét bài toán tối ưu một biến với hàm số $f(x) = \frac{1}{2}(x-1)^2 - 2$, có bảng biến thiên và đồ thị như trong Hình 1.1. Ký hiệu x^* là điểm cực trị của hàm số, khi đó ta có $f'(x^*) = 0$. Đường tiếp tuyến với đồ thị hàm số tại một điểm bất kỳ có hệ số góc bằng đúng giá trị của đạo hàm tại điểm đó. Quan sát từ Hình 2.1 cho thấy: các điểm nằm bên trái x^* có đạo hàm âm, trong khi các điểm nằm bên phải x^* có đạo hàm dương. Đồng thời, càng xa x^* về phía bên trái thì đạo hàm càng âm, và càng xa về phía bên phải thì đạo hàm càng dương.



Hình 2.1: Minh hoạ hàm số $f(x) = \frac{1}{2}(x-1)^2 - 2$.

Giả sử x_t là điểm thu được sau vòng lặp thứ t. Mục tiêu của ta là xây dựng một thuật toán sao cho x_t hội tụ và tiến gần đến x^*. Từ hình vẽ, ta có thể rút ra nhận xét sau:

- Nếu đạo hàm tại x_t thỏa $f'(x_t) > 0$ thì x_t nằm về phía bên phải của x^* (và ngược lại). Do đó, để điểm kế tiếp x_{t+1} gần với x^* hơn, ta cần dịch chuyển x_t theo hướng ngược lại dấu của đạo hàm.

$$x_{t+1} = x_t + \Delta \tag{2.1.1}$$

trong đó Δ là một đại lượng ngược dấu với đạo hàm $f'(x_t)$.

- x_t càng xa x^* về phía bên phải thì $f'(x_t)$ càng lớn hơn 0 (và ngược lại). Vậy, lượng di chuyển Δ, một cách trực quan, là tỉ lệ thuận với $-f'(x_t)$.

Từ nhận xét trên, ta có một cập nhật đơn giản:

$$x_{t+1} = x_t - \eta f'(x_t) \tag{2.1.2}$$

trong đó η là một số dương được gọi là tốc độ học. Dấu trừ thể hiện việc chúng ta phải đi ngược dấu đạo hàm (đây cũng là lý do phương pháp này được gọi là Gradient Descent — đi ngược đạo hàm).

2.1.2 Batch Gradient Descent

Batch Gradient Descent [1] là phương pháp trong đó gradient của hàm mục tiêu đối với tham số θ được tính dựa trên toàn bộ tập dữ liệu. Ở phần trước, ta đã khảo sát trường hợp hàm một biến; bây giờ ta mở rộng sang hàm nhiều biến. Giả sử mục tiêu của ta là tìm điểm cực tiểu toàn cục của một hàm $f(\theta)$, trong đó θ là một vector chứa các tham số của mô hình cần tối ưu. Đạo hàm của hàm tại một vị trí bất kỳ được biểu diễn dưới dạng gradient và được ký hiệu là $\nabla_\theta f(\theta)$. Tương tự như trường hợp một biến, Gradient Descent cho hàm nhiều biến bắt đầu từ một điểm khởi tạo θ_0; tại vòng lặp thứ t, quy tắc cập nhật được viết dưới dạng:

$$\theta_{t+1} = \theta_t - \eta \nabla_\theta f(\theta_t), \tag{2.1.3}$$

trong đó η là tốc độ học.

Với Batch Gradient Descent, gradient được tính từ toàn bộ dữ liệu huấn luyện trong mỗi lần cập nhật. Điều này khiến thuật toán trở nên chậm và tốn tài nguyên, đặc biệt khi kích thước dữ liệu lớn và không thể đưa toàn bộ vào bộ nhớ. Hơn nữa, phương pháp này không phù hợp với môi trường học trực tuyến (online learning), bởi mỗi khi xuất hiện dữ liệu mới, việc cập nhật lại yêu cầu tính gradient trên toàn bộ tập dữ liệu, dẫn đến chi phí tính toán rất cao và làm mất đi tính "online" của quá trình tối ưu.

Trong thực tế, một biến thể đơn giản nhưng hiệu quả hơn thường được sử dụng, đó là Stochastic Gradient Descent (SGD).

2.1.3 Stochastic Gradient Descent

Trong thuật toán này, tại mỗi thời điểm ta chỉ sử dụng một điểm dữ liệu x_i để tính gradient của hàm mục tiêu, rồi cập nhật tham số θ dựa trên gradient đó. Quá trình này được lặp lại lần lượt cho toàn bộ các điểm dữ liệu, sau đó

tiếp tục lặp lại nhiều vòng. Mặc dù cách làm rất đơn giản, phương pháp này lại tỏ ra hiệu quả trong thực tế.

Một lần duyệt qua toàn bộ tập dữ liệu được gọi là một epoch. Đối với Gradient Descent dạng batch, mỗi epoch chỉ tương ứng với một lần cập nhật tham số. Ngược lại, với SGD, một epoch bao gồm N lần cập nhật θ, với N là số lượng mẫu dữ liệu. Nhìn từ góc độ này, việc cập nhật sau từng điểm dữ liệu có thể khiến thời gian hoàn thành một epoch lâu hơn. Tuy nhiên, xét ở góc độ khác, SGD [1] thường chỉ cần một số lượng epoch nhỏ để đạt nghiệm tốt (thường khoảng 10 epoch cho lần huấn luyện đầu, và khi có dữ liệu mới chỉ cần chạy dưới một epoch). Điều này khiến SGD phù hợp với các bài toán có quy mô dữ liệu lớn và các ứng dụng đòi hỏi cập nhật mô hình liên tục, tức là chế độ học online. Một điểm quan trọng là sau mỗi epoch, ta cần thực hiện "xáo trộn" lại thứ tự dữ liệu để đảm bảo tính ngẫu nhiên, vì điều này ảnh hưởng trực tiếp đến hiệu quả của thuật toán.

Tóm lại, SGD cập nhật tham số cho từng mẫu huấn luyện $x^{(i)}$ với nhãn $y^{(i)}$ theo công thức:

$$\theta = \theta - \eta \nabla_\theta J(\theta; x^{(i)}, y^{(i)}). \tag{2.1.4}$$

Nếu Batch Gradient Descent tạo ra một đường giảm hàm mất mát "mượt", thì SGD lại có thể đưa tham số nhảy nhanh tới các vùng nghiệm tốt hơn. Mặc dù vậy, tính chất cập nhật theo từng mẫu khiến đường hội tụ có thể dao động mạnh và đôi khi vượt qua cực tiểu. Tuy nhiên, nhiều nghiên cứu chỉ ra rằng khi giảm dần tốc độ học, SGD không chỉ hội tụ nhanh hơn Batch Gradient Descent mà còn gần như chắc chắn tiến về một điểm cực tiểu cục bộ hoặc toàn cục của hàm mục tiêu.

2.1.4 Mini-Batch Gradient Descent

Trong mini-batch gradient descent, thay vì chỉ sử dụng một mẫu dữ liệu như trong SGD, ta sử dụng một nhóm gồm n điểm dữ liệu cho mỗi lần cập nhật, trong đó n lớn hơn 1 nhưng vẫn nhỏ hơn rất nhiều so với kích thước toàn bộ tập dữ liệu N. Tương tự SGD, thuật toán bắt đầu mỗi epoch bằng cách xáo trộn dữ liệu một cách ngẫu nhiên rồi chia thành nhiều nhóm nhỏ (mini-batch), mỗi nhóm chứa n mẫu, ngoại trừ nhóm cuối có thể ít hơn nếu N không chia hết cho n. Ở mỗi bước cập nhật, thuật toán lấy một mini-batch để tính gradient và cập nhật tham số. Công thức cập nhật có dạng:

$$\theta = \theta - \eta\, J(\theta; x^{(i:i+n)}, y^{(i:i+n)}), \tag{2.1.5}$$

trong đó $x^{(i:i+n)}$ biểu thị các mẫu dữ liệu từ vị trí i đến $i+n-1$ theo cách đánh chỉ số thường thấy trong Python. Sau mỗi epoch, dữ liệu cần được xáo trộn lại để đảm bảo tính ngẫu nhiên trong các mini-batch. Mini-batch gradient descent được sử dụng rộng rãi trong hầu hết các bài toán học máy, đặc biệt là trong học sâu. Kích thước mini-batch thường được chọn trong khoảng từ 50 đến 256. Nhìn chung, phương pháp này giúp giảm phương sai của cập nhật tham số so với SGD, từ đó mang lại quá trình hội tụ ổn định hơn.

2.1.5 Các vấn đề của Gradient Descent

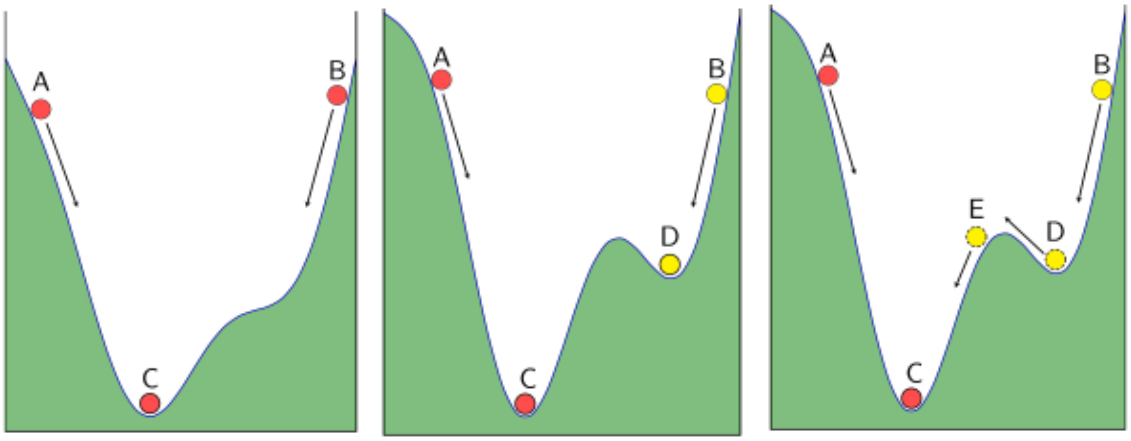
Mặc dù mini-batch gradient descent được sử dụng rộng rãi, thuật toán này không đảm bảo khả năng hội tụ tốt trong mọi trường hợp và vẫn tồn tại một số vấn đề cần được quan tâm:

- **Lựa chọn learning rate phù hợp:** Nếu tốc độ học quá nhỏ, mô hình sẽ tiến gần nghiệm tối ưu một cách rất chậm; ngược lại, nếu quá lớn, quá trình tối ưu có thể dao động quanh điểm cực tiểu hoặc thậm chí bị phân kỳ, khiến việc hội tụ trở nên khó khăn.
- **Thiết lập lịch điều chỉnh learning rate:** Việc giảm dần learning rate theo thời gian hoặc khi hàm mục tiêu đạt đến một mức ngưỡng nào đó có thể cải thiện quá trình học. Tuy nhiên, các phương pháp đặt lịch cố định này thường phải được xác định từ trước và không tự thích ứng với mọi loại dữ liệu hay bài toán.
- **Learning rate đồng nhất cho mọi tham số:** Gradient descent mặc định sử dụng cùng một tốc độ học cho toàn bộ tham số của mô hình, trong khi thực tế mỗi tham số có thể cần mức độ điều chỉnh khác nhau để đạt hiệu quả tối ưu.
- **Khó khăn khi vượt qua cực tiểu địa phương và điểm yên ngựa:** Những vị trí có gradient xấp xỉ bằng 0 theo mọi chiều, đặc biệt là các điểm yên ngựa có thể khiến quá trình tối ưu bị chậm lại đáng kể. Một hướng tiếp cận là chạy gradient descent nhiều lần với các điểm khởi tạo khác nhau và so sánh kết quả. Cách này đôi khi giúp phát hiện các nghiệm tốt hơn, nhưng vẫn không đảm bảo tìm được cực tiểu toàn cục.

Do những hạn chế trên, phần tiếp theo sẽ trình bày các thuật toán cải tiến dựa trên gradient descent như Momentum, NAG hay AdaGrad, nhằm khắc phục các vấn đề nêu trên.

2.2 Momentum

Thuật toán Gradient Descent thường được hình dung giống như việc một hòn bi chuyển động dưới tác dụng của trọng lực trên một bề mặt có dạng thung lũng (Hình 2.2a). Dù hòn bi được đặt tại vị trí A hay B, nó luôn lăn xuống điểm cuối C. Tuy nhiên, nếu bề mặt có nhiều đáy thung lũng như minh họa ở Hình 2.2b, thì tùy thuộc vào vị trí ban đầu, hòn bi có thể dừng lại ở C hoặc mắc kẹt tại D. Điểm D là một cực tiểu địa phương, điều mà ta không mong muốn, như đã được đề cập khi thảo luận về các hạn chế của gradient descent.



(a) GD (b) GD (c) GD with momentum

Hình 2.2: So sánh gradient với các hiện tượng vật lý

Nếu xét hiện tượng này theo góc nhìn vật lý, vẫn ở Hình 2.2b, giả sử hòn bi ở vị trí B được truyền một vận tốc ban đầu đủ lớn. Khi lăn xuống điểm D, quán tính có thể giúp nó tiếp tục vượt lên sườn dốc phía bên trái của D. Nếu vận tốc ban đầu càng lớn, hòn bi có thể tiến tới điểm E và sau đó lăn xuống C, như trong Hình 2.2c. Đây chính là hành vi mong đợi: vượt qua cực tiểu địa phương để đạt nghiệm tốt hơn. Dựa trên quan sát vật lý này, thuật toán Momentum [7] ra đời nhằm giảm khả năng Gradient Descent dừng lại tại những cực tiểu không mong muốn.

Trong Gradient Descent, ta tính độ dịch chuyển tại thời điểm t để cập nhật vị trí nghiệm (tương tự vị trí của hòn bi). Nếu xem đại lượng dịch chuyển này như vận tốc v_t trong cơ học, thì cập nhật tham số có dạng:

$$\theta_{t+1} = \theta_t - v_t, \tag{2.2.1}$$

với dấu trừ biểu thị rằng ta di chuyển ngược hướng gradient. Mục tiêu của ta là xây dựng v_t sao cho nó vừa phản ánh thông tin từ gradient, vừa bao gồm thông

tin về đà chuyển động (tức vận tốc ở bước trước v_{t-1}). Ta giả sử vận tốc ban đầu $v_0 = 0$. Một cách hình thành v_t đơn giản là kết hợp tuyến tính có trọng số của vận tốc cũ và gradient hiện tại:

$$v_t = \gamma v_{t-1} + \eta \nabla_\theta J(\theta), \tag{2.2.2}$$

trong đó γ thường được chọn khoảng 0.9, v_{t-1} là vận tốc từ vòng lặp trước, còn $\nabla_\theta J(\theta)$ là gradient tại vị trí hiện tại. Sau đó, tham số được cập nhật theo: $\theta = \theta - v_t$.. Tóm lại, ta có dạng đầy đủ của thuật toán Momentum:

$$v_t = \gamma v_{t-1} + \eta \nabla_\theta J(\theta), \tag{2.2.3}$$

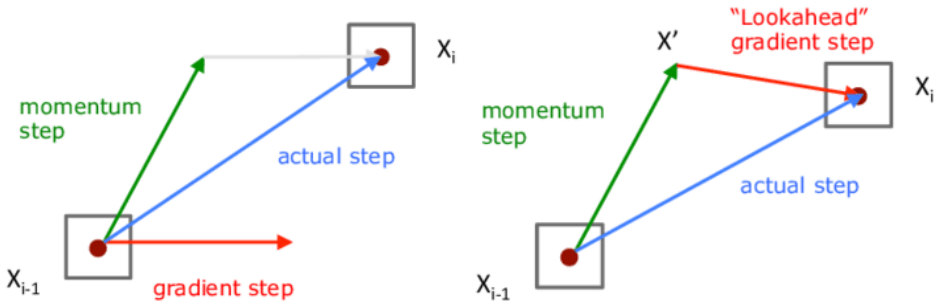
$$\theta = \theta - v_t. \tag{2.2.4}$$

Về mặt trực quan, sử dụng Momentum giống như việc đẩy một quả bóng lăn xuống một thung lũng: bóng liên tục tích lũy vận tốc khi di chuyển và vì vậy ngày càng nhanh hơn. Tuy nhiên, nó vẫn chịu tác động của lực cản như ma sát hoặc lực cản không khí, điều này được mô hình hóa bằng hệ số $\gamma < 1$.

2.3 Nesterow accelerated gradient

Momentum hỗ trợ quá trình tối ưu hóa vượt qua những vùng có gradient nhỏ nhờ tích lũy quán tính, tuy nhiên một hạn chế đáng chú ý là khi thuật toán tiến gần đến nghiệm tối ưu, quán tính này khiến quá trình giảm tốc diễn ra chậm, dẫn đến độ trễ trước khi dừng lại. Hiện tượng này bắt nguồn từ việc vận tốc tích lũy tiếp tục đẩy tham số vượt quá vùng cần thiết. Nesterov Accelerated Gradient (NAG) [8, 9] được đề xuất nhằm khắc phục hạn chế này và tăng tốc độ hội tụ.

Ý tưởng cốt lõi của NAG là ước lượng trước vị trí mà tham số có xu hướng tiến tới, tức là ước lượng trước điểm cập nhật tiếp theo. Cụ thể, thành phần momentum γv_{t-1} cho phép ta dự đoán điểm tiếp theo xấp xỉ là $\theta - \gamma v_{t-1}$ (tạm thời chưa kết hợp gradient vì nó sẽ được sử dụng trong bước cập nhật cuối). Nhờ vậy, thay vì tính gradient tại vị trí hiện tại, NAG sử dụng gradient tại điểm được dự đoán, phản ánh tốt hơn hướng đi trong tương lai. Quan sát Hình 2.3 cho thấy sự khác nhau giữa hai cách tiếp cận:



(a) Momentum update (b) Nesterov momentum update

Hình 2.3: Ý tưởng của Nesterov accelerated gradient

- Trong Momentum thông thường, cập nhật tại bước t được xác định bởi tổng của vector momentum và gradient tại vị trí hiện tại.
- Với NAG, cập nhật được tạo thành từ vector momentum và gradient tại vị trí được ước lượng là bước tiếp theo.

Từ nguyên lý này, ta có được công thức cập nhật của NAG:

$$v_t = \gamma v_{t-1} + \eta \nabla_\theta J(\theta - \gamma v_{t-1}), \tag{2.3.1}$$

$$\theta = \theta - v_t. \tag{2.3.2}$$

2.4 AdaGrad

Trong phần trước, chúng ta đã đề cập rằng Gradient Descent [7] sử dụng một tốc độ học duy nhất cho toàn bộ các tham số. Tuy nhiên, trong thực tế mô hình hóa, có những tham số ít khi được kích hoạt và do đó cần những bước cập nhật lớn hơn; ngược lại, những tham số xuất hiện thường xuyên lại cần bước cập nhật nhỏ để tránh biến động quá mức. Hiện tượng này đặc biệt phổ biến trong các bài toán có dữ liệu thưa. Adagrad được đề xuất để xử lý vấn đề này bằng cách điều chỉnh tốc độ học theo từng tham số riêng biệt. Chẳng hạn, [10] đã áp dụng Adagrad trong quá trình huấn luyện GloVe (một mô hình word embedding trong NLP), nơi các từ hiếm cần bước cập nhật lớn hơn nhiều so với các từ phổ biến. Kết quả cho thấy Adagrad mang lại hiệu quả tốt hơn so với phương pháp Gradient Descent thông thường.

Trước đây, trong Gradient Descent, tất cả tham số θ được cập nhật đồng thời và cùng sử dụng một tốc độ học η. Adagrad thay đổi điều này bằng cách gán một tốc độ học riêng cho từng tham số θ_i tại mỗi bước t. Ký hiệu g_t là gradient tại bước t, và $g_{t,i}$ là đạo hàm riêng theo tham số θ_i:

$$g_{t,i} = \nabla_{\theta_i} J(\theta_t). \tag{2.4.1}$$

Khi đó, quy tắc cập nhật của Gradient Descent với từng tham số là:

$$\theta_{t+1,i} = \theta_{t,i} - \eta\, g_{t,i}. \tag{2.4.2}$$

Trong Adagrad, tốc độ học được điều chỉnh dựa trên tổng bình phương các gradient quá khứ của từng tham số. Cụ thể, tham số được cập nhật theo:

$$\theta_{t+1,i} = \theta_{t,i} - \frac{\eta}{\sqrt{G_{t,ii}} + \epsilon}\, g_{t,i}. \tag{2.4.3}$$

trong đó $G_t \in \mathbb{R}^{d\times d}$ là ma trận đường chéo, với phần tử $G_{t,ii}$ bằng tổng bình phương gradient của θ_i từ đầu quá trình huấn luyện đến thời điểm t, và ϵ là một hằng số nhỏ giúp tránh chia cho 0 (thường dùng 10^{-8}). Thực nghiệm cho thấy việc dùng căn bậc hai trong mẫu số là cần thiết; nếu bỏ đi, thuật toán hoạt động kém hiệu quả hơn. Do G_t tích lũy bình phương gradient dọc theo các bước lặp, công thức cập nhật tổng quát của Adagrad có thể viết dưới dạng vector:

$$\theta_{t+1} = \theta_t - \frac{\eta}{\sqrt{G_t} + \epsilon} \odot g_t. \tag{2.4.4}$$

trong đó ký hiệu "$\odot$" biểu diễn phép nhân theo từng phần tử. Trong thực hành, η thường được chọn khoảng 0.01. Ưu điểm đáng chú ý của Adagrad là khả năng điều chỉnh tốc độ học theo từng tham số, giúp tối ưu mô hình hiệu quả hơn khi dữ liệu có mức độ kích hoạt không đồng đều. Tuy nhiên, hạn chế lớn nhất của Adagrad là tổng bình phương gradient trong mẫu số liên tục tăng, làm cho tốc độ học giảm dần và có thể trở nên rất nhỏ sau một số epoch. Khi đó, thuật toán hầu như không còn khả năng tiếp tục học. Các thuật toán cải tiến ra đời sau này được thiết kế để khắc phục điểm yếu này.

2.5 RMSProp

RMSProp [3] được phát triển nhằm cải thiện hạn chế của AdaGrad, cụ thể là việc tốc độ học giảm dần quá nhanh theo thời gian. Thay vì tích lũy toàn bộ bình phương gradient như AdaGrad, RMSProp sử dụng một trung bình có trọng số mũ của bình phương gradient, giúp thuật toán giữ được khả năng học trong thời gian dài hơn. Giá trị trung bình theo thời gian của bình phương gradient được cập nhật theo quy luật:

$$s_t = \gamma s_{t-1} + (1-\gamma)\,(g_t \odot g_t), \qquad \gamma \in [0,1] \tag{2.5.1}$$

Trong đó, γ thường được đặt khoảng 0.9. Đại lượng s_t phản ánh mức độ "làm mịn" của gradient trong quá trình tối ưu hóa. Sử dụng s_t, RMSProp cập nhật

từng tham số theo công thức:

$$\theta_{t+1} = \theta_t - \frac{\eta}{\sqrt{s_t} + \epsilon} \odot g_t, \tag{2.5.2}$$

với ϵ là hằng số nhỏ nhằm tránh chia cho 0 (thường dùng giá trị 10^{-8}). Công thức này tương tự AdaGrad nhưng thay tổng bình phương gradient bằng một trung bình có trọng số mũ, giúp thuật toán giảm hiện tượng tốc độ học suy giảm đơn điệu.

2.6 Kết luận

Trong các mô hình học máy hiện đại, bài toán tối ưu hoá hàm mất mát đóng vai trò trung tâm và thường được giải bằng các phương pháp dựa trên gradient. Các thuật toán cơ bản như Gradient Descent (GD) và Stochastic Gradient Descent (SGD) dựa trực tiếp trên kiến thức giải tích và tối ưu cơ bản, nhưng chúng bộc lộ nhiều hạn chế về mặt toán học và thực tiễn. Cụ thể, GD yêu cầu tính gradient trên toàn bộ tập dữ liệu và phụ thuộc mạnh vào việc lựa chọn tốc độ học, trong khi SGD – dù hiệu quả hơn, lại gây dao động lớn và hội tụ chậm trong các bài toán có điều kiện kém. Những cải tiến như Momentum phần nào khắc phục được hiện tượng này nhưng vẫn sử dụng một tốc độ học cố định cho toàn bộ không gian tham số, không phản ánh được cấu trúc hình học của hàm mục tiêu.

Để vượt qua những hạn chế đó, các phương pháp thích nghi theo từng chiều tham số được phát triển, với AdaGrad là đại diện đầu tiên. Tuy nhiên, cơ chế tích lũy gradient của AdaGrad lại khiến bước học suy giảm quá nhanh, làm hiệu quả tối ưu kém dần theo thời gian. Trong bối cảnh đó, hai thuật toán AdaDelta và Adam ra đời như các mở rộng mang tính toán học hợp lý: AdaDelta thay thế tổng tích lũy bằng trung bình động suy giảm để duy trì tốc độ học ổn định hơn, còn Adam kết hợp ưu điểm của Momentum và RMSProp, tạo ra một quy tắc cập nhật vừa có tính mượt vừa thích nghi tốt. Nhờ đó, AdaDelta và Adam trở thành hai phương pháp tối ưu quan trọng, minh họa cho cách tiếp cận toán học trong việc thiết kế các thuật toán hiệu quả cho học sâu.

Chương 3.

THUẬT TOÁN ADADELTA VÀ ADAM

3.1 Thuật toán AdaDelta

AdaDelta là một thuật toán tối ưu hóa dùng trong học sâu, được đề xuất bởi [4] nhằm cải tiến Adagrad. Thuật toán thuộc nhóm phương pháp tối ưu thích nghi (adaptive optimization), trong đó tốc độ học của từng tham số được điều chỉnh tự động theo đặc điểm của gradient tại mỗi bước. Khác với Adagrad, vốn làm learning rate giảm nhanh về gần 0 do cộng dồn toàn bộ bình phương gradient, AdaDelta thay phần cộng dồn đó bằng trung bình trượt mũ (Exponential Moving Average - EMA) để giữ cho giá trị cập nhật luôn nằm trong khoảng ổn định. Đồng thời, AdaDelta còn sử dụng thêm một EMA của chính bước cập nhật để chuẩn hóa, giúp tạo ra một dạng learning rate động (adaptive learning rate) mà không cần chọn learning rate ban đầu.

Nói cách khác, AdaDelta tự điều chỉnh bước nhảy dựa trên lịch sử gần đây của gradient và của bước cập nhật, giúp mô hình hội tụ tốt hơn và khắc phục triệt để vấn đề learning rate suy giảm quá nhanh của Adagrad.

3.1.1 Công thức và nguyên lí hoạt động của AdaDelta

Các ký hiệu được sử dụng trong mô tả thuật toán AdaDelta được trình bày trong Bảng 3.1 nhằm giúp thống nhất cách biểu diễn và thuận tiện cho việc theo dõi các biến trong quá trình phân tích.

Bảng 3.1: Các ký hiệu và ý nghĩa trong thuật toán AdaDelta

Ký hiệu	Ý nghĩa
ρ	Hệ số suy giảm, điều khiển mức "ghi nhớ" gradient cũ.
ϵ	Hằng số tránh chia cho 0 và ổn định mẫu số.
$\mathbb{E}[g_t^2]$	EMA của bình phương gradient, xấp xỉ phương sai gradient.
$\mathbb{E}[\Delta x_t^2]$	EMA của bình phương bước cập nhật, giúp cân bằng bước nhảy.
g_t	Gradient tại thời điểm t.
Δx_t	Bước điều chỉnh trọng số tại thời điểm t.
x_t	Trọng số mô hình tại thời điểm t.

Nguyên lý hoạt động của AdaDelta được triển khai thông qua ba bước chính dưới đây, dựa trên việc sử dụng trung bình trượt mũ để điều chỉnh và chuẩn hóa các bước cập nhật.

Bước 1: Ước lượng trung bình bình phương gradient

Thay vì cộng dồn gradient theo thời gian như Adagrad, AdaDelta sử dụng EMA:

$$\mathbb{E}[g_t^2] = \rho\,\mathbb{E}[g_{t-1}^2] + (1-\rho)\,g_t^2 \tag{3.1.1}$$

Trong đó:

- $\mathbb{E}[g_t^2]$: trung bình trượt mũ của bình phương gradient.
- g_t: gradient tại thời điểm t.
- ρ: hệ số suy giảm (0.9 – 0.95).
- ϵ: số nhỏ tránh chia cho 0.

EMA giúp giữ thông tin về gradient gần đây mà không để giá trị tăng vô hạn theo thời gian.

Bước 2: Chuẩn hóa bước cập nhật

Điểm đổi mới quan trọng của AdaDelta nằm ở công thức cập nhật:

$$\Delta x_t = -\frac{\sqrt{\mathbb{E}[\Delta x_{t-1}^2]} + \epsilon}{\sqrt{\mathbb{E}[g_t^2]} + \epsilon}\, g_t \tag{3.1.2}$$

Bước 3: Cập nhật EMA của bước nhảy

$$\mathbb{E}[\Delta x_t^2] = \rho\,\mathbb{E}[\Delta x_{t-1}^2] + (1-\rho)(\Delta x_t)^2 \tag{3.1.3}$$

Sau đó:

$$x_{t+1} = x_t + \Delta x_t \tag{3.1.4}$$

Toàn bộ quá trình hoàn toàn không sử dụng learning rate η - một ưu điểm lớn so với SGD, Momentum và Adagrad.

3.1.2 Chứng minh AdaDelta khắc phục hạn chế của Adagrad

Với Adagrad, bước cập nhật theo từng tọa độ có dạng cập nhật theo công thức:

$$\Delta x_{t,i} = -\frac{\eta}{\sqrt{G_{t,i}} + \epsilon} \cdot g_{t,i}, \tag{3.1.5}$$

$$G_{t,i} = \sum_{\tau=1}^{t} g_{\tau,i}^2. \tag{3.1.6}$$

Ở đây $G_{t,i}$ là tổng cộng dồn bình phương gradient nên luôn không giảm theo thời gian (mỗi bước lại cộng thêm một số không âm). Nếu bài toán không quá "yên lặng" (tức là $|g_{\tau,i}|$ không nhanh chóng về 0), ta sẽ có xấp xỉ $G_{t,i} \approx c_i t \quad (c_i > 0)$, nên "learning rate hiệu dụng"

$$\eta_{t,i}^{(\text{AdaGrad})} = \frac{\eta}{\sqrt{G_{t,i}} + \epsilon} \approx \frac{\eta}{\sqrt{c_i t}} \xrightarrow[t\to\infty]{} 0. \tag{3.1.7}$$

Nói cách khác, càng huấn luyện lâu thì bước nhảy càng nhỏ, tới mức gần như đứng yên dẫn tới mô hình ngừng học mặc dù vẫn còn sai số. AdaDelta thay cơ chế cộng dồn bằng trung bình trượt mũ (EMA):

$$\mathbb{E}[g_{t,i}^2] = \rho\, \mathbb{E}[g_{t-1,i}^2] + (1-\rho) g_{t,i}^2. \tag{3.1.8}$$

Khác với $G_{t,i}$, đại lượng $\mathbb{E}[g_{t,i}^2]$ không tăng vô hạn, vì mỗi bước ta chỉ giữ lại tỉ lệ $\rho < 1$ của giá trị cũ và thêm một phần nhỏ gradient mới. Nếu gradient dao động quanh một giá trị "ổn định" với phương sai xấp xỉ σ_i^2, thì nghiệm cố định của phương trình trên cho ta:

$$\mathbb{E}[g_{t,i}^2] \to \sigma_i^2$$

(một hằng số hữu hạn, không phụ thuộc t). Bước cập nhật của AdaDelta là:

$$\Delta x_{t,i} = -\frac{\sqrt{\mathbb{E}[\Delta x_{t-1,i}^2] + \epsilon}}{\sqrt{\mathbb{E}[g_{t,i}^2] + \epsilon}} \cdot g_{t,i}. \tag{3.1.9}$$

Khi quá trình học đi vào "trạng thái ổn định", cả hai EMA $\mathbb{E}[g_{t,i}^2]$ và $\mathbb{E}[\Delta x_{t,i}^2]$ đều hội tụ tới các hằng số hữu hạn. Khi đó, ta có thể xem:

$$\eta_{t,i}^{(\text{AdaDelta})} = \frac{\sqrt{\mathbb{E}[\Delta x_{t-1,i}^2] + \epsilon}}{\sqrt{\mathbb{E}[g_{t,i}^2] + \epsilon}} \approx \text{hằng số } \neq 0. \tag{3.1.10}$$

Như vậy, learning rate hiệu dụng của AdaDelta không bị kéo về 0 theo thời gian. Đồng thời, việc chia cho $\sqrt{\mathbb{E}[g_{t,i}^2]}$ vẫn giữ được tính "thích nghi theo tọa độ" giống Adagrad: tọa độ nào có gradient lớn lâu dài thì bước nhảy sẽ nhỏ lại, còn những tọa độ ít cập nhật thì vẫn được phép nhảy lớn hơn. Ngoài ra, xét trên hai góc độ:

- **Góc độ toán học**

Trong Adagrad, đại lượng tích lũy $G_t = \sum_{\tau=1}^{t} g_\tau^2$ tăng xấp xỉ tuyến tính theo t (vì mỗi bước đều cộng thêm số không âm). Do đó, learning rate hiệu dụng $\eta_t = \frac{\eta}{\sqrt{G_t}}$ giảm dần theo $\frac{1}{\sqrt{t}}$ và tiến về 0 khi t lớn. Điều này khiến tốc độ học ngày càng nhỏ cho đến khi mô hình gần như dừng cập nhật.

- **Góc độ cơ chế của AdaDelta**

AdaDelta thay thế việc cộng dồn bằng **trung bình trượt mũ (EMA)**:

$$\mathbb{E}[g_t^2] = \rho\,\mathbb{E}[g_{t-1}^2] + (1-\rho)g_t^2, \tag{3.1.11}$$

khiến đại lượng này không tăng vô hạn mà hội tụ về một giá trị hữu hạn.

Đồng thời, AdaDelta chuẩn hóa bước cập nhật bằng tỉ lệ: $\frac{\sqrt{\mathbb{E}[\Delta x_{t-1}^2]}}{\sqrt{\mathbb{E}[g_t^2]}}$ vốn cũng có xu hướng ổn định theo thời gian. Vì vậy, bước cập nhật không suy giảm về 0 mà tự điều chỉnh quanh một mức cố định. Điều này cho thấy AdaDelta khắc phục trực tiếp hiện tượng "learning rate giảm quá nhanh" mà Adagrad gặp phải.

3.1.3 Bài toán minh họa

Tìm cực tiểu của hàm có dạng:

$$f(x) = (x-5)^2, \qquad f'(x) = 2(x-5)$$

Mục tiêu: tìm nghiệm tối ưu $x^* = 5$.
Chọn điểm khởi tạo $x_0 = 20$.
Với công thức Adagrad:

$$G_t = \sum_{j=1}^{t} g_j^2, \qquad x_{t+1} = x_t - \frac{\eta}{\sqrt{G_t + \epsilon}}\, g_t$$

Ta chọn $\eta = 1$, $\epsilon \approx 0$ cho đơn giản.
Bước 1 (t = 0 → 1)

- $x_0 = 20$
- Gradient:

$$g_1 = f'(x_0) = 2(x_0 - 5) = 2(20-5) = 30$$

- Tích luỹ bình phương gradient:

$$G_1 = g_1^2 = 30^2 = 900$$

- Learning rate hiệu dụng:

$$\eta_1 = \frac{\eta}{\sqrt{G_1}} = \frac{1}{\sqrt{900}} = \frac{1}{30} \approx 0.0333$$

- Bước cập nhật:

$$\Delta x_1 = -\eta_1 g_1 = -0.0333 \cdot 30 \approx -1$$

- Giá trị mới:

$$x_1 = x_0 + \Delta x_1 \approx 20 - 1 = 19$$

Bước 2 (t = 1 → 2)

- $x_1 \approx 19$
- Gradient:

$$g_2 = f'(x_1) = 2(x_1 - 5) = 2(19 - 5) = 28$$

- Tích luỹ bình phương gradient:

$$G_2 = G_1 + g_2^2 = 900 + 28^2 = 900 + 784 = 1684$$

- Leaning rate hiệu dụng:

$$\eta_2 = \frac{1}{\sqrt{1684}} \approx 0.0244$$

- Bước cập nhật:

$$\Delta x_2 = -\eta_2\, g_2 \approx -0.0244 \cdot 28 \approx -0.682$$

- Giá trị mới:

$$x_2 = x_1 + \Delta x_2 \approx 19 - 0.682 = 18.318$$

Tiếp tục thực hiện các bước tiếp theo, kết quả được thể hiện ở Bảng 3.2:

Bảng 3.2: Kết quả các bước tính của Adagrad

t	x_{t-1}	$g_t = 2(x_{t-1} - 5)$	G_t	$\eta_t = \frac{1}{\sqrt{G_t}}$	$\Delta x_t = -\eta_t g_t$	$x_t = x_{t-1} + \Delta x_t$
1	20.000	30.000	900.000	0.0333	-1.000	19.000
2	19.000	28.000	1684.000	0.0244	-0.682	18.318
3	18.318	26.636	2393.500	0.0204	-0.544	17.774
4	17.774	25.548	3046.200	0.0181	-0.463	17.311
5	17.311	24.622	3652.400	0.0165	-0.407	16.904
6	16.904	23.808	4219.200	0.0154	-0.366	16.538
7	16.538	23.076	4751.700	0.0145	-0.335	16.203
8	16.203	22.406	5253.700	0.0138	-0.309	15.894
9	15.894	21.788	5728.300	0.0132	-0.288	15.606
10	15.606	21.212	6178.300	0.0127	-0.270	15.336

Nhận xét:

- G_t tăng rất nhanh (900 → 6178).
- Learning rate hiệu dụng $\eta_t = \frac{1}{\sqrt{G_t}}$ giảm từ ≈ 0.0333 xuống ≈ 0.0127.

- Bước cập nhật Δx_t giảm từ -1.000 xuống còn khoảng -0.270, và tiếp tục giảm nữa nếu lặp tiếp.
- Đây chính là hiện tượng learning rate giảm dần về 0, khiến thuật toán ngày một "chậm lại".

Với công thức AdaDelta:

$$\mathbb{E}[g_t^2] = \rho\,\mathbb{E}[g_{t-1}^2] + (1-\rho)g_t^2 \qquad \Delta x_t = -\frac{\sqrt{\mathbb{E}[\Delta x_{t-1}^2] + \epsilon}}{\sqrt{\mathbb{E}[g_t^2] + \epsilon}}\, g_t$$

$$\mathbb{E}[\Delta x_t^2] = \rho\,\mathbb{E}[\Delta x_{t-1}^2] + (1-\rho)(\Delta x_t)^2 \qquad x_{t+1} = x_t + \Delta x_t$$

Chọn tham số: $\rho = 0.9$, $\epsilon = 10^{-6}$.
Ban đầu: $\mathbb{E}[g_0^2] = 0$, $\mathbb{E}[\Delta x_0^2] = 0$.

Bước 1 ($\mathbf{t = 0 \to 1}$)

- $x_0 = 20$
- Gradient:

$$g_1 = 2(x_0 - 5) = 30$$

- EMA của bình phương gradient:

$$\mathbb{E}[g_1^2] = 0.9 \cdot 0 + 0.1 \cdot 30^2 = 0.1 \cdot 900 = 90$$

- Tỉ lệ learning rate hiệu dụng:

$$\alpha_1 = \frac{\sqrt{\mathbb{E}[\Delta x_0^2] + \epsilon}}{\sqrt{\mathbb{E}[g_1^2] + \epsilon}} \approx \frac{\sqrt{10^{-6}}}{\sqrt{90}} \approx \frac{0.001}{9.487} \approx 1.05 \times 10^{-4}$$

- Bước cập nhật:

$$\Delta x_1 = -\alpha_1 g_1 \approx -1.05 \times 10^{-4} \cdot 30 \approx -0.00316$$

- Giá trị mới:

$$x_1 = x_0 + \Delta x_1 \approx 20 - 0.00316 = 19.997$$

- Cập nhật EMA bước nhảy:

$$\mathbb{E}[\Delta x_1^2] = 0.9 \cdot 0 + 0.1 \cdot (0.00316)^2 \approx 1.0 \times 10^{-6}$$

Bước 2 ($\mathbf{t = 1 \to 2}$)

- $x_1 \approx 19.997$
- Gradient:

$$g_2 = 2(x_1 - 5) \approx 29.994$$

- EMA của bình phương gradient:

$$\mathbb{E}[g_2^2] = 0.9 \cdot 90 + 0.1 \cdot (29.994)^2 \approx 0.9 \cdot 90 + 0.1 \cdot 899.6 \approx 171.0$$

- Learning rate hiệu dụng:

$$\alpha_2 = \frac{\sqrt{\mathbb{E}[\Delta x_1^2] + \epsilon}}{\sqrt{\mathbb{E}[g_2^2] + \epsilon}} \approx \frac{\sqrt{2 \times 10^{-6}}}{\sqrt{171}} \approx 1.08 \times 10^{-4}$$

- Bước cập nhật:

$$\Delta x_2 = -\alpha_2 g_2 \approx -1.08 \times 10^{-4} \cdot 29.994 \approx -0.00324$$

- Giá trị mới:

$$x_2 = x_1 + \Delta x_2 \approx 19.997 - 0.00324 = 19.9938$$

- Cập nhật EMA bước nhảy:

$$\mathbb{E}[\Delta x_2^2] \approx 0.9 \cdot 10^{-6} + 0.1 \cdot (0.00324)^2 \approx 2.0 \times 10^{-6}$$

Tiếp tục thực hiện các bước tiếp theo, kết quả được thể hiện ở Bảng 3.3:

Bảng 3.3: Kết quả các bước tính của AdaDelta

t	x_{t-1}	g_t	$\mathbb{E}[g_t^2]$	$\mathbb{E}[\Delta x_{t-1}^2]$	$\alpha_t = \frac{\sqrt{\mathbb{E}[\Delta x_{t-1}^2]+\epsilon}}{\sqrt{\mathbb{E}[g_t^2]+\epsilon}}$	$\Delta x_t = -\alpha_t g_t$	$x_t = x_{t-1} + \Delta x_t$
1	20.000	30.000	90.000	0.000000	$\approx 1.05 \times 10^{-4}$	−0.00316	19.997
2	19.997	29.994	171.000	0.000001	$\approx 1.08 \times 10^{-4}$	−0.00324	19.994
3	19.994	29.987	243.788	0.000002	$\approx 1.10 \times 10^{-4}$	−0.00330	19.990
4	19.987	29.981	309.293	0.000003	$\approx 1.11 \times 10^{-4}$	−0.00335	19.987
5	19.987	29.974	368.208	0.000004	$\approx 1.13 \times 10^{-4}$	−0.00338	19.984
6	19.984	29.967	421.190	0.000005	$\approx 1.14 \times 10^{-4}$	−0.00341	19.980
7	19.980	29.960	468.834	0.000006	$\approx 1.15 \times 10^{-4}$	−0.00344	19.977
8	19.977	29.954	511.673	0.000006	$\approx 1.16 \times 10^{-4}$	−0.00347	19.973
9	19.973	29.947	550.187	0.000007	$\approx 1.16 \times 10^{-4}$	−0.00348	19.970
10	19.970	29.940	584.808	0.000008	$\approx 1.17 \times 10^{-4}$	−0.00351	19.966

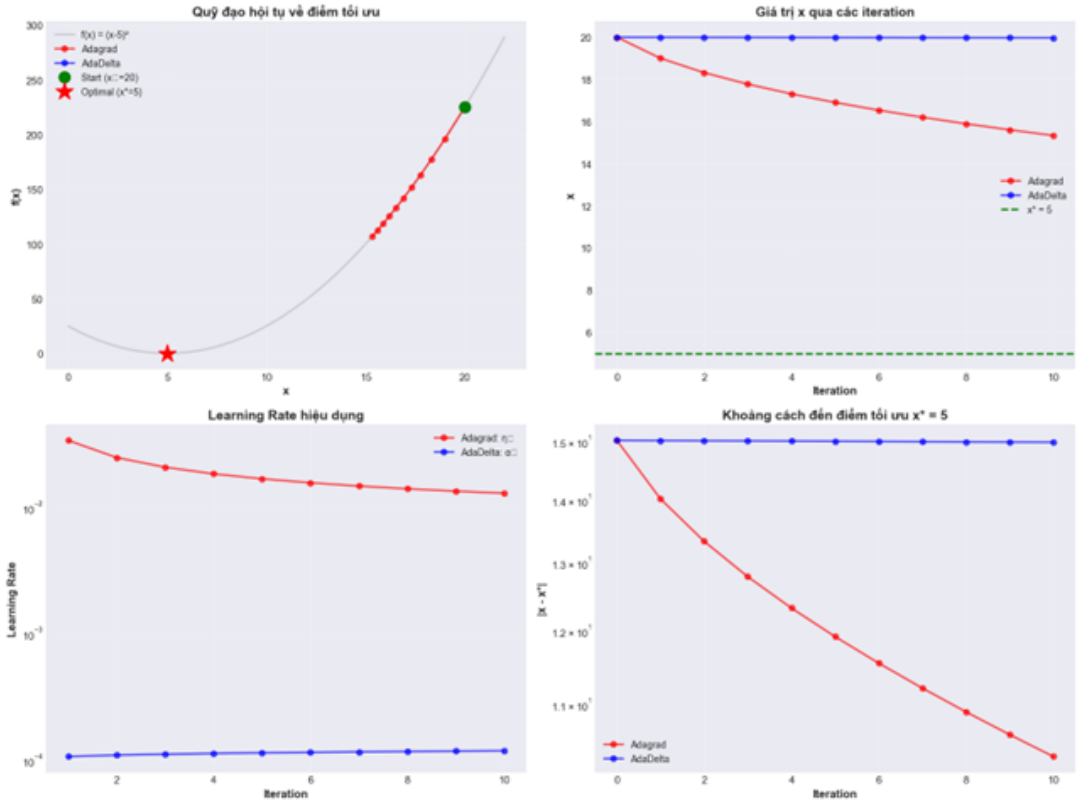
Nhận xét:

- Cả $\mathbb{E}[g_t^2]$ và $\mathbb{E}[\Delta x_t^2]$ tăng dần nhưng sau một thời gian sẽ ổn định quanh một giá trị hữu hạn (không tăng vô hạn như G_t của Adagrad).

- Hệ số "learning rate hiệu dụng" α_t ổn định quanh $\approx 1.1 \times 10^{-4}$.
- Bước cập nhật Δx_t gần như không giảm, dao động quanh -0.0034 đến -0.0035, thay vì tiến về 0 như Adagrad.
- Sau 10 bước, x_t chỉ giảm từ $20 \to 19.97 \to 19.966$, nhưng vẫn tiếp tục dịch chuyển với tốc độ gần như không đổi, cho thấy AdaDelta *không bị chững lại* khi t lớn.

3.1.4 Kết luận

Từ phân tích lý thuyết và bài toán minh họa, có thể rút ra một số kết luận chính về AdaDelta như sau. Thứ nhất, xét trên góc độ toán học, việc thay thế tổng bình phương gradient cộng dồn bằng trung bình trượt mũ giúp các đại lượng $\mathbb{E}[g_t^2]$ và $\mathbb{E}[\Delta x_t^2]$ hội tụ về các giá trị hữu hạn, từ đó làm cho "learning rate hiệu dụng" của AdaDelta ổn định theo thời gian, không suy giảm về 0 như trong Adagrad. Thứ hai, cơ chế chuẩn hóa bước cập nhật bằng tỉ lệ $\sqrt{\mathbb{E}[\Delta x_{t-1}^2]}/\sqrt{\mathbb{E}[g_t^2]}$ vừa giữ được tính thích nghi theo từng tọa độ, vừa cho phép mô hình tiếp tục dịch chuyển với tốc độ hợp lý ngay cả khi số bước lặp lớn, qua đó cải thiện khả năng hội tụ trong thực nghiệm.



Hình 3.1: So sánh quỹ đạo hội tụ và đặc trưng cập nhật của Adagrad và AdaDelta trên bài toán tối ưu $f(x) = (x-5)^2$.

Cuối cùng, bài toán tối ưu hàm $f(x) = (x-5)^2$ với cùng điều kiện ban đầu cho thấy rõ sự khác biệt: trong khi bước nhảy của Adagrad giảm nhanh từ -1 xuống khoảng -0.27 chỉ sau 10 bước, thì bước nhảy của AdaDelta dao động quanh giá trị xấp xỉ -0.0034 và hầu như không suy giảm. Điều này minh họa trực quan ưu điểm của AdaDelta trong việc duy trì động lực học ổn định cho tham số. Cuối cùng, mặc dù hiện nay các thuật toán như RMSProp hay Adam được sử dụng rộng rãi hơn trong thực hành, AdaDelta vẫn là một mảnh ghép quan trọng trong "gia đình" các phương pháp learning rate thích nghi, góp phần hình thành nên tư duy thiết kế các optimizer hiện đại và là ví dụ điển hình cho cách khắc phục hạn chế của Adagrad bằng EMA và chuẩn hóa bước cập nhật.

3.2 Thuật toán Adam

Trong bối cảnh tối ưu hóa trong học máy và đặc biệt là học sâu, việc cập nhật tham số mô hình dựa trên thông tin gradient là nền tảng của hầu hết các thuật toán huấn luyện hiện đại. Tuy nhiên, khi sử dụng stochastic gradient descent (SGD) thuần túy, quá trình học gặp rất nhiều thách thức do bản chất nhiễu và không ổn định của gradient được tính từ các minibatch. Những khó khăn chính có thể được liệt kê một cách rõ ràng như sau:

1. **Tính ngẫu nhiên của minibatch (stochasticity)**: Gradient tại mỗi bước chỉ được ước lượng trên một tập con nhỏ và ngẫu nhiên của toàn bộ tập dữ liệu. Điều này dẫn đến phương sai lớn trong hướng cập nhật, khiến đường đi của tham số trở nên "zig-zag" và làm chậm tốc độ hội tụ, đặc biệt ở những vùng phẳng của hàm mất mát.
2. **Độ cong không đồng nhất của bề mặt hàm mất mát (ill-conditioned loss surface)**: Trong các mạng nơ-ron sâu, ma trận Hessian của hàm mất mát thường có các giá trị riêng trải rất rộng (từ rất nhỏ đến rất lớn). Điều này tạo ra các thung lũng hẹp mà SGD thuần túy di chuyển rất chậm dọc theo các hướng có độ cong lớn (tức là giá trị riêng lớn), trong khi lại dao động mạnh theo các hướng có độ cong nhỏ.
3. **Hiện tượng gradient flipping**: Ở những khu vực gần điểm uốn hoặc khi minibatch thay đổi mạnh, dấu của gradient theo cùng một chiều tham số có thể đảo ngược liên tục giữa các bước huấn luyện liền kề. Điều này làm triệt tiêu động lượng nếu sử dụng momentum thông thường và khiến thuật toán mất nhiều thời gian để thoát khỏi các vùng dao động cục bộ.
4. **Sự khác biệt về tỷ lệ quan trọng giữa các tham số**: Trong các kiến trúc phức tạp (ví dụ lớp fully-connected cuối cùng so với các lớp convolution đầu tiên), độ lớn của gradient có thể chênh lệch hàng chục đến hàng trăm

lần. Một tốc độ học cố định (global learning rate) sẽ không thể đồng thời phù hợp với tất cả các tham số: quá lớn với một số tham số sẽ gây phát tán, quá nhỏ với các tham số khác lại làm hội tụ quá chậm.

Để khắc phục đồng thời tất cả các vấn đề trên, năm 2014, Diederik Kingma và Jimmy Ba đã đề xuất thuật toán Adam (Adaptive Moment Estimation) [11]. Adam là sự kết hợp thông minh giữa hai ý tưởng cốt lõi đã chứng minh hiệu quả trước đó:

- **Momentum**: Tích lũy thông tin gradient từ quá khứ theo kiểu "động lượng" để giảm dao động và tăng tốc độ di chuyển ở những vùng phẳng hoặc khi gradient liên tục đổi dấu.
- **Adaptive learning rates**: Tự động điều chỉnh tốc độ học riêng biệt cho từng tham số dựa trên độ lớn lịch sử của gradient, giúp giải quyết vấn đề ill-conditioning và sự khác biệt tỷ lệ giữa các tham số.

Thuật toán Adam [11] giải quyết các vấn đề này bằng cách kết hợp hai ý tưởng chính: *momentum* (tích lũy thông tin gradient) và *adaptive learning rates* (điều chỉnh tốc độ học theo từng tham số). Cả hai đều dựa trên kỹ thuật Exponential Moving Average (EMA).

3.2.1 Exponential Moving Average

3.2.1.1 Định nghĩa và công thức cơ bản

Cho chuỗi giá trị $\{g_t\}_{t=1}^{\infty}$, Exponential Moving Average (EMA) được định nghĩa đệ quy:

$$v_t = \beta v_{t-1} + (1-\beta)g_t, \quad \beta \in [0,1), \quad v_0 = 0 \tag{3.2.1}$$

Trong đó:

- v_t: giá trị EMA tại thời điểm t;
- g_t: giá trị quan sát (gradient) tại thời điểm t;
- β: hệ số làm mịn (decay factor);
- $1-\beta$: trọng số của quan sát hiện tại.

3.2.1.2 Khai triển công thức EMA

Bằng cách khai triển đệ quy, ta có thể biểu diễn v_t dưới dạng tổ hợp tuyến tính có trọng số của toàn bộ lịch sử gradient. Điều này giúp làm rõ bản chất "nhớ dài hạn" của EMA. Xuất phát từ công thức đệ quy cơ bản (3.2.1) với điều kiện khởi tạo $v_0 = 0$, ta lần lượt khai triển các bước đầu tiên:

Bước 1: Tính v_1

$$\begin{aligned} v_1 &= \beta v_0 + (1-\beta)g_1 \\ &= \beta \cdot 0 + (1-\beta)g_1 \\ &= (1-\beta)g_1 \end{aligned} \tag{3.2.2}$$

Bước 2: Tính v_2 bằng cách thay v_1 vào công thức đệ quy

$$\begin{aligned} v_2 &= \beta v_1 + (1-\beta)g_2 \\ &= \beta \cdot (1-\beta)g_1 + (1-\beta)g_2 \\ &= (1-\beta)[\beta g_1 + g_2] \\ &= (1-\beta)g_2 + \beta(1-\beta)g_1 \end{aligned} \tag{3.2.3}$$

Bước 3: Tính v_3 bằng cách thay v_2 vào công thức đệ quy

$$\begin{aligned} v_3 &= \beta v_2 + (1-\beta)g_3 \\ &= \beta \cdot (1-\beta)[\beta g_1 + g_2] + (1-\beta)g_3 \\ &= (1-\beta)[\beta^2 g_1 + \beta g_2] + (1-\beta)g_3 \\ &= (1-\beta)g_3 + \beta(1-\beta)g_2 + \beta^2(1-\beta)g_1 \end{aligned} \tag{3.2.4}$$

Bước 4: Tính v_4 tương tự

$$\begin{aligned} v_4 &= \beta v_3 + (1-\beta)g_4 \\ &= \beta \cdot (1-\beta)[\beta^2 g_1 + \beta g_2 + g_3] + (1-\beta)g_4 \\ &= (1-\beta)[\beta^3 g_1 + \beta^2 g_2 + \beta g_3] + (1-\beta)g_4 \\ &= (1-\beta)g_4 + \beta(1-\beta)g_3 + \beta^2(1-\beta)g_2 + \beta^3(1-\beta)g_1 \end{aligned} \tag{3.2.5}$$

Bước 5: Tính v_5 và viết dưới dạng gọn

$$\begin{aligned} v_5 &= \beta v_4 + (1-\beta)g_5 \\ &= \beta \cdot (1-\beta)[\beta^3 g_1 + \beta^2 g_2 + \beta g_3 + g_4] + (1-\beta)g_5 \\ &= (1-\beta)g_5 + \beta(1-\beta)g_4 + \beta^2(1-\beta)g_3 + \beta^3(1-\beta)g_2 + \beta^4(1-\beta)g_1 \\ &= (1-\beta)[g_5 + \beta g_4 + \beta^2 g_3 + \beta^3 g_2 + \beta^4 g_1] \end{aligned} \tag{3.2.6}$$

Ý nghĩa của hệ số β^k: Hệ số β^k quyết định mức độ ảnh hưởng của gradient g_{t-k} (từ k bước trước) lên giá trị EMA hiện tại. Vì $0 < \beta < 1$, ta có chuỗi giảm dần nghiêm ngặt:

$$1 = \beta^0 > \beta^1 > \beta^2 > \beta^3 > \beta^4 > \cdots > \beta^k > \cdots \xrightarrow{k\to\infty} 0 \tag{3.2.7}$$

Ví dụ cụ thể với $\beta = 0.9$:

- $\beta^1 = 0.9000 \Rightarrow$ gradient từ 1 bước trước giữ 90% ảnh hưởng
- $\beta^4 = 0.6561 \Rightarrow$ gradient từ 4 bước trước giữ 65.61% ảnh hưởng
- $\beta^{10} = 0.3487 \Rightarrow$ gradient từ 10 bước trước giữ 34.87% ảnh hưởng
- $\beta^{20} = 0.1216 \Rightarrow$ gradient từ 20 bước trước chỉ còn 12.16% ảnh hưởng

Điều này giải thích tại sao EMA có khả năng "lọc nhiễu": các gradient ngẫu nhiên từ quá khứ xa có trọng số rất nhỏ, trong khi xu hướng gần đây được duy trì.

Nhận xét quy luật: Từ các phương trình (3.2.2)–(3.2.6), ta nhận thấy quy luật chung:

- Mỗi gradient g_i (với $i \leq t$) được nhân với hệ số $(1-\beta)\beta^{t-i}$
- Gradient càng xa trong quá khứ (chỉ số i nhỏ) có lũy thừa của β càng lớn, tức trọng số càng nhỏ
- Gradient hiện tại (g_t) có hệ số lũy thừa bằng 0 (tức $\beta^0 = 1$), do đó có trọng số lớn nhất

Công thức tổng quát: Tổng quát hóa quy luật trên cho bước thời gian t bất kỳ:

$$v_t = (1-\beta)\sum_{i=1}^{t} \beta^{t-i} g_i \tag{3.2.8}$$

Hoặc viết theo thứ tự ngược (từ hiện tại về quá khứ) bằng cách đặt $k = t - i$:

$$v_t = (1-\beta)\sum_{k=0}^{t-1} \beta^k g_{t-k} \tag{3.2.9}$$

Minh họa bằng sơ đồ:

$$v_t = (1-\beta)[\underbrace{\beta^0}_{\text{hiện tại}} g_t + \underbrace{\beta^1}_{\text{1 bước trước}} g_{t-1} + \underbrace{\beta^2}_{\text{2 bước trước}} g_{t-2} + \cdots + \underbrace{\beta^{t-1}}_{t-1 \text{ bước trước}} g_1]$$

$$= (1-\beta)[g_t + \beta g_{t-1} + \beta^2 g_{t-2} + \beta^3 g_{t-3} + \cdots + \beta^{t-1} g_1]$$

Ý nghĩa của công thức khai triển:

1. **Trọng số giảm dần theo thời gian**: Hệ số $(1-\beta)\beta^k$ giảm theo hàm mũ khi k tăng
   - Với $\beta = 0.9$: $\beta^{10} \approx 0.349$ (chỉ còn 34.9% sau 10 bước)
   - Với $\beta = 0.99$: $\beta^{10} \approx 0.904$ (còn 90.4% sau 10 bước)
2. **Tổng trọng số chưa đạt 1**: Tổng các hệ số trọng số là

$$(1-\beta)\sum_{k=0}^{t-1} \beta^k = (1-\beta)\cdot\frac{1-\beta^t}{1-\beta} = 1-\beta^t < 1 \tag{3.2.10}$$

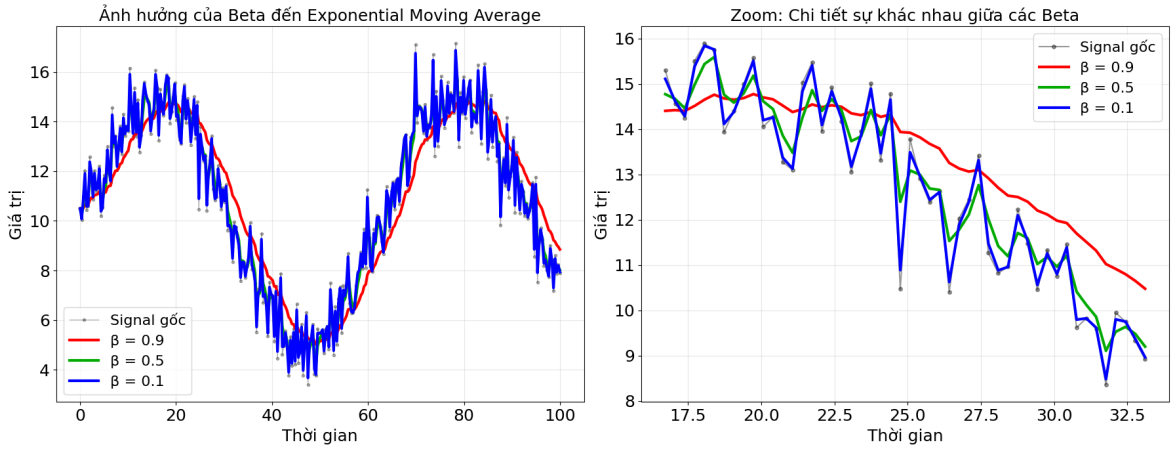
Điều này dẫn đến hiện tượng **initialization bias** (thiên lệch khởi tạo) sẽ được phân tích chi tiết ở Mục 3.2.2.1.

3. **Bộ nhớ hiệu dụng hữu hạn**: Mặc dù lý thuyết EMA "nhớ" toàn bộ lịch sử, nhưng thực tế chỉ khoảng $\frac{1}{1-\beta}$ bước gần nhất có ảnh hưởng đáng kể
   - $\beta = 0.9 \Rightarrow$ bộ nhớ ≈ 10 bước
   - $\beta = 0.99 \Rightarrow$ bộ nhớ ≈ 100 bước
   - $\beta = 0.999 \Rightarrow$ bộ nhớ ≈ 1000 bước

Bảng 3.4: Phân rã trọng số β^k theo thời gian với các giá trị β khác nhau

Bước k	$\beta = 0.9$	$\beta = 0.95$	$\beta = 0.99$	$\beta = 0.999$
0 (hiện tại)	1.000	1.000	1.000	1.000
1	0.900	0.950	0.990	0.999
5	0.590	0.774	0.951	0.995
10	0.349	0.599	0.904	0.990
20	0.122	0.358	0.818	0.980
50	0.005	0.077	0.605	0.951
100	≈ 0	0.006	0.366	0.905

3.2.1.3 Mô phỏng hoạt động với β khác nhau:



Hình 3.2: Phân bố trọng số theo hàm mũ trong EMA với các giá trị β khác nhau. Trục hoành: thời gian ngược (k), trục tung: trọng số $(1-\beta)\beta^k$.

3.2.2 Vấn đề Bias trong EMA và giải pháp hiệu chỉnh

3.2.2.1 Nguồn gốc của Initialization Bias

Khi khởi tạo $v_0 = 0$, EMA gặp phải một vấn đề nghiêm trọng: giá trị ước lượng ban đầu bị **thiên lệch về 0** (initialization bias). Để hiểu rõ vấn đề này,

Bảng 3.5: So sánh các giá trị β điển hình trong tối ưu hóa

β	**Bộ nhớ hiệu dụng**	**Ứng dụng**	$1-\beta^{10}$
0.9	$\approx$ 10 bước	Momentum	0.651
0.99	$\approx$ 100 bước	Adam (m_t)	0.096
0.999	$\approx$ 1000 bước	Adam (v_t)	0.010

ta xét trường hợp đơn giản nhất.

Giả thiết: Gradient có kỳ vọng không đổi theo thời gian: $\mathbb{E}[g_t] = \mu$ với mọi t. Từ công thức khai triển (3.2.9), ta tính kỳ vọng của v_t:

$$\begin{aligned}
\mathbb{E}[v_t] &= \mathbb{E}\left[(1-\beta)\sum_{k=0}^{t-1}\beta^k g_{t-k}\right] \\
&= (1-\beta)\sum_{k=0}^{t-1}\beta^k \mathbb{E}[g_{t-k}] \quad \text{(tuyến tính của kỳ vọng)} \\
&= (1-\beta)\sum_{k=0}^{t-1}\beta^k \cdot \mu \quad \text{(vì } \mathbb{E}[g_{t-k}] = \mu\text{)} \\
&= (1-\beta)\mu\sum_{k=0}^{t-1}\beta^k \quad \text{(đưa } \mu \text{ ra ngoài)}
\end{aligned} \tag{3.2.11}$$

Tính tổng cấp số nhân trong (3.2.11):

$$\sum_{k=0}^{t-1}\beta^k = \frac{1-\beta^t}{1-\beta} \quad \text{(công thức tổng cấp số nhân)} \tag{3.2.12}$$

Thay (3.2.12) vào (3.2.11):

$$\begin{aligned}
\mathbb{E}[v_t] &= (1-\beta)\mu \cdot \frac{1-\beta^t}{1-\beta} \\
&= \mu(1-\beta^t)
\end{aligned} \tag{3.2.13}$$

Kết luận: Kỳ vọng của EMA không bằng giá trị thật μ, mà bị giảm đi một hệ số $(1-\beta^t)$:

$$\mathbb{E}[v_t] = (1-\beta^t)\mu \neq \mu \tag{3.2.14}$$

Sai số tương đối (relative bias) là:

$$\text{Relative Bias} = \frac{\mu - \mathbb{E}[v_t]}{\mu} = \frac{\mu - (1-\beta^t)\mu}{\mu} = \beta^t \tag{3.2.15}$$

Bias β^t đặc biệt lớn ở các bước đầu huấn luyện. Bảng 3.6 minh họa mức độ nghiêm trọng với các giá trị β thường dùng:

- Với $\beta = 0.9$ (thường dùng cho momentum): bias chỉ đáng kể trong 10-20 bước đầu
- Với $\beta = 0.99$ hoặc 0.999 (thường dùng trong Adam): bias kéo dài hàng trăm bước
- β càng lớn (bộ nhớ càng dài), bias càng nghiêm trọng và kéo dài

Bảng 3.6: Relative bias β^t theo thời gian với các giá trị β khác nhau

t	$\beta = 0.9$	$\beta = 0.95$	$\beta = 0.99$	$\beta = 0.999$
1	90.0%	95.0%	99.0%	99.9%
5	59.0%	77.4%	95.1%	99.5%
10	34.9%	59.9%	90.4%	99.0%
20	12.2%	35.8%	81.8%	98.0%
50	0.5%	7.7%	60.5%	95.1%
100	$\approx 0\%$	0.6%	36.6%	90.5%

3.2.2.2 Cơ chế Bias Correction

Để khắc phục initialization bias, ta sử dụng một phép hiệu chỉnh đơn giản nhưng hiệu quả: chia cho hệ số thiếu hụt.

Ý tưởng: Vì $\mathbb{E}[v_t] = (1-\beta^t)\mu$ thay vì μ, ta cần chia v_t cho $(1-\beta^t)$ để bù lại phần thiếu hụt. Giá trị EMA sau hiệu chỉnh bias (Bias-corrected EMA) được định nghĩa:

$$\hat{v}_t = \frac{v_t}{1-\beta^t} \tag{3.2.16}$$

Kiểm chứng tính unbiased: Ta tính kỳ vọng của $\hat{v}_t$:

$$\begin{aligned}
\mathbb{E}[\hat{v}_t] &= \mathbb{E}\left[\frac{v_t}{1-\beta^t}\right] \\
&= \frac{1}{1-\beta^t}\mathbb{E}[v_t] \quad (\text{vì } \frac{1}{1-\beta^t} \text{ là hằng số}) \\
&= \frac{1}{1-\beta^t} \cdot (1-\beta^t)\mu \quad (\text{thay (3.2.13)}) \\
&= \mu
\end{aligned} \tag{3.2.17}$$

Kết luận: Sau bias correction, kỳ vọng của ước lượng bằng đúng giá trị thật:

$$\mathbb{E}[\hat{v}_t] = \mu \quad (\text{unbiased estimator}) \tag{3.2.18}$$

Tính chất của bias correction:

1. **Tự động suy giảm**: Khi $t \to \infty$, $\beta^t \to 0$ nên $(1 - \beta^t) \to 1$, tức $\hat{v}_t \to v_t$. Hiệu chỉnh chỉ ảnh hưởng mạnh ở giai đoạn đầu.
2. **Chi phí tính toán thấp**: Chỉ cần một phép chia, không làm tăng đáng kể độ phức tạp.
3. **Ổn định số học**: Mẫu số $(1 - \beta^t) > 0$ với mọi $t \geq 1$, không gây vấn đề chia cho 0.
4. **Hiệu quả ngay từ bước đầu tiên**: Ngay cả khi $t = 1$, ước lượng đã chính xác.

3.2.2.3 Minh họa Bias Correction

Để thấy rõ hiệu quả của bias correction, ta xét trường hợp gradient không đổi: $g_t = 1$ với mọi t (tức $\mu = 1$), và $\beta = 0.9$.

Tính v_t theo công thức đệ quy:

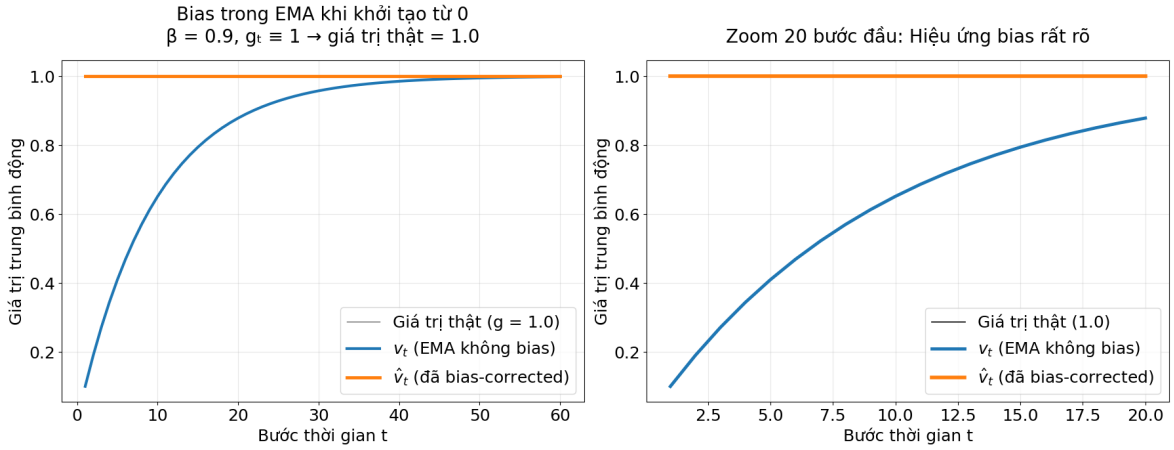
$$\begin{aligned}
v_1 &= 0.9 \times 0 + 0.1 \times 1 = 0.100 \\
v_2 &= 0.9 \times 0.100 + 0.1 \times 1 = 0.190 \\
v_3 &= 0.9 \times 0.190 + 0.1 \times 1 = 0.271 \\
v_4 &= 0.9 \times 0.271 + 0.1 \times 1 = 0.344 \\
v_5 &= 0.9 \times 0.344 + 0.1 \times 1 = 0.410 \\
v_{10} &= \cdots = 0.651
\end{aligned}$$

Tính $\hat{v}_t$ sau bias correction:

$$\begin{aligned}
\hat{v}_1 &= \frac{0.100}{1 - 0.9^1} = \frac{0.100}{0.100} = 1.000 \\
\hat{v}_2 &= \frac{0.190}{1 - 0.9^2} = \frac{0.190}{0.190} = 1.000 \\
\hat{v}_3 &= \frac{0.271}{1 - 0.9^3} = \frac{0.271}{0.271} = 1.000 \\
\hat{v}_5 &= \frac{0.410}{1 - 0.9^5} = \frac{0.410}{0.410} = 1.000 \\
\hat{v}_{10} &= \frac{0.651}{1 - 0.9^{10}} = \frac{0.651}{0.651} = 1.000
\end{aligned}$$

Bảng 3.7: So sánh EMA trước và sau bias correction ($g_t \equiv 1$, $\beta = 0.9$)

t	v_t	$1 - \beta^t$	$\hat{v}_t$	**Error (%)**	**Corrected Error**
1	0.100	0.100	1.000	-90.0	0
2	0.190	0.190	1.000	-81.0	0
3	0.271	0.271	1.000	-72.9	0
5	0.410	0.410	1.000	-59.0	0
10	0.651	0.651	1.000	-34.9	0
20	0.878	0.878	1.000	-12.2	0
50	0.995	0.995	1.000	-0.5	0



Hình 3.3: So sánh EMA có/không bias correction trên tín hiệu nhiễu.

Kết luận từ ví dụ:

- **Không bias correction**: v_t luôn nhỏ hơn giá trị thật (1.0), với sai số từ 90% (bước 1) đến 0.5% (bước 50)
- **Có bias correction**: $\hat{v}_t = 1.000$ chính xác tuyệt đối ngay từ bước đầu tiên
- Hiệu chỉnh đặc biệt quan trọng trong 10-20 bước đầu khi bias còn lớn. Chính vì vậy, bias correction thường được ứng dụng trong các bài toán tối ưu.

3.2.3 Sự kết hợp EMA và Adaptive Learning Rate

Ý tưởng

Adam (Adaptive Moment Estimation) kết hợp hai thành phần:

1. **First moment** (m_t): EMA của gradient → Momentum
2. **Second moment** (v_t): EMA của bình phương gradient → RMSProp

Cả hai đều được hiệu chỉnh bias để đảm bảo ước lượng chính xác ngay từ đầu.

Cho hàm mục tiêu $f(\theta)$ với tham số $\theta \in \mathbb{R}^d$:

Algorithm 1: Adam Optimizer

Input : Learning rate α, decay rates $\beta_1, \beta_2 \in (0, 1)$, stabilizer ϵ

Initialize: θ_0, $m_0 \leftarrow 0$, $v_0 \leftarrow 0$, $t \leftarrow 0$

while θ_t *not converged* **do**

$t \leftarrow t + 1$

$g_t \leftarrow \nabla_\theta f_t(\theta_{t-1})$

$m_t \leftarrow \beta_1 m_{t-1} + (1 - \beta_1) g_t$

$v_t \leftarrow \beta_2 v_{t-1} + (1 - \beta_2) g_t^2$

$\hat{m}_t \leftarrow m_t / (1 - \beta_1^t)$

$\hat{v}_t \leftarrow v_t / (1 - \beta_2^t)$

$\theta_t \leftarrow \theta_{t-1} - \alpha \cdot \hat{m}_t / (\sqrt{\hat{v}_t} + \epsilon)$

return θ_t

3.2.4 Phân tích chi tiết từng thành phần

3.2.4.1 First Moment Estimate (m_t)

$$m_t = \beta_1 m_{t-1} + (1 - \beta_1) g_t \tag{3.2.19}$$

Ý nghĩa: Tích lũy hướng của gradient, giúp thuật toán "nhớ" xu hướng di chuyển trong không gian tham số.

3.2.4.2 Second Moment Estimate (v_t)

$$v_t = \beta_2 v_{t-1} + (1 - \beta_2) g_t^2 \tag{3.2.20}$$

Ý nghĩa: Ước lượng độ biến thiên (variance) của gradient theo từng chiều, cho phép điều chỉnh learning rate riêng cho mỗi tham số.

3.2.4.3 Bias Correction

$$\hat{m}_t = \frac{m_t}{1 - \beta_1^t} \tag{3.2.21}$$

$$\hat{v}_t = \frac{v_t}{1 - \beta_2^t} \tag{3.2.22}$$

Ý nghĩa: Loại bỏ bias do khởi tạo từ zero, đặc biệt quan trọng ở các iteration đầu tiên.

3.2.4.4 Parameter Update Rule

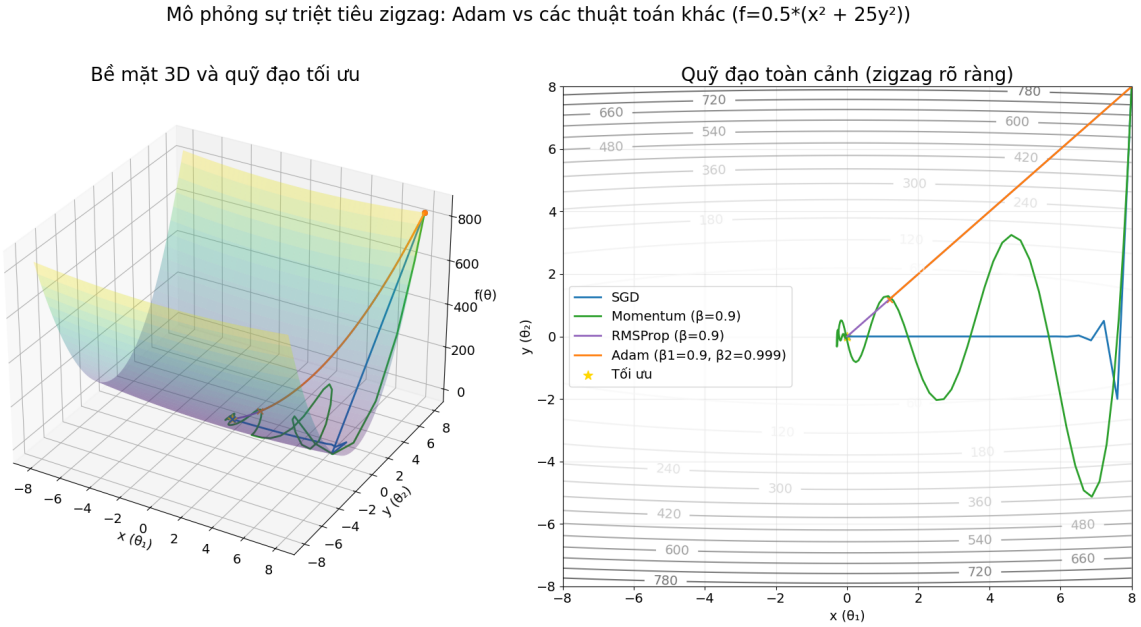
$$\theta_t = \theta_{t-1} - \alpha \cdot \frac{\hat{m}_t}{\sqrt{\hat{v}_t} + \epsilon} \tag{3.2.23}$$

Ý nghĩa:

- Tử số $\hat{m}_t$: hướng di chuyển (có momentum)
- Mẫu số $\sqrt{\hat{v}_t}$: điều chỉnh độ lớn bước nhảy theo từng chiều
- ϵ: tránh chia cho 0, tăng numerical stability

3.2.5 Hiệu quả của Adam

1. **Adaptive per-parameter learning rates**: Mỗi tham số có tốc độ học riêng dựa trên lịch sử gradient của nó
2. **Momentum giúp vượt qua local minima**: Tích lũy gradient giúp không bị mắc kẹt ở các điểm yên ngựa
3. **Robust với noisy gradients**: EMA lọc nhiễu từ stochastic gradients
4. **Bias correction cho khởi động nhanh**: Không cần "warm-up" phase dài
5. **Giảm hiện tượng zig-zag trong tối ưu**: Nhờ momentum và learning rate thích nghi, hướng cập nhật ổn định hơn khi tối ưu trên các hàm dạng hẻm núi (ravine).



Hình 3.4: Mô phỏng sự triệt tiêu zigzag: Adam vs các thuật toán khác.

3.2.6 Khuyến nghị thực hành

Siêu tham số mặc định và các biến thể và cải tiến:

Bảng 3.8: Giá trị siêu tham số khuyến nghị cho Adam

Tham số	Giá trị mặc định	Ghi chú
α	10^{-3} hoặc 10^{-4}	Có thể dùng learning rate schedule
β_1	0.9	Hiếm khi cần thay đổi
β_2	0.999	Có thể giảm nếu training không ổn định
ϵ	10^{-8}	Tăng nếu gặp vấn đề numerical

- **AdamW**: Decoupled weight decay, tốt hơn cho regularization
- **AMSGrad**: Đảm bảo second moment không giảm, cải thiện convergence
- **RAdam**: Rectified Adam với warm-up tự động
- **AdaBound**: Chuyển từ Adam sang SGD ở cuối training

3.2.7 Khi nào nên dùng Adam

Nên dùng Adam khi:
- Training mô hình học sâu phức tạp (CNN, Transformer, etc.)
- Dữ liệu có nhiễu hoặc không cân bằng
- Cần hội tụ nhanh trong thời gian giới hạn
- Không muốn tune learning rate quá kỹ

Cân nhắc SGD khi:
- Cần generalization tốt nhất (SGD với momentum đôi khi tốt hơn)
- Training rất dài với learning rate schedule tốt
- Đã có kinh nghiệm tune SGD cho bài toán cụ thể

Thuật toán Adam là một thành tựu quan trọng trong tối ưu hóa học sâu, kết hợp khéo léo momentum và adaptive learning rates thông qua EMA. Hai đóng góp chính của Adam là: (1) cơ chế EMA kép: Sử dụng EMA cho cả gradient và bình phương gradient; (2) Bias correction: Đảm bảo ước lượng chính xác ngay từ các bước đầu tiên Hiểu rõ nền tảng toán học của EMA và bias correction không chỉ giúp sử dụng Adam hiệu quả mà còn là cơ sở để phát triển các thuật toán tối ưu mới trong tương lai.

3.3 Thuật toán Yogi

Mặc dù Adam thường cho tốc độ hội tụ nhanh và hiệu quả thực nghiệm tốt, thuật toán này vẫn tồn tại hạn chế quan trọng đến từ cơ chế tích lũy moment

bậc hai. Do moment này được cập nhật theo dạng trung bình động của các bình phương gradient không âm, nó có xu hướng tăng đơn điệu và dễ bị ảnh hưởng mạnh bởi một số giá trị gradient đột biến. Khi moment bậc hai phình to, bước cập nhật bị thu nhỏ một cách bất thường, khiến quá trình tối ưu hóa trở nên chậm hoặc thậm chí không tiến đến nghiệm mong muốn. Để khắc phục những hạn chế của Adam, Zaheer et al. (2018) [6] đã đề xuất thuật toán Yogi, với mục tiêu chính: kiểm soát sự tăng lên không mong muốn của moment bậc hai, từ đó ổn định bước cập nhật.

Thay vì sử dụng moment bậc hai tăng đơn điệu như Adam, Yogi cập nhật:

$$v_t = v_{t-1} + (1 - \beta_2) g_t^2 \operatorname{sgn}(g_t^2 - v_{t-1}), \tag{3.3.1}$$

trong đó $\operatorname{sgn}(\cdot)$ là hàm dấu.

Ý nghĩa toán học:

- Nếu $g_t^2 > v_{t-1}$, Yogi tăng v_t một lượng nhỏ.
- Nếu $g_t^2 < v_{t-1}$, Yogi giảm v_t, giúp moment bậc hai không bị phình to.
- v_t có xu hướng tiến về giá trị cân bằng phản ánh trung bình thực sự của gradient.

Moment bậc nhất vẫn giống Adam:

$$m_t = \beta_1 m_{t-1} + (1 - \beta_1) g_t. \tag{3.3.2}$$

Cập nhật tham số:

$$\theta_{t+1} = \theta_t - \alpha \frac{\hat{m}_t}{\sqrt{\hat{v}_t} + \epsilon}. \tag{3.3.3}$$

Nguyên lý chính của Yogi: Yogi không cho phép moment bậc hai tăng không giới hạn, mà điều chỉnh động theo quan hệ giữa g_t^2 và v_{t-1}. Điều này giúp: Bảo toàn sự ổn định của learning rate hiệu dụng; Tránh hiện tượng bước cập nhật bị triệt tiêu; Cải thiện hành vi hội tụ so với Adam, đặc biệt trong điều kiện gradient nhiễu hoặc xuất hiện không thường xuyên. Để hiện thực hóa cơ chế kiểm soát moment bậc hai như đã phân tích, thuật toán Yogi được xây dựng với các bước cập nhật cụ thể sau đây.

Algorithm 2: YOGI Optimizer

Input : Initial point $x_1 \in \mathbb{R}^d$, learning rates $\{\eta_t\}_{t=1}^T$, hyperparameters $0 < \beta_1, \beta_2 < 1$, $\epsilon > 0$

Initialize: $m_0 = 0,\ v_0 = 0$

for $t = 1$ **to** T **do**

1. Sample s_t from data distribution P
2. Compute stochastic gradient: $g_t = \nabla \ell(x_t, s_t)$
3. Update first moment estimate: $m_t = \beta_1 m_{t-1} + (1 - \beta_1) g_t$
4. Update second moment estimate (Yogi update):
   $v_t = v_{t-1} + (1 - \beta_2)\, g_t^2\, \mathrm{sgn}(g_t^2 - v_{t-1})$
5. Update parameters: $x_{t+1} = x_t - \eta_t \dfrac{m_t}{\sqrt{v_t} + \epsilon}$

Thuật toán 1 mô tả mã giả của Yogi, trong đó quy tắc cập nhật có hình thức gần giống AdaGrad nhưng khác biệt quan trọng nằm ở việc sử dụng hàm dấu $\mathrm{sgn}(g_t^2 - v_{t-1})$. Tương tự Adam, tham số ε đóng vai trò điều chỉnh mức độ thích nghi của phương pháp. Sự khác nhau giữa hai thuật toán chủ yếu xuất hiện ở bước cập nhật mô-men bậc hai v_t.

Trong YOGI, thay đổi $v_t - v_{t-1}$ được tính bằng:

$$(1 - \beta_2)\, g_t^2\ \mathrm{sgn}(g_t^2 - v_{t-1}) \tag{3.3.4}$$

trong khi ở ADAM, thay đổi này là:

$$-(1 - \beta_2)\,(v_{t-1} - g_t^2) \tag{3.3.5}$$

Cả hai thuật toán đều duy trì đặc tính: v_t chỉ phụ thuộc vào v_{t-1} và giá trị g_t^2 tại thời điểm hiện tại. Tuy nhiên, không giống Adam, biên độ cập nhật trong YOGI chỉ bị chi phối bởi g_t^2, thay vì phụ thuộc đồng thời vào cả v_{t-1} và g_t^2. Điều này giúp Yogi điều chỉnh mô-men bậc hai một cách thận trọng hơn.

Khi v_{t-1} lớn so với g_t^2, cả Adam lẫn Yogi đều làm tăng tốc độ học hiệu dụng. Tuy vậy, Adam có thể khiến tốc độ học tăng quá nhanh, trong khi Yogi chỉ tăng ở mức vừa phải và dễ kiểm soát hơn — một đặc điểm thường dẫn đến hiệu quả thực nghiệm tốt hơn. Trong những tình huống cần thay đổi tốc độ học mạnh hơn, Yogi vẫn có thể mô phỏng hành vi này bằng cách lựa chọn β_2 nhỏ hơn.

Ngoài ra, Yogi có chi phí tính toán và bộ nhớ bậc $O(d)$, tương tự Adam, nên dễ dàng triển khai trên quy mô lớn.

Mô tả bài toán minh họa sự khác biệt giữa Adam và Yogi: Để minh họa sự khác biệt trong cách Adam và Yogi xử lý mô-men bậc hai v_t, một chuỗi gradient nhân tạo $|g_t|^2$ được sử dụng thay vì một hàm mất mát cụ thể. Chuỗi

này bao gồm ba giá trị đầu tiên lớn ($|g_t|^2 = 1.0$) và các giá trị còn lại rất nhỏ ($|g_t|^2 = 0.01$). Cách tạo gradient như vậy mô phỏng hiện tượng "điểm bất thường" trong huấn luyện mô hình học sâu, nơi gradient có thể tăng đột ngột trong một vài bước rồi trở nên nhỏ trong giai đoạn dài.

Dựa trên chuỗi gradient này, hai thuật toán cập nhật mô-men bậc hai theo công thức:

Adam

$$v_t = \beta_2 v_{t-1} + (1 - \beta_2)\, g_t^2 \tag{3.3.6}$$

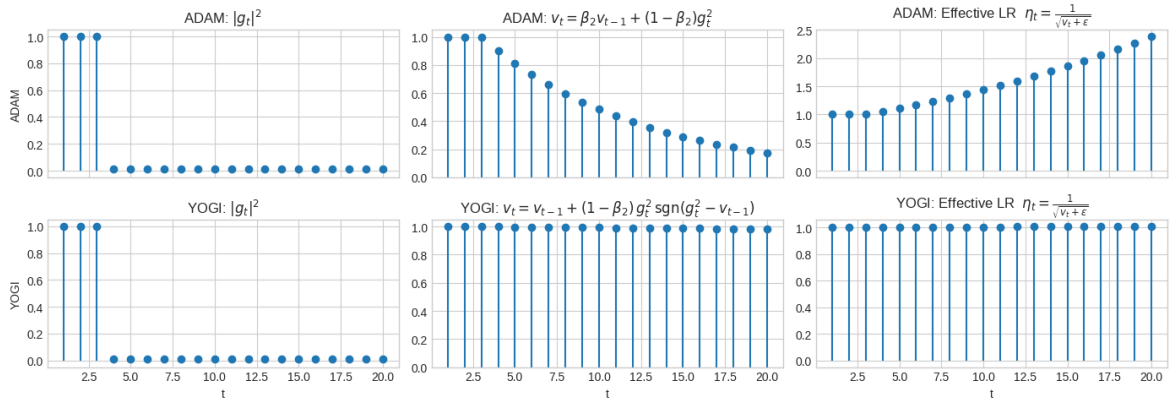
Yogi

$$v_t = v_{t-1} + (1 - \beta_2)\, g_t^2 \, \mathrm{sgn}(g_t^2 - v_{t-1}) \tag{3.3.7}$$

Hệ số học hiệu dụng được suy ra từ mô-men bậc hai:

$$\eta_t = \frac{1}{v_t + \varepsilon} \tag{3.3.8}$$

Mục tiêu của ví dụ này là quan sát cách mỗi thuật toán phản ứng với sự thay đổi đột ngột của gradient: Adam có xu hướng làm v_t suy giảm nhanh dẫn đến hệ số học tăng đột biến, trong khi Yogi điều chỉnh v_t theo hướng ổn định, từ đó tạo ra hệ số học tăng chậm và có kiểm soát hơn. Được thể hiện ở Hình 3.5 bên dưới.



Hình 3.5: Minh họa sự khác biệt giữa Adam vs Yogi.

Từ mô phỏng trên, ta rút ra:

- Adam dễ bị ảnh hưởng mạnh bởi các spike trong gradient → làm v_t giảm quá nhanh → learning rate tăng không kiểm soát → dễ gây bất ổn.
- Yogi điều chỉnh v_t theo hướng ổn định hóa, chỉ tăng hoặc giảm rất nhẹ theo sự khác biệt giữa g_t^2 và v_{t-1}.

Nhờ đó, learning rate của Yogi tăng chậm và đều, giúp quá trình tối ưu ổn định hơn đáng kể. Do đó, Yogi được xem như một phiên bản khắc phục nhược điểm của Adam, đặc biệt hữu ích trong các bài toán non-convex hoặc dữ liệu nhiễu.

Chương 4.

THỰC NGHIỆM VÀ ĐÁNH GIÁ

4.1 Cấu hình thực nghiệm

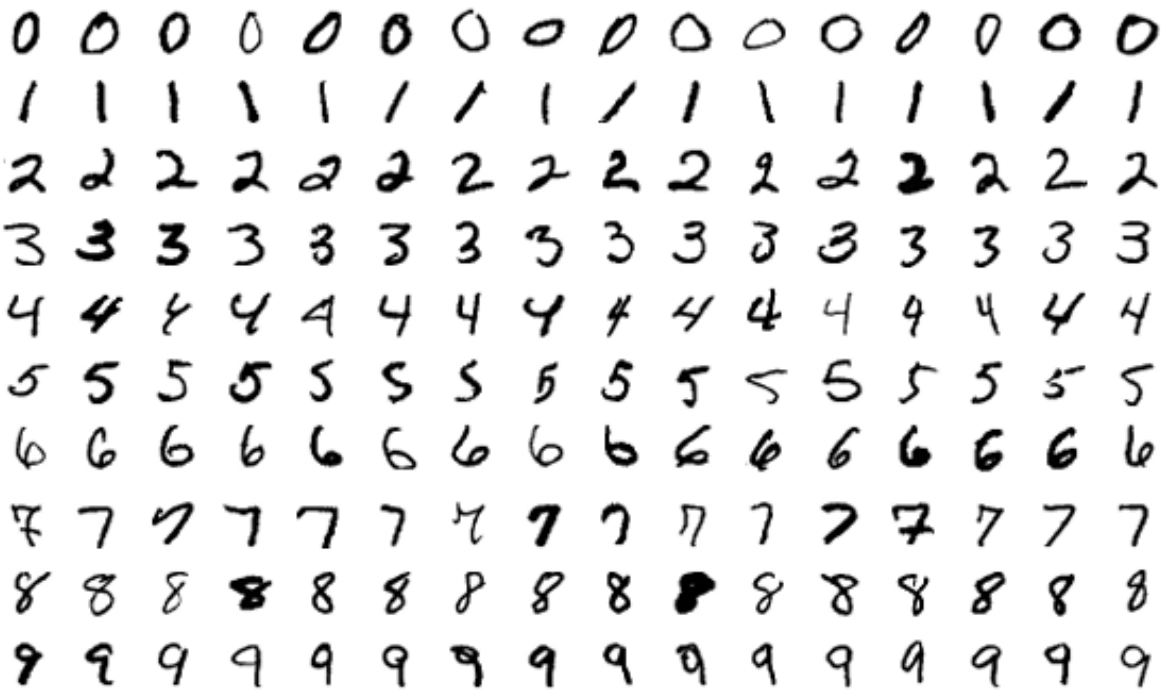
Các thực nghiệm được thiết kế nhằm đánh giá một cách nhất quán hiệu năng của nhiều thuật toán tối ưu khác nhau trong huấn luyện mô hình học sâu. Mỗi mô hình được huấn luyện trong 20 epoch với kích thước batch bằng 128, đồng thời toàn bộ tập dữ liệu, kiến trúc mạng và quy trình tiền xử lý được giữ cố định để đảm bảo tính tái lập và khả năng so sánh công bằng giữa các thuật toán.

Hai nhóm thuật toán được khảo sát: nhóm không thích nghi và nhóm thích nghi. Với các phương pháp không thích nghi, tốc độ học được cố định ở mức 0.01, bao gồm: SGD thuần, SGD kèm Momentum (momentum = 0.9), SGD với Nesterov (momentum = 0.9 và bật cơ chế Nesterov). Ngược lại, các thuật toán thích nghi được cấu hình với tốc độ học nhỏ hơn, 0.001, theo khuyến nghị điển hình trong huấn luyện mạng nơ-ron hiện đại. Nhóm này bao gồm RMSProp, Adagrad, Adam và Yogi và Adadelta thì 1.0. Các siêu tham số nội tại của từng thuật toán (như hệ số tích luỹ gradient hay tham số điều khiển moment) được giữ mặc định theo công bố gốc nhằm phản ánh đúng đặc tính của từng phương pháp.

Trong suốt quá trình huấn luyện, giá trị loss và accuracy trên cả tập train và validation được ghi lại theo từng epoch. Việc này cho phép phân tích toàn diện tốc độ hội tụ, độ ổn định của gradient, và khả năng tổng quát hoá của mô hình dưới từng thuật toán tối ưu khác nhau.

4.2 Tập dữ liệu thực nghiệm

Tập dữ liệu MNIST được sử dụng làm cơ sở để đánh giá hiệu quả của các thuật toán tối ưu trong thực nghiệm. MNIST bao gồm 70,000 ảnh chữ số viết tay, mỗi ảnh có kích thước 28×28, mức xám đơn kênh, và được gán nhãn tương ứng với một trong 10 lớp từ 0 đến 9. Bộ dữ liệu được chia thành 60,000 ảnh cho tập huấn luyện và 10,000 ảnh cho tập kiểm tra, là chuẩn mực phổ biến trong nghiên cứu học sâu nhờ tính đơn giản nhưng vẫn đủ khả năng phản ánh hành vi hội tụ của các thuật toán tối ưu.



Hình 4.1: Minh họa hình ảnh từ bộ dữ liệu thực nghiệm.

Trước khi đưa vào mô hình, dữ liệu được chuẩn hóa về dải giá trị $[0, 1]$ và áp dụng phép biến đổi chuẩn hoá theo thống kê của toàn bộ tập huấn luyện nhằm đảm bảo sự ổn định của quá trình tối ưu. Không áp dụng các kỹ thuật tăng cường dữ liệu để giữ nguyên tính thuần nhất của bộ dữ liệu gốc và tập trung vào việc phân tích sự khác biệt giữa các thuật toán tối ưu.

4.3 Đánh giá kết quả thực nghiệm

Mục tiêu của thực nghiệm là đánh giá khả năng hoạt động và hiệu suất phân loại của các thuật toán tối ưu gồm SGD, SGD kết hợp Momentum, SGD với Nesterov, RMSProp, Adagrad, Adadelta, Adam và Yogi khi áp dụng trên một mô hình học sâu dạng truyền thống. Cụ thể, các thuật toán này được so sánh dựa trên quá trình huấn luyện các mô hình được cài đặt bằng PyTorch.

4.3.1 Multilayer Perceptron- MLP

Mô hình được sử dụng trong thực nghiệm là một mạng MLP nhiều tầng, bao gồm các lớp tuyến tính đan xen với hàm kích hoạt ReLU. Kiến trúc cụ thể của mạng được minh họa trong Hình 4.2, với ba tầng ẩn lần lượt có kích thước 300, 200 và 100 neuron, đầu ra gồm 10 lớp tương ứng với các nhãn của tập MNIST. Tổng số tham số khả huấn luyện của mô hình là 316,810, cho phép đánh giá

rõ ràng mức độ ảnh hưởng của từng thuật toán tối ưu lên quá trình hội tụ của mạng.

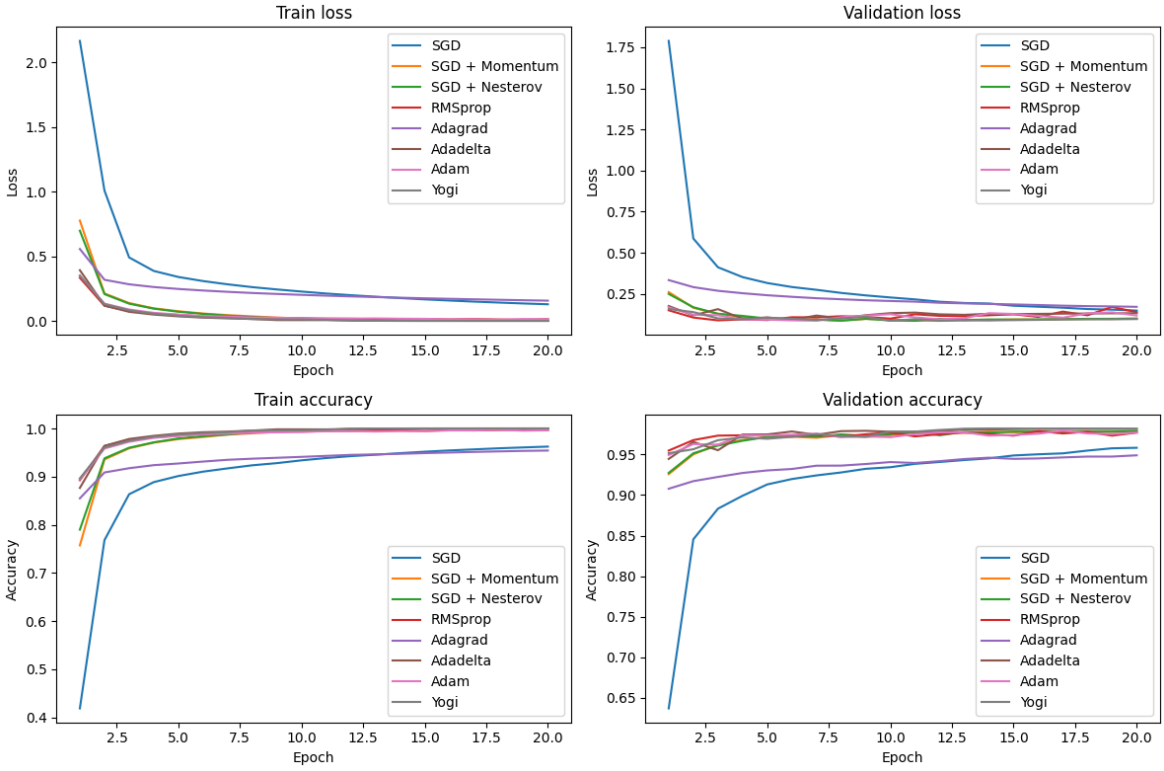
```
==========================================================================================
Layer (type:depth-idx)                   Output Shape              Param #
==========================================================================================
NeuralNet                                [128, 10]                 --
├─Sequential: 1-1                        [128, 10]                 --
│    └─Linear: 2-1                       [128, 300]                235,500
│    └─ReLU: 2-2                         [128, 300]                --
│    └─Linear: 2-3                       [128, 200]                60,200
│    └─ReLU: 2-4                         [128, 200]                --
│    └─Linear: 2-5                       [128, 100]                20,100
│    └─ReLU: 2-6                         [128, 100]                --
│    └─Linear: 2-7                       [128, 10]                 1,010
==========================================================================================
Total params: 316,810
Trainable params: 316,810
Non-trainable params: 0
Total mult-adds (Units.MEGABYTES): 40.55
==========================================================================================
Input size (MB): 0.40
Forward/backward pass size (MB): 0.62
Params size (MB): 1.27
Estimated Total Size (MB): 2.29
==========================================================================================
```

Hình 4.2: Kiến trúc mô hình thực nghiệm MLP.

Với mô hình MLP, chúng tôi huấn luyện trên cùng một tập dữ liệu trong 20 epoch với 8 thuật toán tối ưu. Nhóm thuật toán cơ sở gồm SGD, SGD kèm Momentum, SGD kèm Nesterov, RMSprop và Adagrad; nhóm thuật toán nghiên cứu của đề tài gồm Adadelta, Adam và Yogi (một biến thể của Adam). kết quả được trình bày trong Bảng 4.1 và Hình 4.3

Bảng 4.1: Kết quả thực nghiệm của các thuật toán tối ưu trên mô hình MLP

Optimizer	**Train Loss**	**Train Acc**	**Val Loss**	**Val Acc**	**Test Acc**
SGD	0.1285	0.9623	0.1490	0.9583	0.9580
SGD + Momentum	0.0010	1.0000	0.1006	0.9784	0.9790
SGD + Nesterov	0.0010	1.0000	0.0997	0.9790	0.9775
RMSprop	0.0132	0.9963	0.1390	0.9771	0.9797
Adagrad	0.1566	0.9541	0.1718	0.9491	0.9514
Adadelta	0.0000	1.0000	0.1330	0.9817	0.9819
Adam	0.0126	0.9961	0.1189	0.9771	0.9794
Yogi	0.0001	1.0000	0.0975	0.9818	0.9804



Hình 4.3: Biểu đồ kết quả thực nghiệm các thuật toán tối ưu với mô hình MLP

Ở nhóm cơ sở, SGD thuần cho quỹ đạo hội tụ khá "đều", nhưng tốc độ tăng độ chính xác tương đối chậm: phải tới gần cuối 20 epoch mới đạt Best val Acc ≈ 0.958. Khi bổ sung momentum (SGD + Momentum, SGD + Nesterov), tốc độ hội tụ cải thiện rõ rệt: ngay vài epoch đầu, validation accuracy đã vượt 0.95 và Best val Acc đạt khoảng 0.978–0.979, vượt xa SGD thuần. Trong đó, Nesterov nhỉnh hơn đôi chút so với momentum cổ điển. RMSprop cũng cho hội tụ nhanh và độ chính xác cuối khá cao (Best val Acc ≈ 0.979), nhưng đường *val_loss* dao động lớn hơn ở giai đoạn sau, thể hiện cập nhật tham số có phần kém ổn định hơn. Ngược lại, Adagrad tăng đều và tương đối ổn định, nhưng dừng ở mức thấp hơn các thuật toán khác (Best val Acc ≈ 0.949), phù hợp với đặc trưng bước học giảm dần khiến thuật toán dễ bị "đuối" ở cuối quá trình tối ưu.

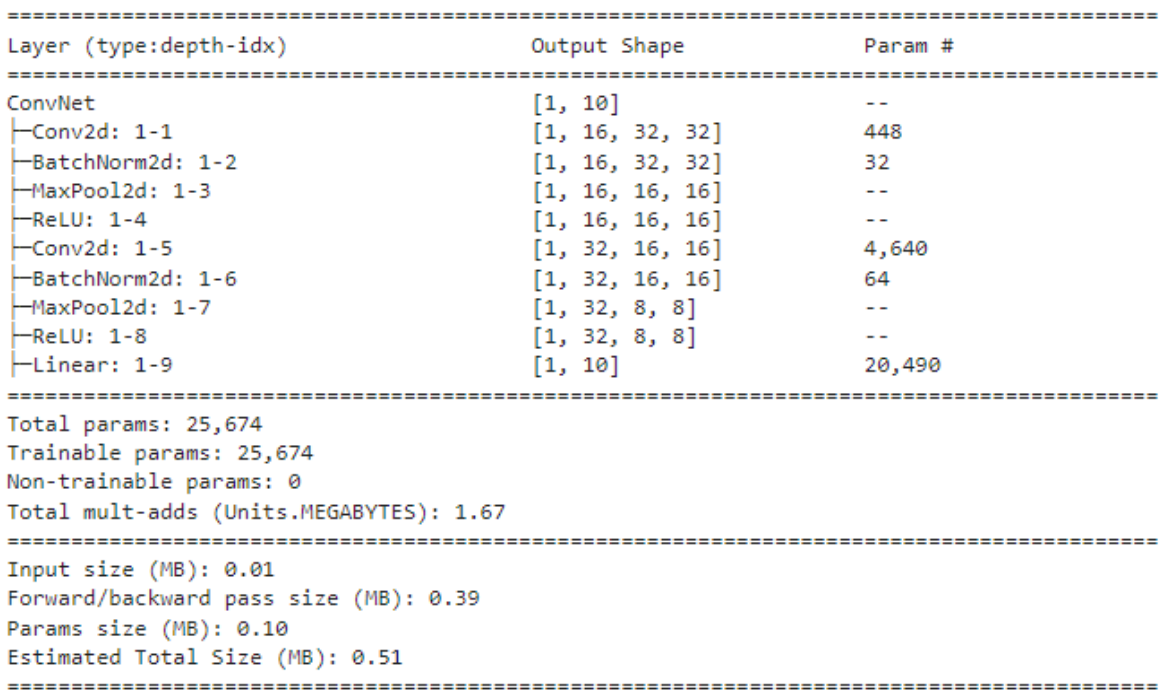
So với nhóm cơ sở, ba thuật toán nghiên cứu (Adadelta, Adam, Yogi) đều đạt mức hiệu năng cao hơn hoặc ít nhất là tương đương các thuật toán tốt nhất trong nhóm cơ sở. Adadelta cho hội tụ rất nhanh: chỉ sau 2–3 epoch, validation accuracy đã đạt trên 0.96 và Best val Acc ≈ 0.9818, cao hơn hầu hết thuật toán cơ sở. Tuy nhiên, *train_loss* giảm về gần 0 trong khi *val_loss* tăng nhẹ ở giai đoạn cuối, cho thấy xu hướng khớp huấn luyện rất mạnh và có dấu hiệu overfitting nhẹ. Adam cũng hội tụ nhanh, là một trong các thuật toán đạt *val_acc* cao sớm, với Best val Acc ≈ 0.9795, tương đương hoặc nhỉnh hơn

Nesterov/RMSprop. Đường *val_loss* của Adam dao động hơn Adadelta, phản ánh đặc trưng cập nhật thích nghi khá "nhạy", nhưng vẫn duy trì được khả năng khái quát hoá tốt. Yogi, với cơ chế điều chỉnh moment bậc hai thận trọng hơn so với Adam, cho kết quả tốt nhất trên MLP: Best val Acc đạt khoảng 0.9822, đồng thời đường *val_acc* ổn định quanh mức 0.98 trong các epoch cuối, chứng tỏ sự cân bằng tốt giữa tốc độ hội tụ và độ ổn định của nghiệm.

Tóm lại, trên mô hình MLP, các thuật toán nhóm nghiên cứu Adadelta, Adam, Yogi đều vượt trội so với SGD thuần và Adagrad, và cạnh tranh hoặc nhỉnh hơn so với các biến thể SGD có momentum và RMSprop. Trong đó, Yogi cho hiệu năng tổng thể tốt nhất, Adadelta hội tụ rất nhanh nhưng có xu hướng khớp huấn luyện mạnh, còn Adam đóng vai trò là lựa chọn cân bằng, dễ dùng, với hiệu năng sát nhóm tốt nhất và ổn định trên nhiều epoch.

4.3.2 Convolutional Neural Networks (CNN)

Hình dưới đây minh họa kiến trúc mạng Convolutional Neural Network (CNN) đơn giản được sử dụng trong thực nghiệm. Mạng bao gồm hai khối tích chập liên tiếp, mỗi khối gồm Conv2d – BatchNorm – ReLU – MaxPool, giúp giảm dần kích thước không gian và tăng cường biểu diễn đặc trưng. Sau khi qua hai tầng gộp, đặc trưng được làm phẳng và đưa vào một lớp tuyến tính để dự đoán đầu ra. Kiến trúc này có tổng cộng 25.674 tham số, đủ nhỏ gọn để huấn luyện nhanh nhưng vẫn đảm bảo khả năng học đặc trưng hiệu quả.



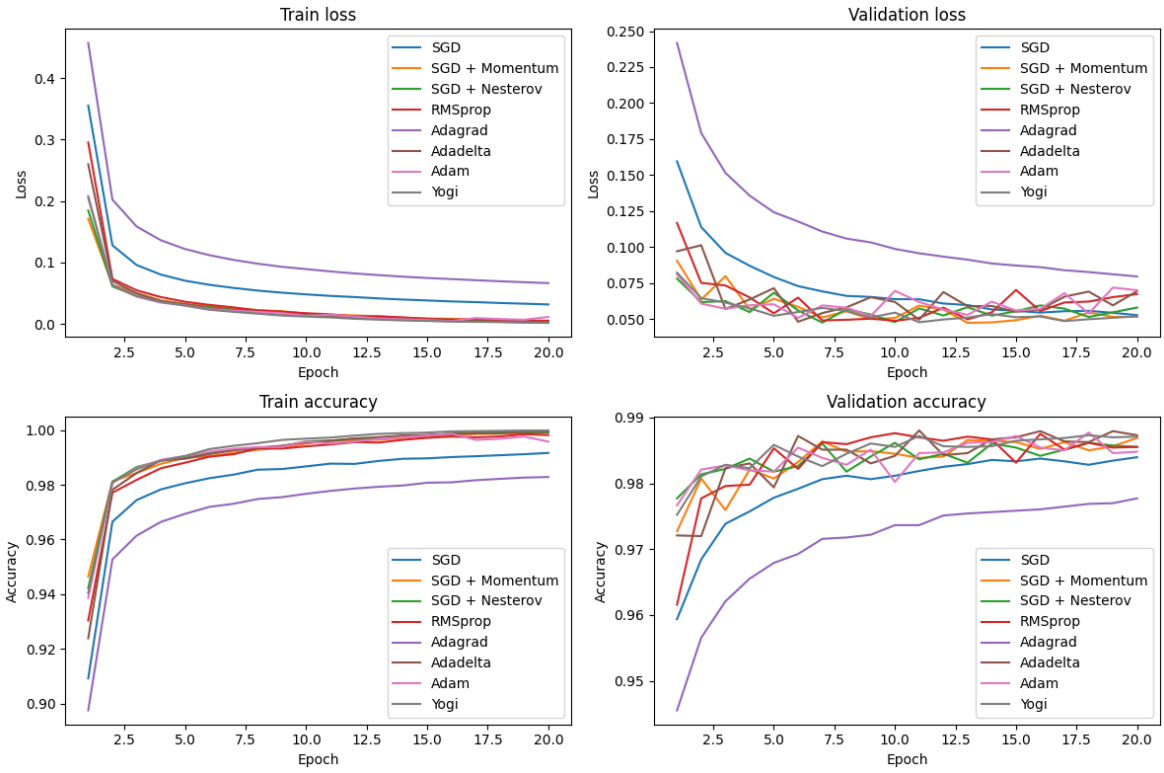
Hình 4.4: Kiến trúc mạng Convolutional Neural Networks đơn giản

Các thuật toán cơ sở trong thực nghiệm bao gồm SGD, SGD + Momentum, SGD + Nesterov, RMSprop và Adagrad, cung cấp một khung tham chiếu về tốc độ hội tụ, độ ổn định và khả năng khái quát hóa. SGD cho quá trình hội tụ chậm hơn nhưng ổn định, đạt $val\_acc = 0.9840$ và $test\_acc = 0.9860$. Momentum và Nesterov cải thiện tốc độ hội tụ rõ rệt, với $test\_acc$ lần lượt là 0.9876 và 0.9889, nhưng đường val_loss vẫn có dao động do biên độ cập nhật lớn. RMSprop đạt hiệu năng cao ($test\_acc = 0.9893$) nhưng dao động validation rõ hơn ở cuối quá trình. Adagrad, ngược lại, thể hiện tốc độ hội tụ chậm và hiệu năng thấp nhất trong nhóm cơ sở ($test\_acc = 0.9810$), phù hợp với đặc trưng *learning rate* giảm nhanh.

Trên nền các kết quả này, ba thuật toán trọng tâm của nhóm nghiên cứu: Adadelta, Adam và Yogi, đều vượt trội hơn hoặc duy trì hiệu năng ngang bằng các thuật toán cơ sở mạnh nhất. Minh họa ở Bảng 4.2 và Hình 4.5.

Bảng 4.2: Kết quả thực nghiệm các thuật toán tối ưu với mô hình CNN

Optimizer	**train_loss**	**train_acc**	**val_loss**	**val_acc**	**test_acc**
SGD	0.0320	0.9917	0.0527	0.9840	0.9860
SGD + Momentum	0.0052	0.9992	0.0515	0.9869	0.9876
SGD + Nesterov	0.0052	0.9991	0.0579	0.9855	0.9889
RMSprop	0.0056	0.9982	0.0673	0.9855	0.9893
Adagrad	0.0666	0.9829	0.0795	0.9777	0.9810
Adadelta	0.0020	0.9995	0.0694	0.9873	0.9896
Adam	0.0117	0.9958	0.0700	0.9848	0.9878
Yogi	0.0024	0.9999	0.0519	0.9871	0.9895



Hình 4.5: Biểu đồ kết quả thực nghiệm các thuật toán tối ưu với mô hình CNN

Adadelta: Adadelta đạt hiệu năng tốt nhất trong toàn bộ thí nghiệm với $test\_acc = 0.9896$ và $val\_acc = 0.9873$, cao hơn tất cả thuật toán cơ sở. *Train_loss* rất thấp (0.0020) và *train_acc* gần tuyệt đối (0.9995) cho thấy quá trình tối ưu diễn ra hiệu quả mà không cần điều chỉnh phức tạp về siêu tham số. Mặc dù *val_loss* có dao động nhẹ, Adadelta vẫn duy trì khả năng khái quát

hóa tốt nhất. Điều này nhấn mạnh ưu điểm lớn nhất của Adadelta: ổn định, hiệu năng cao và không nhạy với lựa chọn *learning rate*.

Adam: Adam đạt *test_acc* = 0.9878, vượt hầu hết thuật toán cơ sở và chỉ kém nhẹ RMSprop và Adadelta. Adam cho tốc độ hội tụ rất nhanh ngay từ các epoch đầu, phản ánh hiệu quả của cơ chế thích nghi dựa trên momen bậc nhất và bậc hai. Tuy *val_loss* dao động nhiều hơn Adadelta, Adam vẫn đảm bảo hiệu năng ổn định và khả năng khái quát hóa cao. Điều này phù hợp với vai trò phổ biến của Adam trong thực tế: tối ưu nhanh, hiệu quả, thích ứng tốt với gradient thay đổi.

Yogi: Yogi: một biến thể của Adam với cơ chế điều chỉnh momen bậc hai ổn định hơn, thể hiện hiệu năng rất cạnh tranh với *test_acc* = 0.9895 và *val_acc* = 0.9871, gần như tương đương Adadelta và vượt trội so với các thuật toán cơ sở. Với *train_acc* = 0.9999 và *train_loss* thấp (0.0024), Yogi cho thấy khả năng tối ưu nhanh và mạnh mẽ giống Adam nhưng giảm dao động validation, mang lại sự ổn định tốt hơn. Điều này khẳng định giá trị của Yogi như một mở rộng hiệu quả của Adam cho các bài toán cần tính ổn định cao hơn.

So sánh với các thuật toán cơ sở, cả ba thuật toán trong phạm vi nghiên cứu: Adadelta, Adam và Yogi đều thể hiện ưu thế rõ rệt về tốc độ hội tụ, mức độ ổn định và khả năng khái quát hóa. Trong đó:

- Adadelta cho hiệu năng tổng thể tốt nhất.
- Adam mang lại tốc độ hội tụ nhanh nhất.
- Yogi cân bằng giữa hiệu năng cao và độ ổn định.

Ba thuật toán này đều vượt trội so với các phương pháp tối ưu chuẩn và là lựa chọn phù hợp cho các mô hình yêu cầu hiệu quả tối ưu hóa cao.

4.3.3 Deep Residual Networks (ResNet-18)

Kiến trúc được sử dụng dựa trên ResNet, trong đó các khối residual cho phép duy trì dòng truyền gradient ổn định khi mạng trở nên sâu hơn. Mô hình gồm lớp tích chập đầu vào và bốn nhóm khối residual có số kênh tăng dần (64–512), tiếp theo là phép tổng hợp trung bình toàn cục và lớp phân loại. Cấu trúc chi tiết được trình bày như sau.

```
==========================================================================
Layer (type:depth-idx)                   Output Shape              Param #
==========================================================================
ResNet                                   [128, 10]                 --
|- Conv2d: 1-1                           [128, 64, 14, 14]         3,136
|- BatchNorm2d: 1-2                      [128, 64, 14, 14]         128
|- ReLU: 1-3                             [128, 64, 14, 14]         --
|- MaxPool2d: 1-4                        [128, 64, 7, 7]           --
|- Sequential: 1-5                       [128, 64, 7, 7]           --
```

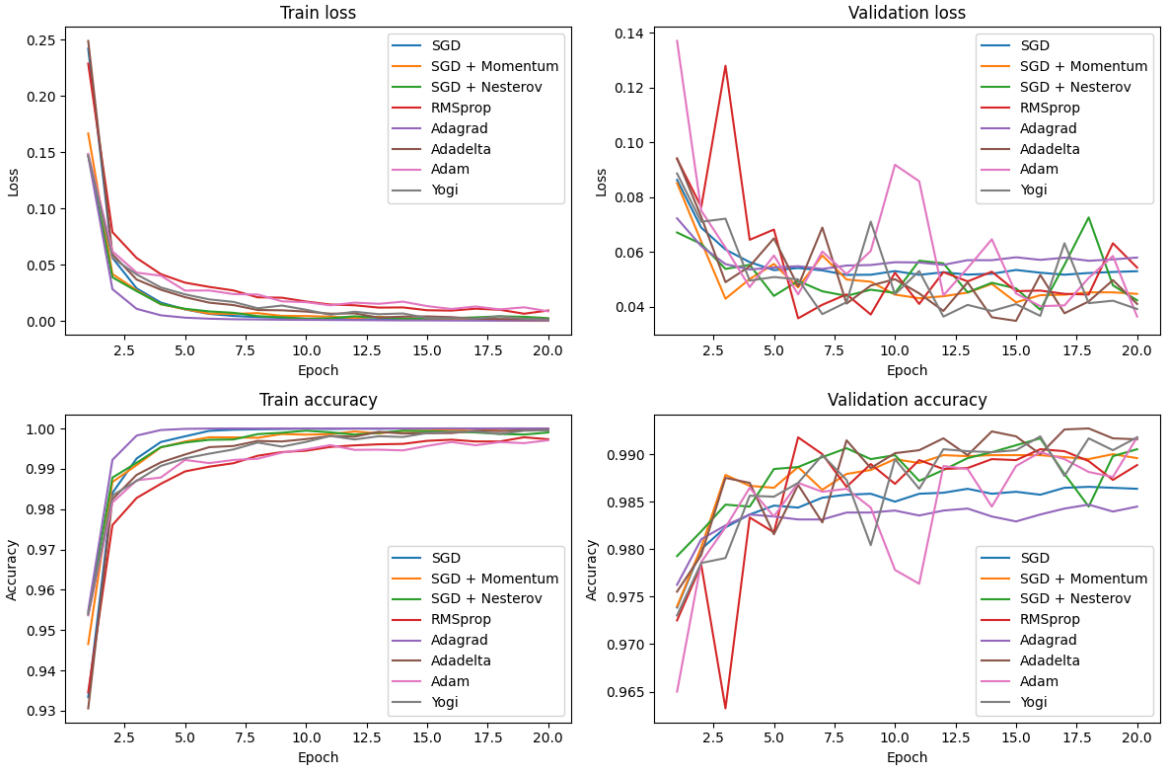
```
|   `- BasicBlock: 2-1                  [128, 64, 7, 7]           --
|       |- Conv2d: 3-1                  [128, 64, 7, 7]           36,864
|       |- BatchNorm2d: 3-2             [128, 64, 7, 7]           128
|       |- ReLU: 3-3                    [128, 64, 7, 7]           --
|       |- Conv2d: 3-4                  [128, 64, 7, 7]           36,864
|       |- BatchNorm2d: 3-5             [128, 64, 7, 7]           128
|       |- Sequential: 3-6              [128, 64, 7, 7]           --
|       `- ReLU: 3-7                    [128, 64, 7, 7]           --
|   `- BasicBlock: 2-2                  [128, 64, 7, 7]           --
|       |- Conv2d: 3-8                  [128, 64, 7, 7]           36,864
|       |- BatchNorm2d: 3-9             [128, 64, 7, 7]           128
|       |- ReLU: 3-10                   [128, 64, 7, 7]           --
|       |- Conv2d: 3-11                 [128, 64, 7, 7]           36,864
|       |- BatchNorm2d: 3-12            [128, 64, 7, 7]           128
|       |- Sequential: 3-13             [128, 64, 7, 7]           --
|       `- ReLU: 3-14                   [128, 64, 7, 7]           --
|- Sequential: 1-6                      [128, 128, 4, 4]          --
|   `- BasicBlock: 2-3                  [128, 128, 4, 4]          --
|       |- Conv2d: 3-15                 [128, 128, 4, 4]          73,728
|       |- BatchNorm2d: 3-16            [128, 128, 4, 4]          256
|       |- ReLU: 3-17                   [128, 128, 4, 4]          --
|       |- Conv2d: 3-18                 [128, 128, 4, 4]          147,456
|       |- BatchNorm2d: 3-19            [128, 128, 4, 4]          256
|       |- Sequential: 3-20             [128, 128, 4, 4]          8,448
|       `- ReLU: 3-21                   [128, 128, 4, 4]          --
|   `- BasicBlock: 2-4                  [128, 128, 4, 4]          --
|       |- Conv2d: 3-22                 [128, 128, 4, 4]          147,456
|       |- BatchNorm2d: 3-23            [128, 128, 4, 4]          256
|       |- ReLU: 3-24                   [128, 128, 4, 4]          --
|       |- Conv2d: 3-25                 [128, 128, 4, 4]          147,456
|       |- BatchNorm2d: 3-26            [128, 128, 4, 4]          256
|       |- Sequential: 3-27             [128, 128, 4, 4]          --
|       `- ReLU: 3-28                   [128, 128, 4, 4]          --
|- Sequential: 1-7                      [128, 256, 2, 2]          --
|   `- BasicBlock: 2-5                  [128, 256, 2, 2]          --
|       |- Conv2d: 3-29                 [128, 256, 2, 2]          294,912
|       |- BatchNorm2d: 3-30            [128, 256, 2, 2]          512
|       |- ReLU: 3-31                   [128, 256, 2, 2]          --
|       |- Conv2d: 3-32                 [128, 256, 2, 2]          589,824
|       |- BatchNorm2d: 3-33            [128, 256, 2, 2]          512
|       |- Sequential: 3-34             [128, 256, 2, 2]          33,280
|       `- ReLU: 3-35                   [128, 256, 2, 2]          --
|   `- BasicBlock: 2-6                  [128, 256, 2, 2]          --
|       |- Conv2d: 3-36                 [128, 256, 2, 2]          589,824
|       |- BatchNorm2d: 3-37            [128, 256, 2, 2]          512
|       |- ReLU: 3-38                   [128, 256, 2, 2]          --
|       |- Conv2d: 3-39                 [128, 256, 2, 2]          589,824
|       |- BatchNorm2d: 3-40            [128, 256, 2, 2]          512
|       |- Sequential: 3-41             [128, 256, 2, 2]          --
|       `- ReLU: 3-42                   [128, 256, 2, 2]          --
|- Sequential: 1-8                      [128, 512, 1, 1]          --
|   `- BasicBlock: 2-7                  [128, 512, 1, 1]          --
|       |- Conv2d: 3-43                 [128, 512, 1, 1]          1,179,648
|       |- BatchNorm2d: 3-44            [128, 512, 1, 1]          1,024
|       |- ReLU: 3-45                   [128, 512, 1, 1]          --
|       |- Conv2d: 3-46                 [128, 512, 1, 1]          2,359,296
```

```
|       |- BatchNorm2d: 3-47            [128, 512, 1, 1]          1,024
|       |- Sequential: 3-48             [128, 512, 1, 1]          132,096
|       `- ReLU: 3-49                  [128, 512, 1, 1]          --
|   `- BasicBlock: 2-8                  [128, 512, 1, 1]          --
|       |- Conv2d: 3-50                 [128, 512, 1, 1]          2,359,296
|       |- BatchNorm2d: 3-51            [128, 512, 1, 1]          1,024
|       |- ReLU: 3-52                   [128, 512, 1, 1]          --
|       |- Conv2d: 3-53                 [128, 512, 1, 1]          2,359,296
|       |- BatchNorm2d: 3-54            [128, 512, 1, 1]          1,024
|       |- Sequential: 3-55             [128, 512, 1, 1]          --
|       `- ReLU: 3-56                  [128, 512, 1, 1]          --
|- AdaptiveAvgPool2d: 1-9               [128, 512, 1, 1]          --
|- Linear: 1-10                         [128, 10]                 5,130
==========================================================================================
Total params: 11,175,370
Trainable params: 11,175,370
Non-trainable params: 0
Total mult-adds (Units.GIGABYTES): 4.23
==========================================================================================
Input size (MB): 0.40
Forward/backward pass size (MB): 88.09
Params size (MB): 44.70
Estimated Total Size (MB): 133.19
==========================================================================================
```

Kết quả thực nghiệm được thể hiện ở Bảng 4.3 và Hình 4.6

Bảng 4.3: Kết quả thực nghiệm của các thuật toán tối ưu với mô hình ResNet-18

Optimizer	**Train Loss**	**Train Acc**	**Val Loss**	**Val Acc**	**Test Acc**
SGD	0.0007	1.0000	0.0530	0.9864	0.9870
SGD + Momentum	0.0002	0.9999	0.0448	0.9896	0.9929
SGD + Nesterov	0.0026	0.9990	0.0424	0.9905	0.9919
RMSprop	0.0094	0.9973	0.0544	0.9889	0.9923
Adagrad	0.0005	1.0000	0.0580	0.9845	0.9846
Adadelta	0.0017	0.9996	0.0411	0.9916	0.9936
Adam	0.0087	0.9971	0.0365	0.9918	0.9916
Yogi	0.0013	0.9996	0.0393	0.9918	0.9928



Hình 4.6: Biểu đồ kết quả thực nghiệm các thuật toán tối ưu với mô hình ResNet-18

Kết quả thực nghiệm cho thấy tất cả các thuật toán đều hội tụ rất nhanh và đạt độ chính xác cao trên cả tập huấn luyện và validation. Các phương pháp dựa trên SGD (SGD, SGD + Momentum, SGD + Nesterov) đạt train_acc xấp xỉ 1.0 chỉ sau vài epoch; trong đó, Momentum và Nesterov cải thiện rõ rệt tốc độ giảm loss và đạt best val_acc lần lượt là 0.99000 và 0.99167, cao hơn so với SGD thuần (0.98656). Nhóm thuật toán thích nghi (RMSprop, Adagrad, Adadelta, Adam, Yogi) tiếp tục cho hiệu năng cạnh tranh: Adadelta đạt best val_acc cao nhất 0.99271, trong khi RMSprop, Adam và Yogi đều xấp xỉ 0.9917–0.9919, vượt nhẹ các biến thể SGD có gia tốc. Ngược lại, Adagrad mặc dù hội tụ nhanh và rất ổn định nhưng dừng ở mức val_acc = 0.98469, thấp hơn các thuật toán còn lại. Nhìn chung, so với các thuật toán cơ sở, Adadelta, Adam và Yogi đều cho thấy khả năng tối ưu tốt, hội tụ nhanh và duy trì hiệu năng validation vượt trội hoặc tương đương nhóm phương pháp tốt nhất trong thí nghiệm.

4.3.4 VGG-16

Mô hình được xây dựng dựa trên một phiên bản rút gọn của kiến trúc VGG, gồm các tầng tích chập kết hợp BatchNorm và ReLU nhằm trích xuất đặc trưng ổn định và hiệu quả. Kích thước đặc trưng được giảm dần thông qua các tầng gộp, trước khi đưa vào các tầng kết nối đầy đủ để thực hiện phân loại. Kiến trúc

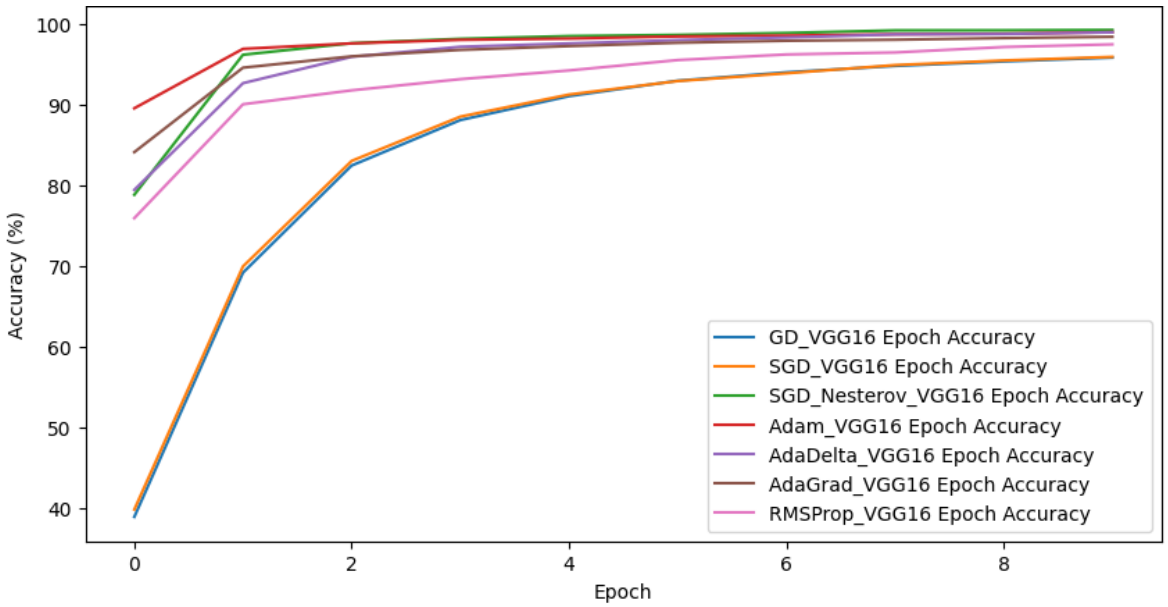
này giữ được tính đơn giản nhưng vẫn đủ sâu để học các đặc trưng quan trọng của dữ liệu.

```
==========================================================================
Layer (type:depth-idx)                   Output Shape              Param #
==========================================================================
VGG_MNIST                                [1, 10]                   --
|- Conv2d: 1-1                           [1, 3, 28, 28]            6
|- Sequential: 1-2                       [1, 256, 3, 3]            --
|    |- Conv2d: 2-1                      [1, 64, 28, 28]           1,792
|    |- BatchNorm2d: 2-2                 [1, 64, 28, 28]           128
|    |- ReLU: 2-3                        [1, 64, 28, 28]           --
|    |- Conv2d: 2-4                      [1, 64, 28, 28]           36,928
|    |- BatchNorm2d: 2-5                 [1, 64, 28, 28]           128
|    |- ReLU: 2-6                        [1, 64, 28, 28]           --
|    |- MaxPool2d: 2-7                   [1, 64, 14, 14]           --
|    |- Conv2d: 2-8                      [1, 128, 14, 14]          73,856
|    |- BatchNorm2d: 2-9                 [1, 128, 14, 14]          256
|    |- ReLU: 2-10                       [1, 128, 14, 14]          --
|    |- Conv2d: 2-11                     [1, 128, 14, 14]          147,584
|    |- BatchNorm2d: 2-12                [1, 128, 14, 14]          256
|    |- ReLU: 2-13                       [1, 128, 14, 14]          --
|    |- MaxPool2d: 2-14                  [1, 128, 7, 7]            --
|    |- Conv2d: 2-15                     [1, 256, 7, 7]            295,168
|    |- BatchNorm2d: 2-16                [1, 256, 7, 7]            512
|    |- ReLU: 2-17                       [1, 256, 7, 7]            --
|    |- Conv2d: 2-18                     [1, 256, 7, 7]            590,080
|    |- BatchNorm2d: 2-19                [1, 256, 7, 7]            512
|    |- ReLU: 2-20                       [1, 256, 7, 7]            --
|    `- MaxPool2d: 2-21                  [1, 256, 3, 3]            --
|- Sequential: 1-3                       [1, 10]                   --
|    |- Linear: 2-22                     [1, 128]                  295,040
|    |- ReLU: 2-23                       [1, 128]                  --
|    |- Dropout: 2-24                    [1, 128]                  --
|    |- Linear: 2-25                     [1, 64]                   8,256
|    |- ReLU: 2-26                       [1, 64]                   --
|    |- Dropout: 2-27                    [1, 64]                   --
|    `- Linear: 2-28                     [1, 10]                   650
==========================================================================
Total params: 1,451,152
Trainable params: 1,451,152
Non-trainable params: 0
Total mult-adds (Units.MEGABYTES): 117.45
==========================================================================
Input size (MB): 0.00
Forward/backward pass size (MB): 2.83
Params size (MB): 5.80
Estimated Total Size (MB): 8.64
==========================================================================
```

Kết quả thực nghiệm được trình bày trong Bảng 4.4 và Hình 4.7

Bảng 4.4: So sánh các thuật toán tối ưu trên mô hình VGG-MNIST

Optimizer	**Train Loss**	**Train Acc**	**Val Loss**	**Val Acc**	**Test Acc**
AdaDelta	0.0081	99.8063	0.0482	99.3083	99.45
AdaGrad	0.0059	99.8542	0.0267	99.3083	99.29
Adam	0.0116	99.7208	0.0507	99.2167	99.39
GD	0.0403	98.8333	0.0480	98.6000	98.80
SGD	0.0416	98.8063	0.0486	98.5667	98.70
SGD + Nesterov	0.0073	99.7521	0.0371	99.1667	99.30
RMSProp	0.0140	99.6479	0.0571	99.1833	99.26



Hình 4.7: Biểu đồ kết quả thực nghiệm các thuật toán tối ưu với mô hình VGG-16

Sau quá trình huấn luyện và đánh giá chuyên sâu trên bộ dữ liệu MNIST, kết quả cho thấy rằng:

AdaDelta liên tục thể hiện hiệu suất mạnh mẽ. Trên VGG-16, nó đạt hội tụ xuất sắc với độ chính xác kiểm tra cuối cùng là 99.45%. Cơ chế tốc độ học thích ứng giúp AdaDelta giữ ổn định và giá trị mất mát thấp, khiến nó trở thành một trong những bộ tối ưu đáng tin cậy nhất trong nghiên cứu này.

AdaGrad cho thấy sự tương phản rõ rệt. Trên VGG-16, nó hoạt động rất tốt, đạt độ chính xác kiểm tra 99.29%. Điều này cho thấy xu hướng của AdaGrad trong việc giảm tốc độ học hiệu quả quá nhanh theo thời gian, có thể cản trở hiệu suất trong các lần huấn luyện dài, mặc dù nó nổi bật trong các mô hình

tích chập sâu.

Adam mang lại kết quả ổn định và cao, với độ chính xác kiểm tra 99.39% trên VGG-16. Cơ chế ước lượng mô-men thích ứng giúp Adam hội tụ nhanh chóng và ổn định. Nhìn chung, Adam vẫn là một bộ tối ưu đa dụng mạnh mẽ, đặc biệt hiệu quả trong các kiến trúc phức tạp.

Gradient Descent (GD) thuần túy đạt mức chấp nhận được trên VGG-16 (98.80%). Không có tính thích ứng hay động lượng, GD gặp khó khăn trong việc hội tụ ổn định, nhấn mạnh những hạn chế của nó khi áp dụng cho mạng nơ-ron hiện đại mà không có cải tiến bổ sung.

SGD thuần túy phản ánh hành vi của GD: chấp nhận được trên VGG-16 (98.70%). Việc thiếu động lượng hoặc cơ chế thích ứng khiến nó không ổn định, cho thấy SGD thuần túy không đủ để đảm bảo hội tụ đáng tin cậy trong thiết lập thí nghiệm này.

SGD với động lượng Nesterov đã cải thiện hiệu suất rõ rệt. Trên VGG-16, nó đạt 99.30%, vượt xa SGD thuần túy. Điều này chứng minh vai trò quan trọng của động lượng trong việc ổn định cập nhật và tăng tốc hội tụ.

RMSProp chứng tỏ là một trong những bộ tối ưu nhất quán nhất. Nó đạt độ chính xác kiểm tra 99.26% trên VGG, với giá trị mất mát cuối cùng thấp. Bằng cách duy trì trung bình động của gradient bình phương, RMSProp cân bằng giữa tính thích ứng và sự ổn định, khiến nó trở thành lựa chọn mạnh mẽ trên nhiều kiến trúc.

Chương 5.

KẾT LUẬN VÀ HƯỚNG PHÁT TRIỂN

5.1 Kết luận

Trong nghiên cứu này, nhiều thuật toán tối ưu khác nhau đã được khảo sát trên các mô hình học sâu để đánh giá đặc tính hội tụ, độ ổn định và khả năng khái quát hóa. Hai nhóm thuật toán chính được xem xét gồm: (i) nhóm thuật toán dựa trên gradient truyền thống như SGD, SGD kèm Momentum và Nesterov; và (ii) nhóm thuật toán thích nghi điều chỉnh tốc độ học như RMSprop, Adagrad, Adadelta, Adam và Yogi. Kết quả thực nghiệm cho thấy mỗi thuật toán đều thể hiện những đặc trưng riêng biệt phản ánh cơ chế cập nhật gradient của chúng. Sự khác biệt này không chỉ ảnh hưởng đến tốc độ huấn luyện mà còn tác động trực tiếp đến chất lượng mô hình và độ tin cậy của quá trình tối ưu hóa.

Nhìn chung, các thuật toán mở rộng từ SGD cải thiện đáng kể khả năng hội tụ so với SGD thuần túy, đặc biệt trong việc giảm dao động của gradient. Trong khi đó, các thuật toán thích nghi lại cho thấy ưu thế rõ rệt về tốc độ huấn luyện và khả năng thích ứng với các miền tham số có đặc tính khác nhau. Những kết luận này góp phần làm rõ vai trò của từng thuật toán trong thực hành và cung cấp định hướng lựa chọn tối ưu hóa theo yêu cầu của mô hình và bài toán cụ thể.

5.2 So sánh và phân tích các thuật toán nghiên cứu

Trong số các thuật toán thuộc nhóm thích nghi, nghiên cứu tập trung vào ba thuật toán tiêu biểu gồm Adam, Adadelta và Yogi. Mỗi thuật toán được thiết kế nhằm giải quyết một số hạn chế tồn tại trong các phương pháp tối ưu truyền thống.

Adam: kết hợp cơ chế Momentum và điều chỉnh tốc độ học theo từng tham số, nhờ đó đạt tốc độ hội tụ nhanh và ít phụ thuộc vào lựa chọn siêu tham số ban đầu. Tuy nhiên, Adam có xu hướng hội tụ vào các điểm tối ưu không ổn định và có thể dẫn đến giảm khả năng tổng quát hóa nếu không được điều chỉnh cẩn thận.

Adadelta: được xây dựng để khắc phục việc tốc độ học suy giảm quá nhanh trong Adagrad, thông qua việc sử dụng trung bình hàm mũ của bình phương

gradient. Ưu điểm lớn nhất của thuật toán là không yêu cầu đặt tốc độ học ban đầu, giúp đơn giản hóa quy trình tối ưu. Mặc dù vậy, tốc độ hội tụ của Adadelta có thể chậm hơn trong giai đoạn đầu huấn luyện so với Adam hoặc Yogi.

Yogi: một biến thể của Adam, được phát triển nhằm giải quyết vấn đề tích lũy mô-men quá mức — một hiện tượng có thể khiến Adam kém ổn định trong một số bối cảnh. Bằng cách kiểm soát chặt chẽ hơn sự thay đổi của Momentum, Yogi đạt được sự cân bằng tốt giữa tốc độ hội tụ và ổn định, đặc biệt trong các bài toán có gradient dao động mạnh hoặc dữ liệu nhiều nhiễu.

Tổng hợp lại, nhóm thuật toán nghiên cứu đều mang tính tự điều chỉnh và thể hiện khả năng thích nghi cao hơn các phương pháp truyền thống. Tuy nhiên, mức độ ổn định và khả năng tổng quát hóa có sự khác biệt, cho thấy tính phù hợp của mỗi thuật toán phụ thuộc vào kiến trúc mô hình, đặc điểm dữ liệu và mục tiêu tối ưu hóa.

5.3 Gợi ý lựa chọn thuật toán theo bối cảnh ứng dụng

Việc lựa chọn thuật toán tối ưu không nên mang tính duy ý chí mà phải dựa trên bối cảnh ứng dụng cụ thể. Từ những phân tích tổng quát, có thể đưa ra các định hướng như sau:

- **Bài toán đơn giản, gradient ổn định:** Các thuật toán dựa trên SGD, đặc biệt là Momentum và Nesterov, thường cho kết quả ổn định và khả năng khái quát hóa tốt, đồng thời chi phí tính toán thấp.
- **Mô hình sâu hoặc tham số lớn:** Adam hoặc RMSprop thường là lựa chọn phù hợp vì khả năng điều chỉnh tốc độ học tự động và hội tụ nhanh.
- **Dữ liệu có nhiều nhiễu hoặc gradient dao động mạnh:** Yogi cho thấy tính ổn định cao và tránh được hiện tượng phân kỳ nhờ cơ chế điều chỉnh Momen chặt chẽ.
- **Bài toán yêu cầu tối thiểu hóa tác động của siêu tham số:** Adadelta là lựa chọn hợp lý vì không yêu cầu thiết lập tốc độ học ban đầu và vẫn duy trì hiệu năng tốt.

Những gợi ý này mang tính định hướng và có thể được điều chỉnh dựa trên thực nghiệm cụ thể của từng mô hình.

5.4 Hướng phát triển của nghiên cứu

Hướng phát triển tiếp theo của nghiên cứu có thể mở rộng theo nhiều phương diện. Trước hết, việc đánh giá các thuật toán tối ưu trên những kiến trúc phức tạp hơn như ResNet, EfficientNet hoặc Transformer sẽ cung cấp cái nhìn toàn

diện hơn về khả năng tổng quát của các thuật toán này trong môi trường có số lượng tham số lớn và landscape phức tạp hơn. Ngoài ra, việc kết hợp tối ưu hóa siêu tham số tự động, chẳng hạn như Bayesian Optimization hoặc Hyperband, hứa hẹn giúp cải thiện hiệu năng và tính ổn định của những thuật toán thích nghi. Đối với Adam và Yogi, việc lựa chọn các tham số β_1, β_2 hoặc ϵ có thể ảnh hưởng đáng kể đến kết quả, do đó các phương pháp tối ưu hóa siêu tham số sẽ đóng vai trò quan trọng.

Một hướng khác là nghiên cứu sâu hơn về đặc tính hình học của không gian hàm mất mát, nhằm hiểu rõ hơn cơ chế hội tụ của từng thuật toán và lý giải nguyên nhân dẫn đến sự khác biệt về hiệu năng. Ngoài ra, việc kết hợp các phương pháp regularization tiên tiến cũng là một hướng triển vọng để tăng khả năng khái quát hóa của mô hình trong các bài toán yêu cầu độ chính xác cao.

Cuối cùng, dựa trên ưu điểm của Yogi, có thể xem xét thiết kế các biến thể mới kết hợp lợi thế của nhiều thuật toán khác nhau, nhằm tạo ra bộ tối ưu hóa có tốc độ hội tụ nhanh, ổn định cao và khả năng tổng quát hóa tốt hơn.

5.5 Tổng kết

Tóm lại, nghiên cứu đã chỉ ra rằng các thuật toán tối ưu không chỉ khác nhau ở tốc độ hội tụ mà còn khác biệt đáng kể về độ ổn định và khả năng khái quát hóa. Nhóm thuật toán dựa trên SGD phù hợp với các bài toán yêu cầu tính ổn định, trong khi nhóm thích nghi lại ưu thế trong các mô hình lớn hoặc dữ liệu phức tạp. Các thuật toán nghiên cứu như Adam, Adadelta và Yogi mang lại nhiều tiềm năng, đặc biệt Yogi cho thấy sự cân bằng tốt nhất giữa tốc độ hội tụ và độ ổn định. Những kết luận rút ra từ nghiên cứu này góp phần định hướng việc lựa chọn thuật toán tối ưu phù hợp trong thực tiễn và mở ra nhiều hướng phát triển trong tương lai.

TÀI LIỆU THAM KHẢO

[1] L. Bottou, "Large-scale machine learning with stochastic gradient descent," in *Proceedings of COMPSTAT'2010*, pp. 177–186, Springer, 2010.

[2] J. Duchi, E. Hazan, and Y. Singer, "Adaptive subgradient methods for online learning and stochastic optimization," *Journal of Machine Learning Research*, vol. 12, pp. 2121–2159, 2011.

[3] T. Tieleman and G. Hinton, "Lecture 6.5 — rmsprop," tech. rep., COURSERA: Neural Networks for Machine Learning, 2012. http://www.cs.toronto.edu/~tijmen/csc321/slides/lecture_slides_lec6.pdf.

[4] M. D. Zeiler, "Adadelta: An adaptive learning rate method," *arXiv preprint arXiv:1212.5701*, 2012.

[5] D. P. Kingma and J. Ba, "Adam: A method for stochastic optimization," in *Proceedings of the 3rd International Conference on Learning Representations (ICLR)*, 2015.

[6] M. Zaheer, S. J. Reddi, D. Sachan, S. Kale, and S. Kumar, "Adaptive methods for nonconvex optimization," in *Advances in Neural Information Processing Systems (NeurIPS)*, 2018.

[7] S. Ruder, "An overview of gradient descent optimization algorithms," *arXiv preprint arXiv:1609.04747*, 2016.

[8] Y. Nesterov, "A method for solving the convex programming problem with convergence rate $o(1/k^2)$," *Soviet Mathematics Doklady*, vol. 27, no. 2, pp. 372–376, 1983.

[9] I. Sutskever, J. Martens, G. Dahl, and G. Hinton, "On the importance of initialization and momentum in deep learning," in *International Conference on Machine Learning (ICML)*, pp. 1139–1147, 2013.

[10] J. Pennington, R. Socher, and C. D. Manning, "Glove: Global vectors for word representation," in *Proceedings of the 2014 Conference on Empirical Methods in Natural Language Processing (EMNLP)*, pp. 1532–1543, 2014.

[11] D. P. Kingma and J. Ba, "Adam: A method for stochastic optimization," *arXiv preprint arXiv:1412.6980*, 2014.