Dự án Cuối kỳ

Môn: Nhập môn Học Máy

Bài 1 (3 điểm): làm riêng từng người

Trình bày một bài nghiên cứu, đánh giá của em về các vấn đề sau:

1) Tìm hiểu, so sánh các phương pháp Optimizer trong huấn luyện mô hình học máy;
2) Tìm hiểu về Continual Learning và Test Production khi xây dựng một giải pháp học máy để giải quyết một bài toán nào đó.

Giải Đáp Bài 1:

1) Tìm hiểu, so sánh các phương pháp Optimizer trong huấn luyện mô hình học máy

Thuật toán tối ưu(Optimizer) là cơ sở để xây dựng mô hình neural network với mục đích "**học**" được các features hoặc pattern của dữ liệu đầu vào, từ đó có thể tìm 1 cặp weights và bias phù hợp để tối ưu hóa model.

Optimizer được sử dụng để huấn luyện các mô hình học máy. Quá trình huấn luyện mô hình học máy bao gồm việc tìm các tham số của mô hình sao cho hàm mất mát của mô hình đạt giá trị thấp nhất.

Đặc trưng của optimizer

- Tốc độ học: là một tham số điều khiển mức độ nhanh chậm của quá trình tối ưu hóa.
- Loại hàm mục tiêu: Một số optimizer chỉ hoạt động hiệu quả với một số loại hàm mục tiêu nhất định.
- Kích thước tập dữ liệu: Một số optimizer có thể hoạt động hiệu quả hơn với các tập dữ liệu lớn hoặc nhỏ.
- Số lượng tham số: Một số optimizer có thể hoạt động hiệu quả hơn với các mô hình có nhiều tham số hoặc ít tham số.

Một số phương pháp optimizer phổ

- Gredient Descen(GD)
- Stochastic Gradient Descent (SGD)
- Momentum
- Adagrad
- RMSprop

- Adam

Optimizer có tác vụ quan trọng là tính toán gradient của hàm mất mát (loss function) theo các tham số của mô hình. Gradient là vector đạo hàm riêng của hàm mất mát theo từng tham số, và cho biết hướng và mức độ thay đổi của hàm mất mát khi các tham số thay đổi. Gradient sẽ điều chỉnh các tham số của mô hình để di chuyển mô hình gần hơn đến điểm tối ưu của hàm mất mát.

Gredient Descen(GD)

Gredient Descen(GD) là giảm dần độ dốc. Nó sẽ chọn 1 nghiệm ngẫu nhiên cứ sau mỗi vòng lặp (hay epoch) thì cho nó tiến dần đến điểm cần tìm.

Mục tiêu của GD là điều chỉnh các tham số của mô hình dựa trên đạo hàm của hàm mất mát để tìm giá trị nhỏ nhất của hàm mất mát đó.

GD hoạt động bằng cách bắt đầu từ một điểm khởi đầu ngẫu nhiên và sau đó cập nhật các tham số của hàm theo hướng giảm dần của gradient hàm tại điểm hiện tại.

Đặc trưng của Gradient Descent:

- Đạo hàm: GD sử dụng đạo hàm của hàm mất mát để xác định hướng và tốc độ cập nhật các tham số. Đạo hàm đo lường sự thay đổi của hàm mất mát khi các tham số được điều chỉnh.

- Hướng giảm: GD điều chỉnh các tham số theo hướng giảm của đạo hàm. Nếu đạo hàm dương, GD sẽ điều chỉnh các tham số giảm; ngược lại, nếu đạo hàm âm, GD sẽ điều chỉnh các tham số tăng.

- Tốc độ học (learning rate): Là một siêu tham số quan trọng trong GD, tốc độ học quyết định độ lớn của bước cập nhật các tham số. Nếu tốc độ học quá nhỏ, quá trình hội tụ sẽ chậm; ngược lại, nếu tốc độ học quá lớn, quá trình hội tụ có thể không ổn định hoặc không hội tụ.

Độ tin cậy của GD:

Độ tin cậy của GD phụ thuộc vào các yếu tố như tốc độ học, cách lựa chọn hàm mất mát, và khởi tạo ban đầu của các tham số. Nếu tốc độ học được chọn sao cho phù hợp và

hàm mất mát được thiết kế tốt, GD có thể hội tụ tới giá trị tối ưu. Tuy nhiên, GD cũng có thể bị mắc kẹt ở điểm cực tiểu cục bộ hoặc không hội tụ nếu tốc độ học không được cân nhắc cẩn thận.

- Kiểu hàm mất mát: Một số hàm mất mát có thể khiến GD dễ bị mắc kẹt trong các điểm cục bộ hơn các hàm mất mát khác.
- Kích thước dữ liệu: GD có thể hoạt động tốt hơn với các tập dữ liệu lớn hơn.
- Số lượng tham số: GD có thể hoạt động kém hiệu quả hơn với các mô hình có nhiều tham số.
- Giá trị của learning rate: Giá trị của learning rate quá cao hoặc quá thấp có thể khiến GD không hội tụ.

Giải thuật GD:

- Khởi tạo các tham số ban đầu.
- Lặp lại cho đến khi đạt được điều kiện dừng (ví dụ: số lần lặp tối đa hoặc đạt đến giá trị mất mát nhỏ hơn một ngưỡng):
- Tính toán đạo hàm của hàm mất mát theo từng tham số.
- Cập nhật các tham số theo

Công thức : xnew = xold – learning_rate*gradient(x)

- xnew : là giá trị mới của tham số,
- xold : là giá trị hiện tị của tham số,
- learning_rate : là hệ số học và gradient là gradient tính toán được.

- Trả về giá trị tối ưu của các tham số.

Code:

```
function gradient_descent(f, x0, learning_rate, max_iters)
   for i = 1 to max_iters
      g = gradient(f, x_i)
      x_i = x_i - learning_rate * g
   end for
   return x_i
end function
```

Trong đó:

- f là hàm cần tối ưu hóa
- x0 là điểm khởi đầu
- learning_rate là tốc độ học

- max_iters là số lượng vòng lặp tối đa

Trong mỗi vòng lặp, GD tính toán gradient của hàm tại điểm hiện tại x_i. Sau đó, GD cập nhật các tham số của hàm theo hướng giảm dần của gradient. Quá trình này được lặp lại cho đến khi đạt được điều kiện dừng, chẳng hạn như số lượng vòng lặp tối đa hoặc độ giảm của gradient nhỏ hơn một giá trị nhất định.

Stochastic Gradient Descent (SGD)

Stochastic Gradient Descent (SGD) là một biến thể của Gradient Descent (GD) được sử dụng để tối ưu hóa các hàm khả vi. SGD cập nhật các tham số của hàm dựa trên một mẫu ngẫu nhiên của tập dữ liệu. SGD có thể giúp GD tránh bị mắc kẹt trong các điểm cục bộ.

Đặc trưng của Stochastic Gradient Descent (SGD):

- Mini-batch: SGD sử dụng một mini-batch để tính toán đạo hàm. Kích thước của mini-batch được lựa chọn trước và thường là một giá trị nhỏ, ví dụ như 32 hoặc 64.

- Ngẫu nhiên: SGD lựa chọn ngẫu nhiên một mini-batch từ dữ liệu huấn luyện trong mỗi bước cập nhật. Nó đảm bảo rằng mỗi mẫu dữ liệu có cơ hội được sử dụng và giúp tránh việc rơi vào điểm cực tiểu cục bộ.

- Tốc độ học (learning rate): Tương tự như GD, SGD cũng sử dụng tốc độ học để điều chỉnh kích thước bước cập nhật các tham số. Tuy nhiên, trong SGD, tốc độ học thường được thiết lập nhỏ hơn do tính ngẫu nhiên của mini-batch có thể làm dao động quá trình hội tụ.

Độ tin cậy của Stochastic Gradient Descent (SGD):

SGD có thể hội tụ tới giá trị tối ưu nhưng không đảm bảo tìm được điểm cực tiểu toàn cục. Do sự ngẫu nhiên của việc lựa chọn mini-batch và đôi khi có thể giúp tránh việc rơi vào điểm cực tiểu cục bộ. Tuy nhiên, SGD có thể dao động quanh vùng tối ưu và không ổn định hơn so với GD truyền thống.

- Kiểu hàm mất mát: Một số hàm mất mát có thể khiến SGD dễ bị mắc kẹt trong các điểm cục bộ hơn các hàm mất mát khác.
- Kích thước dữ liệu: SGD có thể hoạt động tốt hơn với các tập dữ liệu lớn hơn.
- Số lượng tham số: SGD có thể hoạt động kém hiệu quả hơn với các mô hình có nhiều tham số.
- Giá trị của learning rate: Giá trị của learning rate quá cao hoặc quá thấp có thể khiến SGD không hội tụ.

Giải thuật (SGD):

- Khởi tạo các tham số ban đầu.
- Lặp lại cho đến khi đạt được điều kiện dừng (ví dụ: số lần lặp tối đa hoặc đạt đến giá trị mất mát nhỏ hơn một ngưỡng):
- Lựa chọn ngẫu nhiên một mini-batch từ dữ liệu huấn luyện.
- Tính toán đạo hàm của hàm mất mát dựa trên mini-batch đó.
- Cập nhật các tham số theo

Công thức:

thamsoNew = thamsoOld - learning_rate * gradient

trong đó,

- thamsoNew là giá trị mới của tham số
- thamsoOld là giá trị hiện tại của tham số
- learning_rate là tỷ lệ học (learning rate)

- Trả về giá trị tối ưu của các tham số.

Code:

```
function stochastic_gradient_descent(f, x0, learning_rate, max_iters)
  for i = 1 to max_iters
    x_i = x_i - learning_rate * gradient(f, x_i, sample)
  end for
  return x_i
end function
```

Trong đó:

- f là hàm cần tối ưu hóa

- x0 là điểm khởi đầu
- learning_rate là tốc độ học
- max_iters là số lượng vòng lặp tối đa
- sample là một mẫu ngẫu nhiên của tập dữ liệu

Trong mỗi vòng lặp, SGD chọn một mẫu ngẫu nhiên từ tập dữ liệu và tính toán gradient của hàm tại điểm hiện tại x_i dựa trên mẫu ngẫu nhiên đó. Sau đó, SGD cập nhật các tham số của hàm theo hướng giảm dần của gradient. Quá trình này được lặp lại cho đến khi đạt được điều kiện dừng, chẳng hạn như số lượng vòng lặp tối đa hoặc độ giảm của gradient nhỏ hơn một giá trị nhất định.

Adagrad

Adagrad là một biến thể của Gradient Descent (GD) được sử dụng để tối ưu hóa các hàm khả vi. Adagrad điều chỉnh tốc độ học của GD theo độ lớn của gradient của hàm tại điểm hiện tại. Adagrad có thể giúp GD tránh bị mắc kẹt trong các điểm cục bộ và hội tụ nhanh hơn GD.

Đặc trưng của Adagrad:

- Tích lũy đạo hàm: Adagrad tích lũy bình phương của các đạo hàm trước đây để đánh giá mức độ quan trọng của các tham số. Các tham số có đạo hàm lớn hơn sẽ có tốc độ học giảm nhanh hơn, trong khi các tham số có đạo hàm nhỏ hơn sẽ có tốc độ học tăng lên.

- Điều chỉnh tốc độ học: Adagrad điều chỉnh tốc độ học cho từng tham số bằng cách chia tốc độ học cho căn bậc hai của tổng bình phương của đạo hàm đã tích lũy. Nó giúp giảm tốc độ học các tham số có đạo hàm lớn và tăng tốc độ học các tham số có đạo hàm nhỏ.

- Lịch sử đạo hàm: Adagrad lưu trữ lịch sử của các đạo hàm bằng cách tích lũy bình phương của chúng. Quá trình tích lũy này đảm bảo rằng các tham số có đạo hàm lớn sẽ có tốc độ học giảm dần theo thời gian.

Độ tin cậy của Adagrad:

Adagrad có thể giúp tìm giá trị tối ưu nhanh hơn trong quá trình huấn luyện vì nó tự điều chỉnh tốc độ học cho từng tham số dựa trên lịch sử của đạo hàm.

Adagrad giúp giảm tốc độ học cho các tham số quan trọng và tăng tốc độ học cho các tham số ít quan trọng, từ đó cân nhắc giữa sự ổn định và sự hội tụ.

- Kiểu hàm mất mát: Một số hàm mất mát có thể khiến Adagrad dễ bị mắc kẹt trong các điểm cục bộ hơn các hàm mất mát khác.
- Kích thước dữ liệu: Adagrad có thể hoạt động tốt hơn với các tập dữ liệu lớn hơn.
- Số lượng tham số: Adagrad có thể hoạt động kém hiệu quả hơn với các mô hình có nhiều tham số.

Giải thuật Adagrad:

- Khởi tạo các tham số ban đầu và một biến tích lũy (accumulator) khởi đầu bằng 0 cho mỗi tham số.
- Lặp lại cho đến khi đạt được điều kiện dừng (ví dụ: số lần lặp tối đa hoặc đạt đến giá trị mất mát nhỏ hơn một ngưỡng):
  - Tính toán đạo hàm của hàm mất mát theo từng tham số.
  - Cập nhật biến tích lũy bằng cộng dồn bình phương của đạo hàm.
  - Cập nhật các tham số theo công thức: tham số mới = tham số cũ - (tốc độ học / căn bậc hai của biến tích lũy) * đạo hàm.
- Trả về giá trị tối ưu của các tham số.

Code:

```
function adagrad(f, x0, learning_rate, max_iters)
  h = zeros(size(x0))
  for i = 1 to max_iters
    g = gradient(f, x_i)
    h = h + g * g
    x_i = x_i - learning_rate / sqrt(h) * g
  end for
  return x_i
end function
```

Trong đó:

- f là hàm cần tối ưu hóa
- x0 là điểm khởi đầu

- learning_rate là tốc độ học
- max_iters là số lượng vòng lặp tối đa
- g là gradient của hàm tại điểm hiện tại x_i

Trong mỗi vòng lặp, Adagrad tính toán gradient của hàm tại điểm hiện tại x_i. Sau đó, Adagrad cập nhật tốc độ học theo hướng giảm dần của gradient. Quá trình này được lặp lại cho đến khi đạt được điều kiện dừng, chẳng hạn như số lượng vòng lặp tối đa hoặc độ giảm của gradient nhỏ hơn một giá trị nhất định.

RMSprop

RMSprop là một biến thể của Adagrad được sử dụng để tối ưu hóa các hàm khả vi. RMSprop điều chỉnh tốc độ học của Adagrad theo độ lớn của gradient của hàm tại điểm hiện tại, nhưng không tính tổng bình phương của gradient như Adagrad. RMSprop có thể giúp Adagrad tránh bị mắc kẹt trong các điểm cục bộ và hội tụ nhanh hơn Adagrad.

RMSprop được thiết kế để giảm độ dốc của hàm mất mát và điều chỉnh learning rate cho từng tham số.

RMSprop sử dụng bình phương độ dốc trung bình để điều chỉnh tốc độ học, giúp thuật toán hội tụ nhanh hơn và tránh bị mắc kẹt ở các cực tiểu cục bộ.

RMSprop giải quyết vấn đề tỷ lệ học giảm dần của Adagrad bằng cách chia tỷ lệ học cho trung bình của bình phương gradient. Nó tương tự như AdaGrad, RMSProp cũng điều chỉnh learning rate cho từng tham số.

Đặc trưng của RMSprop:

- Tích lũy đạo hàm: RMSprop tích lũy bình phương của các đạo hàm trước đó để đánh giá mức độ quan trọng của các tham số tương tự như Adagrad. Tuy nhiên, RMSprop chỉ lưu trữ một số lượng đạo hàm gần đây nhất, thay vì tích lũy toàn bộ lịch sử.

- Điều chỉnh tốc độ học: RMSprop điều chỉnh tốc độ học cho từng tham số bằng cách chia tốc độ học cho căn bậc hai của tổng trung bình di động của bình phương các đạo hàm đã tích lũy.

- Trọng số giảm dần: RMSprop áp dụng một hệ số giảm dần (decay rate) để định rõ mức độ quan trọng của các đạo hàm cũ hơn so với đạo hàm mới. Nó giúp mô hình tập trung vào thông tin mới nhất và loại bỏ sự ảnh hưởng của thông tin lỗi thời.

Độ tin cậy của RMSprop:

RMSprop có tính năng tương tự như Adagrad trong việc thích ứng tốc độ học cho từng tham số. Tuy nhiên, RMSprop giảm ảnh hưởng của đạo hàm nhiễu và chỉ lưu trữ một số lượng đạo hàm gần đây hơn, từ đó cải thiện tính ổn định của thuật toán.

- Kiểu hàm mất mát: Một số hàm mất mát có thể khiến Adam dễ bị mắc kẹt trong các điểm cục bộ hơn các hàm mất mát khác.
- Kích thước dữ liệu: Adam có thể hoạt động tốt hơn với các tập dữ liệu lớn hơn.
- Số lượng tham số: Adam có thể hoạt động kém hiệu quả hơn với các mô hình có nhiều tham số.

Giải thuật RMSprop:

- Khởi tạo các tham số ban đầu và một biến tích lũy (accumulator) khởi đầu bằng 0 cho mỗi tham số.
- Lặp lại cho đến khi đạt được điều kiện dừng:
  - Tính toán đạo hàm của hàm mất mát theo từng tham số.
  - Cập nhật biến tích lũy bằng cộng dồn bình phương của đạo hàm.
  - Cập nhật các tham số theo công thức: tham số mới = tham số cũ - (tốc độ học / căn bậc hai của biến tích lũy) * đạo hàm.
  - Áp dụng trọng số giảm dần cho biến tích lũy.
- Trả về giá trị tối ưu của các tham số.

Code:

```
function rmsprop(f, x0, learning_rate, max_iters)
    h = zeros(size(x0))
    for i = 1 to max_iters
        g = gradient(f, x_i)
        h = (0.9 * h) + (0.1 * g * g)
        x_i = x_i - learning_rate / sqrt(h) * g
    end for
    return x_i
end function
```

Trong đó:

- f là hàm cần tối ưu hóa
- x0 là điểm khởi đầu
- learning_rate là tốc độ học
- max_iters là số lượng vòng lặp tối đa
- g là gradient của hàm tại điểm hiện tại x_i

Trong mỗi vòng lặp, RMSprop tính toán gradient của hàm tại điểm hiện tại x_i. Sau đó, RMSprop tính toán bình phương của gradient và cập nhật tốc độ học theo hướng giảm dần của bình phương gradient. Quá trình này được lặp lại cho đến khi đạt được điều kiện dừng, chẳng hạn như số lượng vòng lặp tối đa hoặc độ giảm của gradient nhỏ hơn một giá trị nhất định.

Adam

Adam (Adaptive Moment Estimation) là một phương pháp tối ưu hóa trong quá trình huấn luyện mô hình machine learning, kết hợp cả Momentum và RMSprop. Adam được thiết kế để tự động điều chỉnh learning rate cho từng tham số dựa trên gradient và lịch sử của gradient. Adam giúp cải thiện tốc độ hội tụ và độ ổn định của mô hình.

Đặc trưng của Adam:

- Tích lũy đạo hàm: Adam tích lũy bình phương của các đạo hàm trước đó tương tự như RMSprop. Nó giúp xác định mức độ quan trọng của các tham số.

- Điều chỉnh tốc độ học: Adam điều chỉnh tốc độ học cho từng tham số bằng cách sử dụng một phần tử động được gọi là "momentum". Momentum giúp lưu trữ thông tin về sự thay đổi của đạo hàm trong quá khứ, giúp mô hình di chuyển nhanh hơn trong các hướng có đạo hàm lớn và giảm đà trong các hướng có đạo hàm nhỏ.

- Bước điều chỉnh chuẩn hóa: Adam sử dụng một bước điều chỉnh chuẩn hóa để giảm ảnh hưởng của độ lớn khởi tạo ban đầu của các tham số. Đảm bảo rằng các tham số được điều chỉnh một cách nhất quán và tránh hiện tượng mất cân bằng trong quá trình huấn luyện.

Độ tin cậy của Adam:

Adam kết hợp tính ổn định của RMSprop và khả năng di chuyển nhanh của Momentum. Có thể giúp nó đạt được độ tin cậy cao trong quá trình tối ưu hóa mô hình máy học.

- Kiểu hàm mất mát: Một số hàm mất mát có thể khiến Adam dễ bị mắc kẹt trong các điểm cục bộ hơn các hàm mất mát khác.
- Kích thước dữ liệu: Adam có thể hoạt động tốt hơn với các tập dữ liệu lớn hơn.
- Số lượng tham số: Adam có thể hoạt động kém hiệu quả hơn với các mô hình có nhiều tham số.

Giải thuật Adam:

- Khởi tạo các tham số ban đầu và hai biến tích lũy (accumulators) khởi đầu bằng 0 cho mỗi tham số.
- Lặp lại cho đến khi đạt được điều kiện dừng:
  - Tính toán đạo hàm của hàm mất mát theo từng tham số.
  - Cập nhật biến tích lũy thứ nhất bằng cộng dồn đạo hàm.
  - Cập nhật biến tích lũy thứ hai bằng cộng dồn bình phương của đạo hàm.
  - Hiệu chỉnh các biến tích lũy bằng lấy căn bậc hai và thêm một giá trị nhỏ để tránh chia cho 0.
  - Cập nhật các tham số theo công thức: tham số mới = tham số cũ - (tốc độ học / căn bậc hai của biến tích lũy thứ hai) * biến tích lũy thứ nhất.
- Trả về giá trị tối ưu của các tham số.

Code:

```
function adam(f, x0, learning_rate, beta1, beta2, epsilon, max_iters)
    m = zeros(size(x0))
    v = zeros(size(x0))
    for i = 1 to max_iters
        g = gradient(f, x_i)
        m = (1 - beta1) * g + beta1 * m
        v = (1 - beta2) * g * g + beta2 * v
        x_i = x_i - learning_rate * m / sqrt(v + epsilon)
    end for
    return x_i
end function
```

Trong đó:

- f là hàm cần tối ưu hóa
- x0 là điểm khởi đầu
- learning_rate là tốc độ học
- beta1 là tham số điều chỉnh tốc độ học ban đầu
- beta2 là tham số điều chỉnh tốc độ học theo thời gian
- epsilon là hằng số để tránh chia cho 0
- max_iters là số lượng vòng lặp tối đa
- g là gradient của hàm tại điểm hiện tại x_i

Trong mỗi vòng lặp, Adam tính toán gradient của hàm tại điểm hiện tại x_i. Sau đó, Adam cập nhật các tham số của hàm theo hướng giảm dần của gradient, điều chỉnh tốc độ học theo hướng giảm dần của bình phương gradient. Quá trình này được lặp lại cho đến khi đạt được điều kiện dừng, chẳng hạn như số lượng vòng lặp tối đa hoặc độ giảm của gradient nhỏ hơn một giá trị nhất định.

❖ Bảng so sánh các phương pháp Optimizer

	Ưu điểm	**Nhược điểm**
Gredient Descen	- Dễ hiểu và triển khai. - Có thể áp dụng cho các mô hình lớn. - Tính toán đơn giản. - không yêu cầu nhiều siêu tham số.	- có thể bị mắc kẹt trong các điểm cục bộ, đặc biệt là với các tập dữ liệu lớn và các mô hình có nhiều tham số. - GD phụ thuộc vào nhiều yếu tố, bao gồm kích thước của tập dữ liệu, số lượng tham số của mô hình và giá trị của learning rate.
Stochastic Gradient Descent	- Nhanh hơn GD vì chỉ sử dụng một mẫu dữ liệu trong mỗi lần cập nhật. - Hướng tới điểm cực tiểu	- Không đảm bảo hội tụ tới điểm cực tiểu toàn cục.

	địa phương tốt hơn GD. - Phù hợp khi dữ liệu lớn.	
Adagrad	- Tự động điều chỉnh tỷ lệ học. - Hiệu quả với độ dốc thưa (sparse gradients). - - Không yêu cầu thiết lập tỷ lệ học ban đầu.	- Tích lũy gradient bình phương có thể giảm tỷ lệ học quá nhanh. - Không có cơ chế thích nghi với các tham số riêng lẻ. - Yêu cầu khối lượng bộ nhớ lớn.
RMSprop	- Giảm ảnh hưởng của squared gradient bằng cách giới hạn lịch sử gradient. - Hiệu quả khi mô hình có các tham số thưa (sparse).	- Cần thiết lập thêm siêu tham số. - Có thể dễ rơi vào điểm cực tiểu cục bộ.
Adam	- Kết hợp ưu điểm của RMSProp và Momentum. - Hiệu quả và ổn định với đa dạng các bài toán. - - Tính toán hiệu quả và hội tụ nhanh.	- Cần chọn giá trị cho các siêu tham số (learning rate, beta1, beta2). - Đôi khi có thể bị mắc kẹt ở điểm cực tiểu địa phương.

❖ Bảng so sánh độ tin cây của các phương pháp Optimiszer

Phương pháp tối ưu hóa	Tốc độ hội tụ	Tránh điểm cực tiểu cục bộ	Độ ổn định

Gradient Descent (GD)	Trung bình	Không	Trung bình
Stochastic Gradient Descent (SGD)	Trung bình	Không	Không
Adagrad	Chậm	Trung bình	Tốt
RMSProp	Trung bình - Nhanh	Trung bình	Tốt
Adam	Nhanh	Có	Tốt

2) Tìm hiểu về Continual Learning và Test Production khi xây dựng một giải pháp học máy để giải quyết một bài toán nào đó.

Continual Learning (CL) là một lĩnh vực nghiên cứu quan trọng trong machine learning mà mô hình học máy được thiết kế để liên tục học từ các tập dữ liệu mới mà nó gặp phải, mà không làm mất đi khả năng thích ứng với dữ liệu đã học được trước đó.

- ❖ Đặc trưng

  Continual Learning có những khả năng như:
  - Khả năng học liên tục từ dữ liệu mới mà nó không quên tri thức đã học trước đó.
  - Khả năng thích ứng và đối mặt với sự thay đổi của môi trường và dữ liệu.
  - Khả năng học từ nhiều nhiệm vụ và từ miề dữ liệu
  - Khả năng tổng hợp kiến thức các nhiệm vụ và các dữ liệu đã từng học
- ❖ Đọ tin cậy

  Về độ tin cậy nó là một thách thức lớn, chính là mô hình có thể quên kiến thức cũ khi học từ dữ liệu mới. Thậm chí, đôi khi có khả năng mất độ tin cậy khi tích hợp dữ liệu mới, đó là hiện tượng quên mô hình(catastrophic Forgetting).

- ❖ Ưu điểm
  - Có khả năng tích hợp và sử dụng lại kiến thức từ nhiều nguồn dữ liệu và nhiệm vụ
  - Có thể giảm thiểu dữ liệu thừa bằng cách tận dụng lại thông tin đã đọc
- ❖ Nhược điểm
  - Thách thức lớn là do hiện tượng quên mô hình(catastrophic Forgetting) khiến cho mô hình mất đi kiến thức quan trong từ dữ liệu cũ.
  - Nó có thể đòi hỏi về cả dữ liệu lận thời gian nhiều hơn so vơi mô hình thông thường.

Trong khi Continual Learning mang lại nhiều lợi ích đối với sự linh hoạt và sự thích ứng, thách thức lớn nhất là làm thế nào để duy trì và sử dụng kiến thức từ dữ liệu cũ trong khi vẫn có khả năng học từ dữ liệu mới một cách hiệu quả. Điều này là một lĩnh vực nghiên cứu đang phát triển và đầy thách thức trong machine learning.

Test production trong ngữ cảnh xây dựng một giải pháp học máy để giải quyết một bài toán cụ thể, chúng ta có thể liên quan đến việc thực hiện bài kiểm thử để đảm bảo rằng giải pháp đó đáp ứng yêu cầu chất lượng và hoạt động hiệu quả.

- ❖ Đặc trưng
  - Quá trình đánh giá hiệu suất của mô hình trên tập dữ liệu kiểm thử để đảm bảo rằng nó hoạt động chính xác và đáng tin cậy.
  - Quá trình triển khai mô hình đã được đào tạo vào môi trường sản xuất để xử lý dữ liệu thực tế và tạo ra dự đoán cho ứng dụng thực tế.
- ❖ Độ tin cậy
  - Về sự hiệu suất, kiểm thử giúp đánh giá độ chính xác và hiệu suất của mô hình trước khi nó được triển khai.
  - Kiểm thử giúp đánh giá xem mô hình có xu hướng quá mức học từ dữ liệu huấn luyện không hoặc có khả năng tổng quát hóa tốt trên dữ liệu mới hay không.

- ❖ Ưu điểm
  - Kiểm thử giúp đảm bảo rằng mô hình hoạt động đúng đắn và chính xác
  - Kiểm soát rũi ro, quá trình kiểm thử giúp kiểm soát rủi ro và giảm thiểu sự cố trong quá trình triển khai.
- ❖ Nhược điểm
  - Quá trình kiểm thử và triển khai có thể tốn kém về chi phí và thời gian, đặc biệt là khi áp dụng các phương pháp chặt chẽ.
  - Các mô hình phức tạp có thể khó kiểm thử và triển khai do độ phức tạp và tính linh hoạt cao
  - Triển khai mô hình trong môi trường sản xuất có thể đặt ra các vấn đề về bảo mật, đặc biệt là khi xử lý dữ liệu nhạy cảm