**ĐẠI HỌC QUỐC GIA THÀNH PHỐ HỒ CHÍ MINH**

**TRƯỜNG ĐẠI HỌC CÔNG NGHỆ THÔNG TIN**



**MÔN HỌC CẤU TRÚC DỮ LIỆU VÀ GIẢI THUẬT NÂNG CAO**

**BÁO CÁO ĐỒ ÁN MÔN HỌC**

**COMPUTATIONAL GRAPH**

GVHD: Nguyễn Thanh Sơn

Sinh viên thực hiện:

23521187 – Nguyễn Văn Phú

23520957 – Trần Nhật Minh

🙡🙢 Tp. Hồ Chí Minh, 03/2025 🙠🙣

**NHẬN XÉT CỦA GIÁO VIÊN HƯỚNG DẪN**

*……., ngày……...tháng……năm 2025*

**Người nhận xét**

*(Ký tên và ghi rõ họ tên****)***

**MỤC LỤC**

[I. Giới thiệu chung 4](#_Toc196405519)

[II. Computational Graph 4](#_Toc196405520)

[III. Lan truyền tiến 6](#_Toc196405521)

[IV. Lan truyền ngược 6](#_Toc196405522)

[V. Mã giả 8](#_Toc196405523)

[VI. Phân loại 10](#_Toc196405524)

[1. Đồ thị tính toán tĩnh (static computational graph): 10](#_Toc196405525)

[2. Đồ thị tính toán động (dynamic computational graph): 12](#_Toc196405526)

[3. Đồ thị lai (hybrid computational graph): 13](#_Toc196405527)

[VII. Ứng dụng trong neural network 15](#_Toc196405528)

[VIII. Mốt số ứng dụng khác 18](#_Toc196405529)

**NỘI DUNG BÀI LÀM**

1. Giới thiệu chung

Grap: Một đồ thị là một dạng biểu diễn hình ảnh của một tập các đối tượng, trong đó các cặp đối tượng được kết nối bởi các link.

Các đối tượng được nối liền nhau được biểu diễn bởi các điểm được gọi là các đỉnh (vertices), và các link mà kết nối các đỉnh với nhau được gọi là các cạnh (edges).

Có hai loại đồ thị:

-Đồ thị vô hướng

Ảnh có chứa hàng, vòng tròn, biểu đồ, màu trắng

Nội dung do AI tạo ra có thể không chính xác.

-Đồ thị có hướng

Ảnh có chứa hàng, biểu đồ, vòng tròn, màu trắng

Nội dung do AI tạo ra có thể không chính xác.

1. Computational Graph

**Đồ thị tính toán (Computational Graph)** là một công cụ quan trọng trong học máy và học sâu, được sử dụng để biểu diễn các phép toán và mối quan hệ giữa chúng dưới dạng một đồ thị có hướng. Trong đồ thị này, các **nút (nodes)** đại diện cho các toán tử hoặc biến, trong khi các **cạnh (edges)** biểu thị luồng dữ liệu hoặc sự phụ thuộc giữa các toán tử và biến đó.

**Nguyên nhân hình thành:**

Sự ra đời của đồ thị tính toán xuất phát từ nhu cầu quản lý và tối ưu hóa các phép toán phức tạp trong mô hình học sâu. Khi các mô hình trở nên phức tạp hơn, việc theo dõi và tính toán các phép toán trở nên khó khăn. Đồ thị tính toán giúp:

* **Biểu diễn rõ ràng các phép toán**: Giúp người phát triển hiểu rõ cấu trúc và luồng dữ liệu trong mô hình.
* **Tối ưu hóa hiệu suất**: Cho phép xác định và loại bỏ các phép toán dư thừa, tối ưu hóa việc sử dụng tài nguyên tính toán.
* **Hỗ trợ tính toán đạo hàm tự động**: Đặc biệt hữu ích trong quá trình huấn luyện mô hình, giúp tính toán gradient một cách hiệu quả.

**Ý tưởng tạo ra:**

Ý tưởng chính của đồ thị tính toán là sử dụng một cấu trúc đồ thị có hướng để biểu diễn các phép toán toán học phức tạp. Trong đó:

* **Nút (node)**: Đại diện cho các biến hoặc phép toán.
* **Cạnh (edge)**: Biểu thị mối quan hệ phụ thuộc hoặc luồng dữ liệu giữa các nút.

Việc sử dụng đồ thị này cho phép thực hiện cả **lan truyền xuôi (forward propagation)** và **lan truyền ngược (backward propagation)** một cách hiệu quả, hỗ trợ quá trình huấn luyện mô hình học sâu.

Cấu trúc của computatinal graph

Node (Nút): Trong Computational Graph, mỗi toán tử (ví dụ: phép cộng, nhân) và biến đều được biểu diễn bởi một node. Node có thể là một toán tử hoặc một biến đơn giản.

Edge (Cạnh): Các node được nối với nhau bởi các cạnh, thể hiện mối quan hệ giữa chúng. Cạnh có thể mang theo dữ liệu (kết quả của phép toán) từ node này sang node khác.

Graph (Đồ thị): Tất cả các node và cạnh tạo nên một đồ thị, mô tả quá trình tính toán từ input đến output.

Ảnh có chứa vòng tròn, biểu đồ, màu hồng, hàng

Nội dung do AI tạo ra có thể không chính xác.

1. Lan truyền tiến

**Truyền tiến** (feedforward) là quá trình tính toán tuần tự từ đầu vào đến đầu ra của mô hình. Cụ thể, quá trình này diễn ra như sau:​

**Tính toán giá trị tại mỗi nút:** Bắt đầu từ các nút đầu vào, ta thực hiện các phép toán tại mỗi nút theo thứ tự, dựa trên cấu trúc và quan hệ phụ thuộc trong đồ thị tính toán.​

**Lưu trữ các giá trị trung gian:** Các giá trị được tính toán tại mỗi nút được lưu lại để sử dụng trong quá trình lan truyền ngược (backpropagation) sau này.

1. Lan truyền ngược

-Đạo hàm riêng

Ảnh có chứa biểu đồ, văn bản, hàng, ảnh chụp màn hình

Nội dung do AI tạo ra có thể không chính xác.

Đồ thị tính toán giúp tính đạo hàm riêng theo các quy tắc:

Đạo hàm của biểu thức cộng tại 1 biến = 1

Đạo hàm của biểu thức nhân theo biến này = giá trị của biến còn lại

Ảnh có chứa văn bản, ảnh chụp màn hình, Phông chữ

Nội dung do AI tạo ra có thể không chính xác.

-Truyền ngược (backpropagation): trong đồ thị tính toán là quá trình "đi ngược" theo luồng dữ liệu từ đầu ra trở về đầu vào, với mục tiêu tính đạo hàm (gradient) của hàm mất mát đối với từng tham số của mô hình (như trọng số và bias). Đây là bước thiết yếu để cập nhật các tham số theo hướng giảm sai số dự đoán (loss) trong quá trình huấn luyện.

Ý tưởng đầu tiên về thuật toán Backpropagation được đề xuất bởi Paul Werbos vào năm 1974 trong bài báo "Beyond Regression: New Tools for Prediction and Analysis in the Behavioral Sciences". Tuy nhiên ý tưởng này chưa được biết đến rộng rãi cho đến khi nó được các tác giả David Rumelhart, Geoffrey Hinton và Ronald Williams phát triển và áp dụng vào mạng nơ-ron nhân tạo trong bài báo “Learning representations by back-propagating errors” vào năm 1986. Bài báo này đã giúp cho thuật toán Backpropagation trở thành một phương pháp quan trọng để huấn luyện các mô hình mạng nơ- ron nhân tạo.

1. Mã giả

|  |
| --- |
| import math  from graphviz import Digraph  from IPython.display import display, Image  # Lớp cơ sở cho một nút trong đồ thị tính toán  class Node:      def \_\_init\_\_(self):          self.value = None   # Giá trị tính được của nút          self.grad = 0       # Giá trị gradient sau lan truyền ngược      def forward(self):          raise NotImplementedError      def backward(self, grad):          raise NotImplementedError      # Hàm lấy danh sách các "con" (children) của nút (dùng cho việc vẽ đồ thị)      def get\_children(self):          return []      def get\_expr(self):          '''Trả về biểu thức (formula) của node dưới dạng chuỗi.'''          return type(self).\_\_name\_\_  # Lớp đại diện cho biến đầu vào  class Variable(Node):      def \_\_init\_\_(self, value, name=None):          super().\_\_init\_\_() #gọi từ lớp node          self.value = value #gán giá trị          self.name = name if name is not None else f"Var({value})" #xác định tên biến      def forward(self):          return self.value  #trả về giá trị của lan truyền xuôi      def backward(self, grad):          self.grad += grad   #xác định dạo hàm      def get\_children(self):          return []  # Biến đầu vào không có nút con      def get\_expr(self):          return self.name  # Lớp thực hiện phép cộng  class Add(Node):      def \_\_init\_\_(self, left, right):          super().\_\_init\_\_()          self.left = left          self.right = right      def forward(self):          self.value = self.left.forward() + self.right.forward()          return self.value      def backward(self, grad):          self.grad += grad  # Cập nhật gradient cho node Add          self.left.backward(grad)          self.right.backward(grad)      def get\_children(self):          return [self.left, self.right]      def get\_expr(self):          return f"({self.left.get\_expr()} + {self.right.get\_expr()})"  # Lớp thực hiện phép nhân  class Multiply(Node):      def \_\_init\_\_(self, left, right):          super().\_\_init\_\_()          self.left = left          self.right = right      def forward(self):          self.value = self.left.forward() \* self.right.forward()          return self.value      def backward(self, grad):          self.grad += grad  # Cập nhật gradient cho node Multiply          self.left.backward(grad \* self.right.value)          self.right.backward(grad \* self.left.value)      def get\_children(self):          return [self.left, self.right]      def get\_expr(self):          return f"({self.left.get\_expr()} \* {self.right.get\_expr()})"  # Nếu cần thêm một số hàm khác (như Sin) thì định nghĩa thêm  # Hàm xây dựng đồ thị trực quan từ computational graph  def build\_graph(node, dot=None, visited=None):      if visited is None: #kiểm soát việc duyệt các node          visited = set()      if dot is None: # tạo node cho đối tượng          dot = Digraph(format='png')      node\_id = str(id(node))      if node\_id in visited: #xác định việc cập nhập          return dot      visited.add(node\_id)      # Chỉ hiện thị công thức      label = f"{node.get\_expr()}"      dot.node(node\_id, label)      # Vẽ cạnh từ con đến node      for child in node.get\_children():          child\_id = str(id(child))          dot.edge(child\_id, node\_id)          build\_graph(child, dot, visited)      return dot  # Ví dụ: xây dựng đồ thị cho hàm f(x, y, z) = x + y\*z  if \_\_name\_\_ == "\_\_main\_\_":      # Khởi tạo các biến đầu vào      x = Variable(2.0, name="x")      y = Variable(3.0, name="y")      z = Variable(4.0, name="z")      # Xây dựng computational graph      multiplication = Multiply(y, z)  # y \* z      f = Add(x, multiplication)        # x + y\*z      # Bước 1: Tính giá trị đầu ra      result = f.forward()      print("Giá trị của f(x, y, z):", result)      # Bước 2: Lan truyền ngược để tính đạo hàm      f.backward(1.0)      print("Gradient của x:", x.grad)      print("Gradient của y:", y.grad)      print("Gradient của z:", z.grad)      # Bước 3: Xây dựng đồ thị và hiển thị trực tiếp      dot = build\_graph(f)      png\_data = dot.pipe(format='png')      display(Image(png\_data)) |

1. Phân loại
2. Đồ thị tính toán tĩnh (static computational graph):

(Define-Then-Run) Toàn bộ đồ thị tính toán được định nghĩa **trước khi** thực hiện bất kỳ phép tính nào. Khi đã định nghĩa xong, đồ thị được “đóng băng”, tức là **không thể thay đổi** trong quá trình thực thi.

a)Ưu điểm:

-Độ ổn định cao: Với đồ thị cố định, ta có thể kiểm chứng và debug một lần rồi thực thi lại nhiều lần với cùng một cấu trúc, đảm bảo tính nhất quán của kết quả

-Tối ưu hóa hiệu suất: Vì toàn bộ đồ thị đã được định nghĩa từ đầu, các công cụ có thể tối ưu hóa quá trình thực thi (như ghép nối các phép tính, tối ưu sử dụng bộ nhớ, và parallelization). Điều này dẫn đến hiệu suất chạy cao và khả năng mở rộng tốt trên các hệ thống phân tán.

-Triển khai và sản xuất dễ dàng: Do đồ thị đã cố định trước nên việc chuyển giao từ môi trường huấn luyện sang sản xuất thường trở nên thống nhất và kiểm soát được hơn.

b) Nhược điểm:

-Thiếu linh hoạt: Do đồ thị được xây dựng hoàn toàn trước khi thực thi, nên việc thay đổi cấu trúc mô hình trong quá trình huấn luyện (ví dụ như điều chỉnh cấu trúc theo dữ liệu đầu vào) trở nên khó khăn.

-Khó debug: Việc debug một đồ thị tĩnh có thể phức tạp vì lỗi xảy ra trong quá trình thực thi không luôn rõ ràng, và thông báo lỗi thường không chỉ ra trực tiếp vị trí trong code gây ra vấn đề.

-Yêu cầu xây dựng đồ thị trước: Bất kỳ sự thay đổi nào (ví dụ như thay đổi kích thước batch, điều kiện nhánh logic) đều yêu cầu xây dựng lại đồ thị, gây cản trở trong quá trình nghiên cứu và phát triển.

c) Mã giả:

Các thư viện hỗ trợ như TensorFlow (phiên bản 1.x), Theano, MXNet (Symbolic API)

|  |
| --- |
| import tensorflow as tf  # Tạo một đồ thị mới  g = tf.Graph()  with g.as\_default():      # Khai báo biến a và b      a = tf.Variable(2.0, name="a")      b = tf.Variable(3.0, name="b")      # Truyền xuôi: tính toán f = a + b      f = tf.add(a, b, name="c")      # Truyền ngược: tính đạo hàm của c đối với a và b      grad\_a, grad\_b = tf.gradients(f, [a, b])  # Khởi tạo session và chạy đồ thị  with tf.compat.v1.Session(graph=g) as sess:      sess.run(tf.compat.v1.global\_variables\_initializer())      # Truyền xuôi: tính giá trị của f      f\_val = sess.run(f)      print("Giá trị của f (a + b):", f\_val)      # Truyền ngược: tính đạo hàm của f đối với a và b      grad\_a\_val, grad\_b\_val = sess.run([grad\_a, grad\_b])      print("Đạo hàm của f đối với a:", grad\_a\_val)      print("Đạo hàm của f đối với b:", grad\_b\_val) |

1. Đồ thị tính toán động (dynamic computational graph):

(Define-By-Run) đồ thị được **xây dựng tức thời** khi các phép toán được thực hiện. Điều này nghĩa là cấu trúc của đồ thị được xác định **trong thời gian chạy**, dựa trên dòng dữ liệu đi qua nó.

1. Ưu điểm:

-Linh hoạt và dễ sử dụng: Người dùng có thể xây dựng và thay đổi cấu trúc của mô hình trong quá trình chạy, giúp dễ dàng triển khai các mô hình phức tạp có cấu trúc điều kiện hoặc lặp phức tạp. Điều này thuận tiện trong quá trình thử nghiệm và phát triển mô hình.

-Dễ debug: Vì đồ thị được xây dựng theo từng bước, lỗi thường phát sinh ngay lập tức và có thông báo trực quan, giúp việc gỡ lỗi trở nên đơn giản hơn.

-Tương tác thời gian thực: Phù hợp với môi trường nghiên cứu và phát triển khi cần xem trực tiếp kết quả của từng bước tính toán, cho phép các điều chỉnh ngay lập tức.

1. Nhược điểm:

-Hiệu suất có thể thấp hơn: Do việc xây dựng đồ thị diễn ra trong quá trình thực thi, quá trình này có thể thêm chi phí tính toán và làm giảm hiệu quả tối ưu hóa so với đồ thị tĩnh.

-Khả năng tối ưu hạn chế: Một số tối ưu hóa cấp độ cao (như tối ưu toàn cục trên đồ thị) khó đạt được vì đồ thị không được cố định trước. Điều này có thể làm giảm khả năng sử dụng tài nguyên tối ưu khi huấn luyện các mô hình quy mô lớn.

-Không nhất quán khi triển khai sản xuất: Trong môi trường sản xuất (production), người ta thường ưu tiên **đồ thị tĩnh** vì dễ tối ưu hóa, đóng gói

1. Mã giả:

Các thư viện hỗ trợ như PyTorch (Autograd), TensorFlow Eager (tf.GradientTape), Chainer, DyNet,JAX

|  |
| --- |
| import torch  # Khởi tạo 2 biến với requires\_grad=True để theo dõi gradient  x = torch.tensor(2.0, requires\_grad=True)  w = torch.tensor(3.0, requires\_grad=True)  # Forward pass: y = x \* w  y = x \* w  # Backward pass: tính ∂y/∂x và ∂y/∂w  y.backward()  print(f"y = {y.item()}")              # y = 6.0  print(f"dy/dx = {x.grad.item()}")    # dy/dx = 3.0  print(f"dy/dw = {w.grad.item()}")    # dy/dw = 2.0 |

1. Đồ thị lai (hybrid computational graph):

Là sự kết hợp giữa đồ thị tính toán tĩnh và đồ thị tính toán động.

1. Ưu điểm:

-Linh Hoạt và Hiệu Suất: Nhận được tính linh hoạt của dynamic graph và sau đó có thể chuyển đổi thành một static graph để tối ưu hóa hiệu năng.

-Phát Triển Linh Hoạt: Dễ dàng debug và phát triển, sau đó chuyển sang dạng tối ưu để sử dụng trong môi trường sản xuất.

1. Nhược điểm:

-Độ Phức Tạp: Việc chuyển đổi giữa các chế độ có thể làm tăng độ phức tạp của hệ thống, đòi hỏi quá trình “tracing” và biên dịch, từ đó có thể tạo ra một số hạn chế riêng về hiệu năng nếu không được quản lý cẩn thận.

-Hỗ Trợ và Ổn Định: Chế độ hybrid có thể không được hỗ trợ đầy đủ hoặc ổn định trong tất cả các trường hợp, đặc biệt là với các mô hình có logic điều kiện phức tạp

1. Mã giả

Các thư viện hỗ trợ như Apache MXNet Gluon (HybridBlock), PyTorch + TorchScript, JANUS (via GluonCV / MXNet + Python), Ecto

|  |
| --- |
| import torch  import torch.nn as nn  class SimpleModule(nn.Module):      def \_\_init\_\_(self):          super().\_\_init\_\_()          self.w = nn.Parameter(torch.tensor(3.0))      def forward(self, x):          return x \* self.w  model = SimpleModule()  scripted = torch.jit.trace(model, torch.tensor(2.0))  # Tạo static graph  x = torch.tensor(2.0, requires\_grad=True)  y = scripted(x)               # Forward  y.backward()                  # Backward  print("y =", y.item())        # y = 6.0  print("dy/dw =", model.w.grad.item())  # dy/dw = 2.0 |

1. Ứng dụng trong neural network

McCulloch và Pitts (1943) đã trình bày **mô hình neural đầu tiên** bằng cách đại số hoá hoạt động neuron thành các mệnh đề logic từ đó mở ra **kỷ nguyên** của mạng nơ‑ron nhân tạo và lý thuyết tính toán sinh học. Tầm ảnh hưởng của nó vẫn hiện diện trong mọi nghiên cứu về mạng neuron và học sâu ngày nay.

-Neural network là gì?

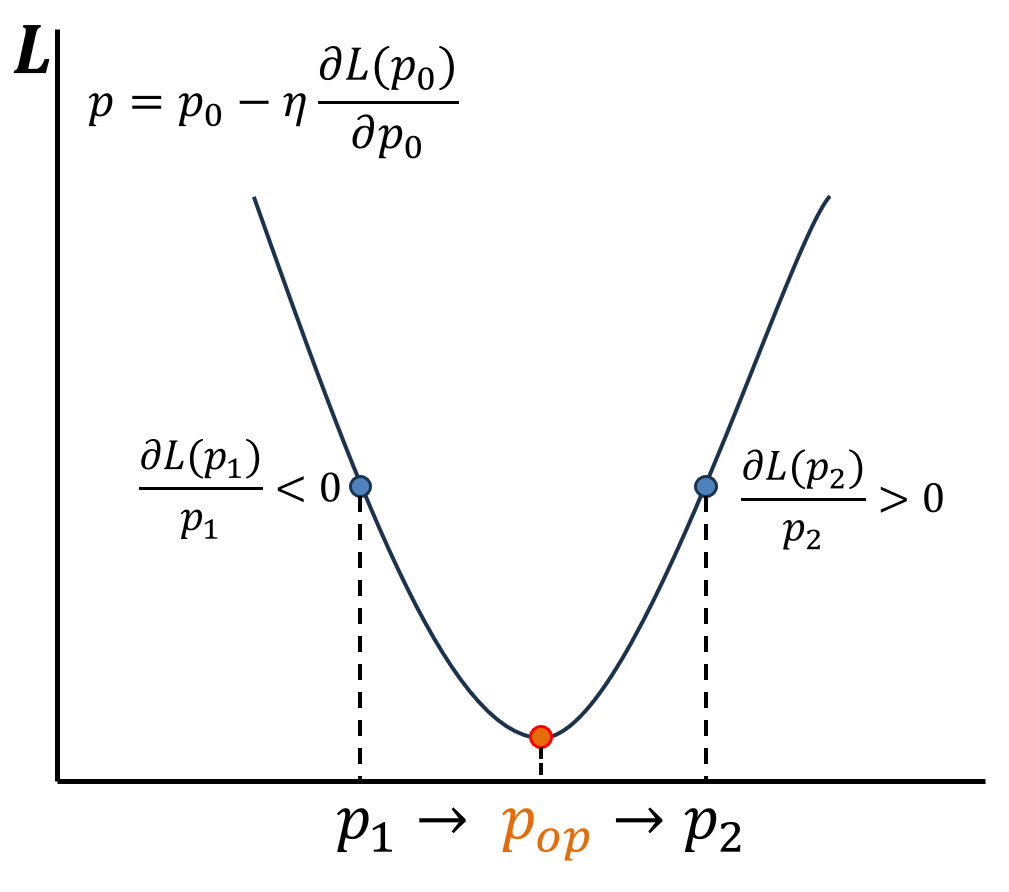
Mạng nơ-ron nhân tạo (Artificial Neural Network - ANN) là một mô hình tính toán được thiết kế để mô phỏng cách thức hoạt động của bộ não con người, nhằm xử lý thông tin và học hỏi từ dữ liệu. Mạng này bao gồm các đơn vị tính toán gọi là nơ-ron nhân tạo, được tổ chức thành các lớp và kết nối với nhau thông qua các trọng số.

-Cấu trúc: gồm lớp đầu vào, lớp ẩn, lớp đầu ra.

Ảnh có chứa biểu đồ, vòng tròn, hàng, Đối xứng

Nội dung do AI tạo ra có thể không chính xác.

* Ứng dụng của đồ thị tính toán trong neural network:
  + - Giải thích cách tổ chức tính toán trong Neural Network (theo forward pass hay backpropagation để tính toán được output của Network).
    - Neural Network là một dạng đặc biệt của Computational Graph.
    - Là công cụ mạnh để đạo hàm theo từng biến.
    - Trong các bài toán network nhiều lớp, cần phải tính đạo hàm theo từng biến. Để làm việc này hiệu quả hơn, người ta sử dụng thuật toán lan truyền ngược (backpropagation).
    - Parallelization (tính toán song song)
* Với n mẫu và m lớp thì đò thị tính toán có tác dụng:
* Giúp dễ kiểm soát và debug mạng sâu, mạng phức tạp
* Phục vụ lan truyền ngược và cập nhật tham số
* Tâng tốc độ huấn luyện và tận dụng phần cứng tốt
* Giảm lỗi người dùng và tăng năng suất phát triển



* Tại sao lại sử dụng đạo hàm để cập nhật trọng số?
* Tại vì đạo hàm giúp thể hiện rõ xu hướng thay đổi của hàm mất mát khi thay đổi một tham số (trọng số).
* Nếu cập nhật trọng số một cách ngẫu nhiên sẽ khiến mô hình học chậm và không hiệu quả. Đạo hàm giúp chọn hướng đi đúng nhanh nhất để mô hình học hiệu quả.
* Ví dụ minh hoạ:

Giả sử ta có mô hình hồi quy logistic dùng để phân loại nhị phân. Với đầu vào và nhãn ,ta tính xác suất dự đoán qua hàm sigmoid:

|  |
| --- |
|  |

với và là các tham số cần học.

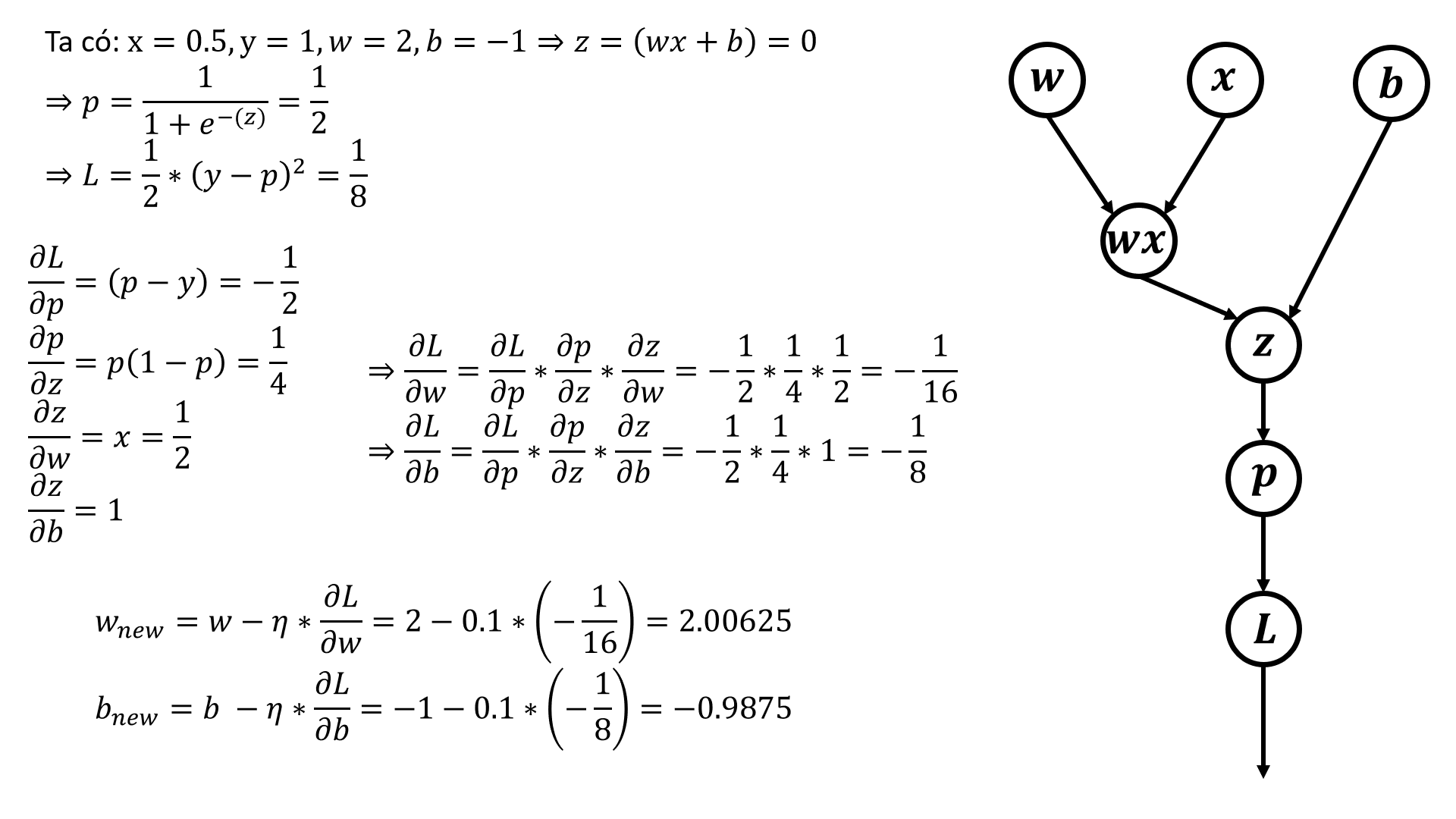
Hàm mất mát được định nghĩa theo công thức Mean Squared Error (MSE):

|  |
| --- |
|  |

Yêu cầu:

1. Xây dựng đồ thị tính toán cho mô hình này.
2. Với giá trị , khởi tạo tham số ban đầu , thực hiện lan truyền tiến để tính giá trị và hàm mất mát .
3. Tính các gradient ​ và ​ qua quy tắc dây chuyền.
4. Mô phỏng một bước cập nhật tham số theo thuật toán gradient descent với tốc độ học (Learning rate (tốc độ học) là một siêu tham số quyết định kích thước của bước nhảy khi mô hình cập nhật trọng số trong quá trình giảm hàm mất mát (loss function))

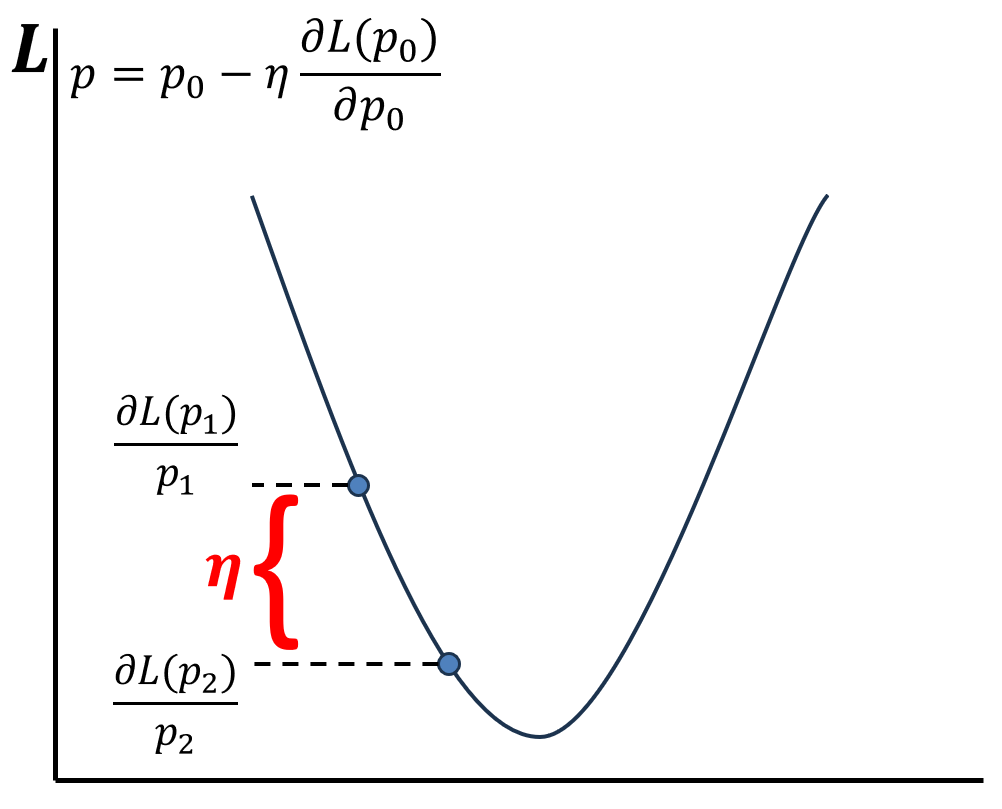
**Lời giải:**



* Tại sao lại lấy ?
* Nếu tốc độ học quá lớn, mô hình có thể bỏ qua điểm tối ưu, không hội tụ hoặc dao động
* Nếu tốc đọ học quá nhỏ, mô hình sẽ học rất chậm, dễ bị kẹt ở điểm không tối ưu

=> Cần chọn learning rate phù hợp để mô hình học hiệu quả và đạt kết quả tốt.

* Tại sao lại không trừ một khoảng vừa khích mà trừ từng khoảng nhỏ?
* Vì trong thực tế hàm mất mát trong mạng nơ-ron không phải một đường thẳng đơn giản mà là một hàm phi tuyến vô cùng phức tạp. Vậy nên chúng ta không thể biết trước nên nhảy một bước bao nhiêu để tới được điểm thấp nhất.
* Và có thể dẫn tới tình trạng quá khớp(overfitting) nghĩa là mô hình học quá kỹ dữ liệu huấn luyện, đến mức nó học cả nhiễu và chi tiết không mang tính khái quát. Điều này khiến cho mô hình hoạt động rất tốt trên tập huấn luyện nhưng quá kém trên dữ liệu mới



1. Mốt số ứng dụng khác
   1. **Ứng dụng trong SQL và cơ sở dữ liệu**
2. **Cơ sở dữ liệu đồ thị (Graph Database)**

Cơ sở dữ liệu đồ thị sử dụng cấu trúc đồ thị với các nút (đại diện cho thực thể) và các cạnh (đại diện cho mối quan hệ) để biểu thị dữ liệu. Điều này cho phép mô hình hóa và truy vấn dữ liệu có mối quan hệ phức tạp một cách hiệu quả. Các ngôn ngữ truy vấn như Gremlin, Cypher hoặc SPARQL được sử dụng để thực hiện các truy vấn trên cơ sở dữ liệu đồ thị. Ví dụ, trong mạng xã hội, việc tìm "bạn của bạn" hoặc đề xuất kết nối mới có thể được thực hiện nhanh chóng nhờ vào cấu trúc đồ thị của dữ liệu .​

1. **Tối ưu hóa truy vấn SQL**

Trong các hệ quản trị cơ sở dữ liệu quan hệ, trình tối ưu hóa truy vấn sử dụng đồ thị tính toán để xác định kế hoạch thực thi hiệu quả nhất cho một truy vấn SQL. Bằng cách biểu diễn các phép toán như JOIN, FILTER và AGGREGATE dưới dạng các nút trong đồ thị, hệ thống có thể xác định thứ tự thực hiện các phép toán sao cho tối ưu về hiệu suất

* 1. **Hệ thống điện tử viễn thông và mạng máy tính**

Đồ thị tính toán (computational graph) đóng vai trò quan trọng trong việc tối ưu hóa và đảm bảo hiệu suất cho các hệ thống viễn thông và mạng máy tính. Bằng cách mô hình hóa các thành phần và kết nối dưới dạng đồ thị, chúng ta có thể xác định đường truyền tối ưu, phân tích tải trọng mạng và đảm bảo tính ổn định của hệ thống.

* 1. **Hoá học và cấu trúc phân tử**

Đồ thị tính toán (computational graph) có ứng dụng quan trọng trong hóa học, đặc biệt trong việc mô hình hóa cấu trúc phân tử. Bằng cách biểu diễn các nguyên tử dưới dạng **nút (nodes)** và các liên kết hóa học dưới dạng **cạnh (edges)**, chúng ta có thể trực quan hóa và phân tích cấu trúc của các phân tử một cách hiệu quả.

* 1. **Sinh học và cây tiên hoá**

Đồ thị tính toán (computational graph) đóng vai trò quan trọng trong sinh học, đặc biệt trong việc mô hình hóa cây tiến hóa (phylogenetic tree). Bằng cách biểu diễn các loài sinh vật và mối quan hệ tiến hóa giữa chúng dưới dạng đồ thị, các nhà khoa học có thể phân tích và hiểu rõ hơn về quá trình tiến hóa và mối quan hệ giữa các loài.

**TÀI LIỆU THAM KHẢO**

<https://www.geeksforgeeks.org/computational-graphs-in-deep-learning/>

<https://websitehcm.com/computation-graph-trong-deep-learning-voi-python/>

<https://itnavi.com.vn/blog/neural-network-la-gi/>

<https://d2l.aivivn.com/chapter_multilayer-perceptrons/backprop_vn.html>

<https://dominhhai.github.io/vi/2018/04/nn-bp/#2-%C4%91%E1%BB%93-th%E1%BB%8B-t%C3%ADnh-to%C3%A1n>

<https://ant.ncc.asia/nhap-mon-machine-learning-phan-5-computational-graph/>

<https://medium.com/@abhishekjainindore24/static-vs-dynamic-computational-graphs-5a49d1e3030b#:~:text=A%20computational%20graph%20is%20a,tensors>

<https://www.youtube.com/watch?v=hM74RH82LyI>

<https://nguyentruonglong.net/giai-thich-chi-tiet-thuat-toan-backpropagation.html>

<https://en.wikipedia.org/wiki/Backpropagation>