Differentiable Graph

Yiqing Li

July 9, 2019

Contents

1 Introduction

The main topic of this article is differentiable graph and how it's used to model machine learning models like neural networks. A differentiable graph is a directed acyclic graph where each node/operator represents an operation/computation and each directed edge defines the data flow. Each node's output is a vector/tensor and its inputs are outputs from one or more other nodes. Meanwhile the function each node represents is differentiable, that is how the name comes from. For example figure 1 represents a common logistic regression model where the last operator is the loss which is the sum of cross entropy error plus $L2$ regularization of the weights.

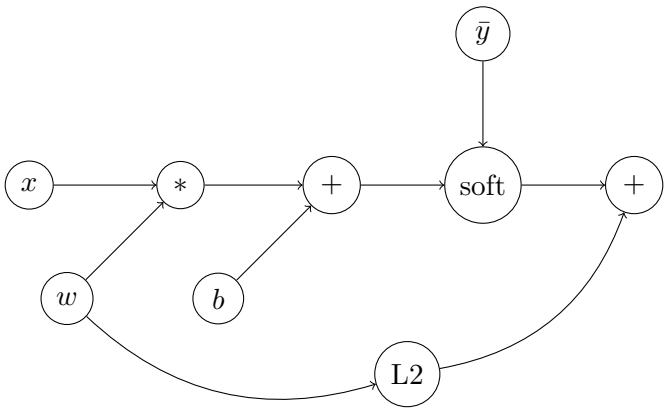

Figure 1: a differentiable graph example

If an operator represents the function $y = f(x)$, then $\frac{\partial y_i}{\partial x_j}$ essentially-exists for all i and j. Since all operators are differentiable, the gradient of one output variable of node A w.r.t. one output variable of node B exists. The gradient can be calculated analytically by applying chain rule following the data path from node B to node A. In practice, back propagation algorithm is used to calculate gradients. The question is if we know the gradients of a particular value L w.r.t the output y, can we calculate the gradient w.r.t the input x? The answer is yes and it equals

$$\frac{\partial L}{\partial x_i} = \sum_j \frac{\partial L}{\partial y_j} \frac{\partial y_j}{\partial x_i} \tag{1}$$

Back propagation is essentially applying this formula one by one following the backward data path.

Differentiable graph is very powerful in modeling complex computations as long as the computation consists of many elementary operations chained together. This is often true for most of the machine learning models. In machine learning, another important task is to do optimization over one particular output or loss. For example in figure 1, the goal is to minimize the output value of the last node. Back propagation algorithm is computationally convenient here to calculate gradients of loss w.r.t weight and bias. Then gradient descent algorithm can be applied to update those weights and biases. For feed forward neural networks it's no more than stacking many graphs like in figure 1. For recurrent neural networks they can also be modeled as a differentiable graph with weights shared by many homogeneous operators.

Notice that differentiable graph is a general setting where gradients does not only live on weights, but also live on any node from the graph. One can also calculate gradients with respect to input data, which is used in many applications like constructing adversarial examples

to fool machine learning models.

2 Design

Modeling computations in terms of graph is a good news for implementing them in program languages. We only need to define all the basic differentiable operators and treat them as building blocks of a differentiable graph. In neural networks one often do batched training, but this does not affect the defining of differentiable graph once the loss is taking account of the batch size.

Here we propose a python-written package that implements the idea, it has the following design:

1. Graph layer: where graph manipulations like breath first search, depth first search, subgraph, backward pass, forward pass and etc. are defined.
2. Operator layer: operators like matrix addition, matrix multiplication, softmax and so on are defined.
3. Optimization layer, the training process consists of the following steps:
   (a) Pick an objective node and a subgraph rooted at the node.
   (b) Randomly choose a batch of training data and feed to the network and do a forward pass.
   (c) Do a backward pass and accumulate gradients.
   (d) Update the weights.
   (e) Repeat steps from. 3b to 3d

3 Some operator

Once the infrastructure, namely the graph, has been defined, it is next to define operators that build the graph. If we want to build a Convolutional Neural Network, we can build a convolution operator, a max pooling operator, a ReLU operator and chain them together along with other neural network components like fully connected operator and sofmax operator(figure 2).

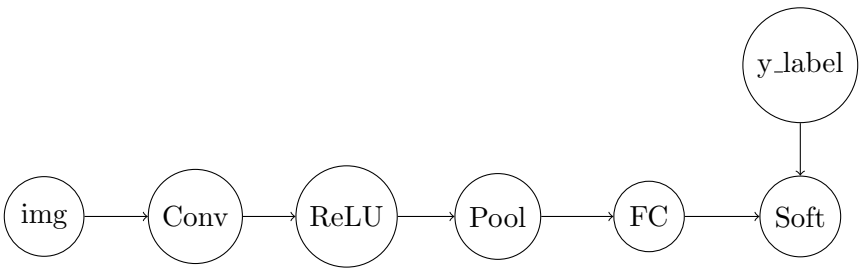

Figure 2: a differentiable graph example

For an operator, the main API it needs to implement is the forward() and backward() methods. The forward() method is mapping the inputs to the output while the backward() method is to

calculate gradient w.r.t inputs given the gradients w.r.t output.

In practice, some operators may get very complicated(like the convolution operator), the following principles can be applied to ease the process.

1. If $x = (x^1, ...x^n)$, $y = (y^1, ..., y^n)$ and y_i and x_j are independent for $i \neq j$, only need to compute $\frac{\partial y^i}{\partial x^i}$ independently.

2. If the operator f is a linear combination of $f = a_1 f_1(x) + ...a_n f_n(x)$, $\frac{\partial y_i}{\partial x_j} = a_1 \frac{\partial y_i^1}{\partial x_j} + ...a_n \frac{\partial y_i^n}{\partial x_j}$.

When it comes to implement forward and backward, try to vectorize the operation and implement it using native operators in numpy.

3.1 Tensor Addition

Forward pass is clear for tensor addition, the backward pass is just copying the output gradients to both input tensors. If one tensor is batched, need to sum the output gradient on axis 0 for the other not-batched tensor. Why? It is like finding derivative of $ax + bx$ w.r.t x, needs to sum a and b. The basic principle is to write out the formula for the operator and calculate equation (1).

3.2 Matrix Multiplication

The forward pass for matrix multiplication is trivial. If the matrices involved has the following dimension

$$C_{m \times p} = A_{m \times n} B_{n \times p} \tag{2}$$

Let $\mathbf{C}$ be $\frac{\partial L}{\partial C}$, which is of shape $m \times p$ the same as C. Now fix A_{ij} and we have

$$\frac{\partial L}{\partial A_{ij}} = \sum_l^m \sum_d^p \frac{\partial L}{\partial C_{ld}} \frac{\partial C_{ld}}{\partial A_{ij}} \tag{3}$$

Since $C_{ld} = \sum_t A_{lt} B_{td}$, we have

$$\frac{\partial C_{ld}}{\partial A_{ij}} = \begin{cases} 0 & \text{if } l \neq i \\ B_{jd} & \text{if } l \neq i \end{cases} \tag{4}$$

Thus

$$\frac{\partial L}{\partial A_{ij}} = \sum_l^m \sum_d^p \mathbf{C}_{ld} \frac{\partial C_{ld}}{\partial A_{ij}} \tag{5}$$

$$= \sum_d^p \mathbf{C}_{id} B_{jd} \tag{6}$$

$$= \sum_d^p \mathbf{C}_{id} B_{jd} \tag{7}$$

$$= \sum_d^p \mathbf{C}_{id} B_{dj}^T \tag{8}$$

$$= \mathbf{C} B^T \tag{9}$$

where B^T is transpose of B.

References