**8. Optimize for Training Deep Models**

* **Intro**

Tối ưu được nói đến trong deep learning algorithms trong nhiều bối cảnh. Chúng tôi thường sử dụng tối ưu hóa để viết chứng minh hoặc để thiết kế thuật toán. Có nhiểu vấn đề tối ưu trong deep learning mà khó nhất là huấn luyện mạng neural. Chúng ta có thể đầu tư hàng ngày, hàng tháng trên hàng trăm máy chỉ để giải quyết 1 trường hợp huấn luyện mạng neural. Chính vì vấn đề quan trọng và rất tốn kém nên 1 chuyên gia tập hợp các kỹ thuật tối ưu đã được phát triển để giải quyết.

Chương này trình bày các kỹ thuật tối ưu hóa này để đào tạo mạng neural.

Chương này tập trung vào 1 trường hợp tối ưu hóa cụ thể: Tìm của mạng neural để làm giảm cost function**J**( trong đó bao gồm một phép đo hiệu suất(performance measure) được đánh giá trên toàn bộ tập train cũng như các phương pháp chính quy hóa (regularization).

Lộ trình chương này:

* Tối ưu hóa được sử dụng như 1 thuật toán đào tạo trong nhiệm vụ học máy và Tối ưu hóa thuần túy khác nhau ntn ???
* Một số thách thức trong việc tối ưu hóa mạng neural ??
* Một số thuật toán thực tế : gồm thuật toán và cách khởi tạo tham số
* Các thuật toán nâng vào, cách điều chỉnh tốc độ học (learning rate), tận dụng thông tin từ đạo hàm cấp 2 của cost function.
* Tổng kết : Đánh giá về một số chiến lược tối ưu hóa bằng cách kết hợp các thuật toán đơn giản thành các thủ tục cấp cao hơn.

8.1 Tối ưu hóa trong ML khác Tối ưu hóa thuần túy như thế nào ???

ML thường hoạt động một cách gián tiếp(indirect). Tức là khi có 1 phép đo hiệu suất **P** được xác định với tập test và nó khó để tối ưu trực tiếp và chúng tôi sẽ tối ưu hóa **P** một cách gián tiếp.

* Chú thích :
  + Tối ưu hóa trực tiếp(direct): Trực tiếp tối ưu phép đo hiệu suất P( có thể là phân phối xác suất) bằng các phương pháp như Maximum LikeHood (MLE) hay MAP.
  + Tối ưu hóa gián tiếp(indirect): Giảm hàm cost **J**(bằng cách cập nhật các tham số(weights,bias) bằng các phương pháp như Gradient Descent . Qua đó hy vọng cải thiện được **P**

Thì điều này trái ngược với tối ưu hóa thuần túy(Pure Optimize) – *Giảm thiểu tối đa hàm* ***J****( là mục đích chính*

Thông thường cost function thường được viết dưới dạng trung bình trên tập train:

<=> trên training set

Trong đó:

* + **E**  là kỳ vọng(expected) của hàm loss L(hay nói cách khác là tính trung bình)
  + là phân phối xác suất trên tập mẫu(sample)/train – *empirical distribution*

Tuy nhiên chúng tôi thường muốn tối thiểu hóa hàm mục tiêu tương ứng trong đó kỳ vọng được tính trên – *data generating distribution* tức là trên tập lớn(population)/train+test hoặc là cả tập dữ liệu mới chứ không p chỉ trên tập hữu hạn:

8.1.1 Empirical Risk Minimization

Mục tiêu của ML algorithm là giảm expected generalization error . Đại lượng này được gọi là risk. Nhấn mạnh ở đây là expected error trên tập dữ liệu thực sự.

Ở đây nếu chúng ta biết thì bài toán trở thành pure optimize. Tuy nhiên trong thực tế, tập dữ liệu thực sự quá lớn và thường thì chưa biết được.

Còn khi chúng ta (thường ) biết tập training set là một mẫu(sample) thì chúng ta sẽ có bài toán machine learning

Cách để chuyển 1 vấn đề ML thành Optimize là tối thiểu hóa hàm loss trên tập train. Nói cách khác chúng ta sẽ thay thế true distribution -> empirical distribution trên tập train.

Bây h chúng ta có empirical risk:

=

Cách đào tạo dựa trên giảm thiểu lỗi trên tập train này được biết đến là đào tạo theo kinh nghiệm ( empirical risk minimization). Thay vì tối ưu hóa risk 1 cách trực tiếp thì lựa chọn tối ưu risk thực nghiệm với hy vọng nó sẽ tốt.

Tuy nhiên tối ưu gián tiếp theo kinh nghiệm thường bị overfitting. Model với độ phức tạp lớn có thể ghi nhớ tập train nên trong một số trường hợp thì empirical risk minimization không phù hợp, chưa kể một số hàm loss không có đạo hàm khi sử dụng các thuật toán như gradient descent

8.1.2 Surrogate Loss Function and Early Stopping

Surrogate Loss Function

Đôi khi loss function chúng ta thực sự quan tâm không thể (hoặc khó) để tối ưu hiệu quả.

* Ví dụ: Trong bài toán binary classification thường chúng ta sẽ sử dụng nhưng hàm loss kiểu như MSE hay Cross-Entropy. Tuy nhiên trong một số trường hợp, việc tính toán các hàm mất mát trên có thể phức tạp hoặc tốn kém

Chính vì thế, thay vào đó chúng ta sử dụng 1 surrogate loss func ( hàm mất mát thay thế) để xấp xỉ hàm mất mát gốc và dễ tính toán hơn

* Ví dụ: Tương tự trong bài toán binary classification chúng ta có thể sử dụng hàm hinge loss để làm surrogate loss function.

Early Stopping

Một điểm khác biệt rất quan trọng giữa optimize in generate và optimize khi chúng ta sử dụng cho các thuật toán đào tạo là các thuật toán đào tạo thường không dừng lại ở local minimum.

Các điều kiện dừng sớm thường dựa trên true loss function và được thiết kế dừng lại bất kì khi nào xảy ra overfitting.

Quá trình train thường dừng lại khi mà đạo hàm surrogate loss vẫn còn lớn, rất khác với optimize nói chung – được coi là hội tụ khi đạo hàm rất nhỏ

8.1.3 Batch and Minibatch Algorithms

Trong ML, hàm mục tiêu thường được tách thành tổng các thành phần tương ứng với các sample của training set

Trong quá trình huấn luyện, các thuật toán tối ưu thường tính toán và update các tham số dựa trên kỳ vọng của loss func được ước tính bằng cách chỉ sử dụng các thành phần trong loss func đầy đủ. Tức là thay vì tính toán trên toàn bộ training set , các thuật toán tối ưu thường chỉ sử dụng một số sample được chọn ngẫu nhiên của training set sau đó lấy trung bình cộng. Việc này giúp giảm bớt tính toán và đơn giản hóa quá trình tối ưu.

Tuy nhiên việc chọn ngẫu nhiên sample để ước tính có thể gây ra sai số so với tính toán trên toàn bộ training set.

Nhớ lại về standard error of the mean khi ước tính từ n sample là trong đó là true standard error. Hiểu đơn giản nếu ta lấy 100 sample và 10000 sample thì yêu cầu tính toán tăng gáp 100 lần nhưng giảm standard error chỉ là 10. Vì thế có thể thấy rằng hầu hết các thuật toán tối ưu hội tụ nhanh hơn nhiều nếu có thể ước tính gần đúng các gradient chứ không cần phải tính toán gradient một cách chính xác.

Một yếu tố khác thúc đẩy ước tính xấp xỉ của gradient thay vì tính chính xác là ngoài về tốc độ tính toán thì còn là vấn đề dư thừa trong training set. Trong trường hợp xấu nhất thì cả m sample đều là bản sao của nhau

Các thuật toán tối ưu sử dụng toàn bộ training set được gọi là batch. Thuật ngữ này có thể hơi lạ vì từ batch thường được sử dụng để mô tả các thuật toán minibatch. Người ta thường sử dụng batch\_size để mô tả kích thước của 1 minibatch

Ít nhất thì là thuật toán stocha khi chỉ sử dụng 1 sample vào mỗi lần cập nhật tham số. Nhiều nhất là thuật toán online, lấy sample từ 1 luồng dữ liệu không ngừng và tập train ko có kích thước cố định, thường là trong các bài toán real-time.

Phânf lớn các thuật toán DL thì ở đâu đó giữa, nhiêu hơn 1 nhưng không p toàn bộ tập train.

Chúng được gọi truyền thống là : minibatch hay minibatch stochastic.

Minibatch\_size được xác định bởi các yếu tố sau:

\* Batch\_size càng lớn thì ước tính cho gradient càng chính xác nhưng phải đánh đổi bằng tốc độ tính toán

\* Với kiến trúc đa lõi (VD như máy tính gồm nhiều CPU, CPU gồm nhiều core,….) thì việc thì việc để batch\_size quá nhỏ không làm tăng tốc độ tính toán, vì thế có một ngưỡng min batch\_size và nếu ở dưới ngưỡng thì thời gian xử lí ko giảm nữa.

\* Thường các ví dụ trong batch xử lí song song thì dung lượng bộ nhớ sẽ thay đổi theo batch\_size và đấy sẽ là yếu tố hạn chế batch\_size về mặt phần ứng thiết bị.

\* Một số loại phần cứng đạt được giới hạn chạy tốt hơn với các batch\_size cụ thể. Đặc biệt khi dùng GPU người ta thường cung cấp batch\_size = lũy thừa của 2 (. Điển hình là nằm trong khoảng từ 32 -> 256, đôi khi là 16 với các mô hình lớn.

\* Các batch\_size nhỏ có thể mang lại hiệu quả regulazition có thể do các noise mà nó thêm vào khi ước tính gradient. Sự khái quát hóa(Generalization) thường tốt nhất khi batch\_size = 1. Đào tạo với batch\_size nhỏ như vậy cớ thể yêu cầu 1 learning rate nhỏ để duy trì sự ổn định vì phương sai cao trong ước tính của gradient. Tổng thời gian chạy cũng có thể rất cao do phải thực hiện nhiều bước hơn, do lr nhỏ, do mất nhiều bước hơn để quan sát toàn bộ tập dữ liệu vì batch\_size = 1.

Các thuật toán khác nhau sử dụng thông tin từ các mini\_batch là khác nhau. Một số thuật toán nhạy cảm với lỗi lấy mẫu(sample error) hơn các thuật toán khác, có thể là do chúng sử dụng thông tin khó để ước lượng chính xác với ít mẫu hoặc do chúng sử dụng thông tin theo các khuếch đại lỗi lấy mẫu.

Ví dụ:

* + Với các pp cập nhật dựa trên gradient thường khá ổn với các batch\_size nhỏ như 100.
  + Các pp cấp 2( hay còn gọi là sử dụng matrix Hessia) để cập nhật thì thường yêu cầu batch\_size lớn, chẳng hạn như 10000 sample.

Điều quan trọng nữa là các batch nhỏ được chọn ngẫu nhiên, tính toán 1 ước tính unbias của expect gradient từ 1 sample yêu cầu các sample đó là độc lập tuyến tính. Đặc biệt hai ước tính gradient tiếp theo phải độc lập với nhau, vì trong cùng 1 batch thường sẽ sử dụng các sample gần nhau. Tuy nhiên trong tự nhiên, dữ liệu thường được sắp xếp mà các ví dụ liên tiếp có mỗi tương quan cao. Ví dụ về hồ sơ xét nghiệm mẫu máu, thường danh sách sẽ sắp xếp liên tiếp nhau là mẫu máu của 1 người được đo nhiều lần. May mắn thay trong thực tế thường chỉ xáo trộn(shuffle) là có thể cải thiện. Hãy nhớ không bao giờ shuffle các ví dụ theo bất kỳ cách nào có thể giảm hiệu quả của thuật toán một cách nghiêm trọng.

Chúng ta có thể tối ưu hóa từng batch hoặc thực hiện nó song song. Cách tiếp cận phân tán song song không đồng bộ này sẽ được nói sau.

Một động lực của thuật toán stochstic gradient với minibatch là nó đi theo đạo hàm của lỗi thực sự(generalization) miễn là không có ví dụ nào được lặp lại. Hiểu đơn giản trong stochastic gradient thì mỗi minibatch là một tập con nhỏ của dữ liêu huấn luyện và nó được sử dụng để tính gradient. Khi không bị lặp lại thì mỗi minibatch là duy nhất và đại diện cho một phần của dữ liệu gốc . Tuy nhiên khi chúng ta trộn ngẫu nhiên là lặp lại, mỗi minibatch sẽ chứa các ví dụ đã được sử dụng trước đó dẫn đến việc ước lượng generate error bị bias.

Thực tế việc stochastic gradient giảm thiểu generalization error là trong online data. Nói cách khác tập train không bao giờ lặp lại.

Tất nhiên các giải thích về stochastic gradient và generlization error chỉ áp dụng khi các ví dụ không được sử dụng lại. Tuy nhiên thông thường tốt nhất là nên thực hiện 1 số lượt đi qua tập train(epochs) trừ khi tập train quá lớn. Khi nhiều epochs như vậy được sử dụng thì chỉ epochs1 là tuận theo gradient unbias của generalization error. Tất nhiên các epochs sau thường mang lại đủ lợi ích do giảm train error để bù đắp tác hại mà chúng gây ra bằng các tăng khoảng cách giữa train error và test error

Với bộ dữ liệu tăng nhanh về kích thước, mỗi lần train chỉ 1 lần hoặc thậm chí chưa sử dụng hết tập train thì khi đó overfitting không là vấn đề mà thay vào đó underfitting mới là điều cần quan tâm.

8.2 Challenges in Neural Nerwork Optimization