**COMPUTATIONAL GRAPH**

CT là gì?

Là đồ thị có hướng, với những node tương ứng với các phép toán hoặc các biến. Các biến có thể cung cấp giá trị của chúng cho các phép toán và các phép toán có thể cung cấp giá trị đầu ra của chúng cho các phép toán khác. Bằng cách này, tất cả các node trong đồ thị này đều là 1 hàm của các biến trong đồ thị.

* CT sử dụng để biễu diễn và tính toán biểu thức toán học.

Ví dụ 1:

Cho phép toán sau:

p = x + y

Hãy vẽ CT dựa trên công thức đã cho.

Diagram

Description automatically generated

Hay:

Diagram

Description automatically generated

Ví dụ 2:

Để đơn giản hơn, ta đưa biểu thức trên về 2 biến d và e. Trong đó và suy ra . Ở đây, chúng ta có 3 phép toán cộng, trừ và nhân. Để tạo ra CT, ta cần tạo ra những node ứng với các biến và các phép toán trên. Hướng của những mũi tên cho ta biết hướng đầu vào của các biến.

Diagram

Description automatically generated

Chúng ta tìm thấy giá trị đầu ra bằng cách khởi tạo các biến đầu vào và tính toán theo các node trong CT.

Các loại CT

Hầu hết các frameworks trong DL đều dựa trên cách tạo ra CT cho việc tính giá trị đạo hàm được yêu cầu để tối ưu hóa truyền đạo hàm. Nói chung, bạn cần xây dựng forward propagation graph và các frameworks sẽ đảm nhận việc backward.

+ CT tĩnh (static CTs)

Thông thường sẽ có 2 bước:

Bước 1: Xác định kích thước graph mà ta muốn xử lý

Ví dụ như ta cho bức ảnh 16\*16 pass qua 10 lớp convolution, tính toán loss và dự đoán lớp của bức ảnh đó.

Bước 2: Chạy tất cả dữ liệu thông qua nó để đào tạo mô hình hoặc đưa ra dự đoán.

Ưu điểm:

* Một trong những lợi ích là cho phép tối ưu hóa, lập lịch sơ đồ ngoại tuyến 1 cách mạnh mẽ.
* Nhanh hơn so với CT động (không có sự khác biệt quá lớn trong nhiều trường hợp và phụ thuộc vào từng sơ đồ).
* Sử dụng được nhiều lần 🡺 tiết kiệm được chi phí.

Nhược điểm:

* Bên cạnh đó, nó không giải quyết được dự liệu quá lớn. Ví dụ: nếu đầu vào không được giới hạn là 16\*16, nó sẽ trở nên khá phức tạp để xác định 1 cấu trúc đơn lẻ của các phép toán đơn giản.
* Đầu vào, ra có cấu trúc khác nhau: 1 trường hợp khá phức tạp là từng dữ liệu đầu vào không chỉ khác nhau về kích thước mà còn khác nhau về cấu trúc. Ví dụ, dữ liệu của bạn là các bức ảnh, khác nhau về văn bản, hình ảnh, bảng,…
* Tuy nhiên, điểm khác nhau đó có thể ngăn chặn bằng cách khai báo 1 graph với kích thước đầu vào không xác định tại thời điểm khai báo,… vì cung cấp hàm dynamic\_rnn trong Tensorflow.
* Mặc dù có thể giải quyết bằng việc khai báo tĩnh nhưng nó vẫn gây ra 1 số khó khăn trong thức tế:
* Độ phức tạp của việc khai triển CT: để hỗ trợ cách giải quyết động, CT phải có khả năng giải quyết các dữ liệu phức tạp và các hàm có sẵn 🡺 điều này làm tăng lên độ phức tạp của việc thực hiện và cách thức của CT, giảm cơ hội tối ưu hóa.
* Khó khăn trong việc gỡ lỗi: Trong khi phân tích tĩnh cho phép xác định 1 số lỗi trong suốt quá trình khai báo, một số lỗi logic sẽ phải chờ phát hiện cho đến khi thực thi (đặc biệt khi nhiều biến không xác định tại thời điểm khai báo) – nhất thiết phải gỡ bỏ phần code khai báo 🡪 dễ dẫn đến việc sửa, xóa phần code sau.

+ CT động (dynamic CTs):

* Được thực hiện chỉ qua 1 bước kỹ thuật. Ví dụ như bức ảnh 16\*16 sau khi được load sẽ chạy qua 2 lớp tích chập và tính toán lỗi trong lúc train và tính toán xác suất dự đoán trong lúc test 🡪 sơ đồ được tạo ra trong từng giai đoạn train nên CT động sẽ rất nhẹ.
* Dễ tìm và fix lỗi do nó cho phép thực thi từng dòng code và bạn được phép truy cập vào từng biến.

Pytorch sử dụng CT động. Tensorflow cho phép tạo sơ đồ tĩnh tối ưu và cũng có thể thực thi với việc cho phép 1 số sơ đồ động tương tự - nó là 1 môi trường lập trình mang tính bắt buộc, tính toán các hàm ngay lập tức, không vẽ sơ đồ, các hàm trả về các giá trị cụ thể thay vì xây dựng đồ thị để chạy sau này.

Hoạt động

Mọi hoạt động được dựa trên 3 yếu tố:

+ compute tính toán giá trị đầu ra của hoạt động từ các đầu vào

+ Danh sách của input\_nodes có thể là giá trị hoặc các toán tử

+ Danh sách của consumers sử dụng chứa các output của hoạt động tương ứng các đầu ra

class Operation:

    """Represents a graph node that performs a computation.

    An `Operation` is a node in a `Graph` that takes zero or

    more objects as input, and produces zero or more objects

    as output.

    """

    def \_\_init\_\_(self, input\_nodes=[]):

        """Construct Operation

        """

        self.input\_nodes = input\_nodes

        # Initialize list of consumers (i.e. nodes that receive this operation's output as input)

        self.consumers = []

        # Append this operation to the list of consumers of all input nodes

        for input\_node in input\_nodes:

            input\_node.consumers.append(self)

        # Append this operation to the list of operations in the currently active default graph

        \_default\_graph.operations.append(self)

    def compute(self):

        """Computes the output of this operation.

        "" Must be implemented by the particular operation.

        """

        Pass

Một số thao tác sơ cấp

Thực hiện 1 số thao tác sơ cấp để trở nên quen với class Operation phía trên

+ Phép cộng

class add(Operation):

    """Returns x + y element-wise.

    """

    def \_\_init\_\_(self, x, y):

        """Construct add

        Args:

          x: First summand node

          y: Second summand node

        """

        super().\_\_init\_\_([x, y])

    def compute(self, x\_value, y\_value):

        """Compute the output of the add operation

        Args:

          x\_value: First summand value

          y\_value: Second summand value

        """

        return x\_value + y\_value

+ Nhân ma trận

class matmul(Operation):

    """Multiplies matrix a by matrix b, producing a \* b.

    """

    def \_\_init\_\_(self, a, b):

        """Construct matmul

        Args:

          a: First matrix

          b: Second matrix

        """

        super().\_\_init\_\_([a, b])

    def compute(self, a\_value, b\_value):

        """Compute the output of the matmul operation

        Args:

          a\_value: First matrix value

          b\_value: Second matrix value

        """

        return a\_value.dot(b\_value)

* Cả 2 thao tác trên, ta thấy rằng các tensors là các mảng Numpy, trong đó phép cộng theo phần tử và phép nhân ma trận được thực hiện

CT trong Pytorch

Về cốt lõi, Pytorch cung cấp 2 đặc trưng:

+ Tensor n chiều tương tự như Numpy nhưng có thể chạy trên GPUs.

+ Phân biệt tự động cho quá trình build và train neraul networks.

Cấu trúc DL và sự train của chúng ta bao gồm các hàm ma trận. 1 Tensor không là gì khác ngoài 1 mảng n chiều. Đối với những người học Python, Numpy sẽ rất gần gũi. Nó là 1 thư viện rất mạnh và tối ưu cho các hoạt động ma trận. Tuy nhiên, với mục tiêu DL, ma trận rất rộng lớn và yêu cầu sự tính toán khổng lồ.

Pytorch Tensor không khác gì mảng n chiều. Framework cung cấp các chức năng để hoạt động trên các Tensor. Nhưng tang nhanh sự tính toán cho các Tensor, PyTorch cho phép sự dụng GPUs, nó có thể cung cấp tốc độ nhanh gấp 50 lần. PyTorch Tensor cũng có thể theo dõi đồ thị tính toán và đạo hàm.

Trong Pytoch, gói autograd cung cấp khả năng phân biệt tự động để tự động tính toán backward pass trong neural network. Forward pass trong network của bạn là CT, các node trong graph là các tensor và các cạnh là các hàm tạo ra các Tensor đầu ra từ các Tensor đầu vào. Backpropagation thông qua graph này sau đó qua đạo hàm.

Mọi Tensor trong Pytorch là 1 cờ: required\_grapd cho phép loại trừ chi tiết các đồ thị con từ việc tính toán đạo hàm và có thể tang hiệu quả. Nếu x là 1 Tensor có x.requires\_grad = True thì x.grad à 1 Tensor khác giữ đạo hàm của x với 1 giá trị vô hướng nào đó.

|  |
| --- |
| import torch |
|  | x = torch.randn(3,3) #requires\_grad=False by default |
|  | y = torch.randn(3,3) #requires\_grad=False by default |
|  | z = torch.randn((3,3),requires\_grad=True) |
|  | a = x+y # since both x and y don't require gradients, a also doesn't require gradients |
|  | print(a.requires\_grad) #output: False |
|  | b = a+z #since z requires gradient, b also requires gradient |
|  | print(b.requires\_grad) #output: True |

Như thấy ở ví dụ trên, nếu đó là đầu vào duy nhất để hoạt động yếu cầu đạo hàm, thì đầu ra cũng sẽ yêu cầu đạo hàm. Ngược lại, chỉ khi tất cả các đầu vào không yêu cầu đạo hàm, đầu ra cũng sẽ không yêu cầu.

Autograd under the hood

Diagram

Description automatically generated

CT trong Tensorflow

Tensorflow sử dụng dataflow graph để biểu diễn tính toán về sự phụ thuộc các hoạt động riêng lẻ. Nó dẫn đến mô hình lập trình mức độ thấp, trond đó người ta xác định dataflow graph, sau đó tạo ra 1 phiên Tensorflow để chạy từng phần của graph trên 1 vùng cục bộ và toàn cục.

Ví dụ về dataflow hoặc CT trong Tensorflow:

Diagram

Description automatically generated

Trong Tensorflow, bất kể loại tính toán cũng được biểu diễn dưới dạng đối tượng tf.Graph. Đối tượng này bao gồm tf.Tensor và tf.Operation. Trong Tensorflow, đối tượng tf.Tensor phục vụ cho cạnh trong khi tf.Operation phục vụ cho node được add vào bằng tf.Graph.

Trong Tensorflow, tf.Session() chứa context dưới dạng phép toán được thực hiện. Nó là 1 class cho việc chạy Tensorflow.