Deep Learning Software

Nan Ye

School of Mathematics and Physics
The University of Queensland

Gradient-phobia

- Backprop provides an efficient way to compute the gradient of the error function for a neural network.
- It is helpful to understand the algorithm but it is not easy to implement
  - you probably never want to implement it if you don't have to
  - in the old days, it is not unusual that people spent hours to derive expressions for the gradients, and then hours for implementation and debugging...

- The good news is that for practical purposes, you don't have to implement gradient computation for neural nets
  - Many machine learning software platforms now provide automatic differentiation (autodiff) tools
  - Autodiff automatically compute the gradients for you — you only need to write code to evaluate the function

Software Frameworks

TensorFlow

theano

mxnet

Caffe2

GLUON

Sonnet

PyTorch

DL4J

Keras

ConvNetJS
Deep Learning in your browser

scikit learn

SWIFT

- Some early software frameworks have become obsolete (e.g. Theao, Caffe)
- TensorFlow, originally developed by Google Brain Team, is the most popular deep learning frameworks, with a few high-level API built on top of it (e.g. Sonnet, Keras, Swift, TFLearn)
- PyTorch, developed by Facebook, is a more recent player, but has become a main competitor of TensorFlow.
    - Simple and flexible
    - We will discuss PyTorch in this lecture.

Automatic Gradient Computation

- Deep learning algorithms mostly use gradient-based learning.
- A key building block of deep learning frameworks is the support for automatic gradient computation.
- There are three main approaches to do this
  - Numerical differentiation (or finite differencing)
  - Symbolic differentiation
  - Automatic differentiation (autodiff, or algorithmic differentiation)

Numerical differentiation

- If we have implemented the function $f(\mathbf{w})$, then we can numerically compute its gradient by choosing a small δ, and compute each partial derivative using

$$\frac{\partial f(\mathbf{w})}{\partial w_i} \approx \frac{f(\mathbf{w} + \delta \mathbf{e}_i) - f(\mathbf{w})}{\delta},$$

  where $\mathbf{e}_i$ is the i-th standard unit vector.
- This is easy to implement, but approximate and slow.

Symbolic differentiation

- We represent a function symbolically, and apply differentiation rules to generate symbolic representation of its gradient.
- For example, if $f(a, b) = a^2 b + ab^2$, a direct application of differentiation rules lead to $\frac{\partial f}{\partial a} = 2ab + a^2 0 + 1b^2 + a0$.
- Symbolic differentiation can lead to large symbolic representations and inefficient code.
  - e.g. consider $f_{100}(x)$ defined recursively by $f_1(x) = x$, and $f_{k+1}(x) = e^{x^2 + f_k(x)}$ for $k \geq 1$.

Automatic differentiation (Autodiff)

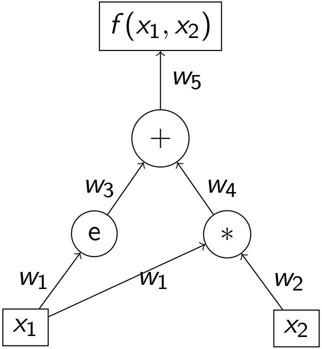
- Autodiff transforms the code for evaluating the function to the code for evaluating the gradient.
- The computation for the function is broken down into a composition of elementary operations, and then chain rule is repeatedly applied to these operations.
- This uses the concept of computational graph, and can be done in *forward mode* or *reverse mode*.
- How is this different from symbolic computation?
    - We pass values around, not symbols.

Autodiff

Computational graph

- Consider the function $f(x_1, x_2) = e^{x_1} + x_1 x_2$.
- We can break down its computation as shown in the table below, and represent it using a computational graph.

$$
\begin{aligned}
w_1 &= x_1 \\
w_2 &= x_2 \\
w_3 &= e^{w_1} \\
w_4 &= w_1 w_2 \\
w_5 &= w_3 + w_4
\end{aligned}
$$



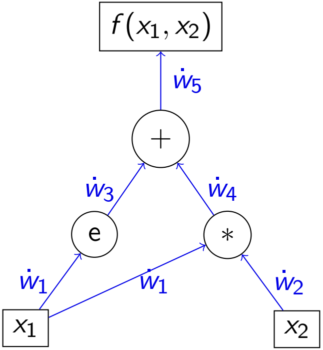
- Each node stores its output and passes it forward (bottom-up in our example).

Forward mode

- Consider computing $\frac{\partial f}{\partial x_1}$.
- In forward mode autodiff (aka forward accumulation), we recursively compute each $\dot{w}_i = \frac{\partial w_i}{\partial x_1}$ using the chain rule.
  - $\dot{w}_5$ is our target $\frac{\partial f}{\partial x_1}$.
  - E.g. $w_3 = e^{w_1} \Rightarrow \frac{\partial w_3}{\partial x_1} = e^{w_1} \frac{\partial w_1}{\partial x_1} \Rightarrow \dot{w}_3 = e^{w_1} \dot{w}_1$.
- This requires traversing the graph in the forward direction (bottom-up in our example).

- The recursive computation is shown in the table and the computational graph below.

$$
\begin{aligned}
\dot{w}_1 &= 1 \\
\dot{w}_2 &= 0 \\
\dot{w}_3 &= e^{w_1}\dot{w}_1 \\
\dot{w}_4 &= \dot{w}_1 w_2 + w_1 \dot{w}_2 \\
\dot{w}_5 &= \dot{w}_3 + \dot{w}_4
\end{aligned}
$$



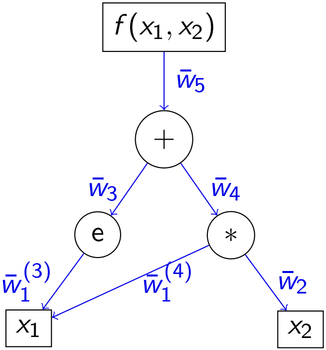
- We pass both the output and its derivative for each node.
- Note that the intermediate results are values (not symbols).

Reverse mode

- Consider computing $\frac{\partial f}{\partial x_1}$ and $\frac{\partial f}{\partial x_2}$.
- In reverse mode autodiff (aka backward accumulation), we recursively compute $\bar{w}_i = \frac{\partial f}{\partial w_i}$ using the chain rule.
  - $\bar{w}_1$ and $\bar{w}_2$ are our targets $\frac{\partial f}{\partial x_1}$ and $\frac{\partial f}{\partial x_2}$ respectively.
  - E.g. f depends on w_1 vias $w_3 = e^{w_1}$ and $w_4 = w_1 w_2$ $\Rightarrow \frac{\partial f}{\partial w_1} = \frac{\partial f}{\partial w_3}\frac{\partial w_3}{\partial w_1} + \frac{\partial f}{\partial w_4}\frac{\partial w_4}{\partial w_1} \Rightarrow \bar{w}_1 = \bar{w}_3 e^{w_1} + \bar{w}_4 w_2$.
- This requires traversing the graph in the backward direction (top-down in our example).

- The recursive computation is shown in the table and the computational graph below.

$$
\begin{aligned}
\bar{w}_5 &= 1 \\
\bar{w}_4 &= \bar{w}_5 \\
\bar{w}_3 &= \bar{w}_5 \\
\bar{w}_2 &= \bar{w}_4 w_1 \\
\bar{w}_1^{(3)} &= \bar{w}_3 e^{w_1} \\
\bar{w}_1^{(4)} &= \bar{w}_4 w_2 \\
\bar{w}_1 &= \bar{w}_1^{(3)} + \bar{w}_1^{(4)}
\end{aligned}
$$



- Note that the intermediate results are values (not symbols).
- Backprop is a special case of reverse mode autodiff.

PyTorch

- PyTorch has a very polished Python interface, and a C++ frontend.
- PyTorch provides great support for
  - Tensor computing (like NumPy), with strong GPU acceleration
  - Deep neural networks, based on autodiff.
- See `https://pytorch.org/` for details including installation instructions, tutorials, and documentation.

Neural Networks in PyTorch

- PyTorch provides several packages
  - `torch`: a general-purpose tensor package with GPU support
  - `torch.autograd`: a package for automatic differentiation
  - `torch.nn`: a neural net library with common layers and loss functions
  - `torch.optim`: contains common optimization algorithms
- We cover basics of these packages in this lecture.

Tensor Computation

```
import torch
x, y, z = torch.zeros(3, 3), torch.ones(3, 3), torch.rand(3, 3)
print(x, y, z)
print(x + y)
print(y @ z) # matrix multiplication
print(z.int()) # convert to integer array
print(z.numpy()) # convert to numpy array
if torch.cuda.is_available(): # use GPU if available
  y, z = y.cuda(), z.cuda()
  print(y @ z)
```

Autodiff for $f(\mathbf{x}) = \|\mathbf{x}\|_2^2$

```
def f(x):
  return torch.dot(x, x)

x = torch.ones(2, requires_grad=True)
y = f(x)

# use the autograd library to compute all gradient information
y.backward()

# print the gradient of the function with respect to x
print(x.grad)
```

Exercise: try replacing f with your favourite function.

OLS using PyTorch

Data

```
def regression_data(n=500, d=2):
  X = torch.rand(n, d)
  beta = torch.rand(d+1)
  Y = torch.mv(X, beta[1:]) + beta[0] + torch.rand(n) * 0.1
  return X, Y

X, Y = regression_data()
```

- The output is a perturbed linear function of the inputs.

First version (exploit autograd)

```
X = torch.cat([torch.ones(X.shape[0], 1), X], dim=1) # add 1
beta = torch.zeros(X.shape[1], requires_grad=True)

for i in range(200):
  loss = torch.mean((X @ beta - Y)**2)
  if beta.grad is not None:
      beta.grad.zero_() # important: reset the stored gradient to 0
  loss.backward()
  beta.data.add_(-0.5*beta.grad.data)

print(beta)
```

- We only use the autodiff feature in PyTorch, but control all other aspects.
- Exercise: try the above code and use the closed-form formula to compute β. Do you get the same answers? (You should)

Second version (exploit optim)

```
import torch.optim as optim

X = torch.cat([torch.ones(X.shape[0], 1), X], dim=1) # add 1
beta = torch.zeros(X.shape[1], requires_grad=True)

optimizer = optim.SGD([beta], lr=0.5, momentum=0)
for i in range(200):
    optimizer.zero_grad()
    loss = torch.mean((X @ beta - Y)**2)
    loss.backward()
    optimizer.step()

print(beta)
```

- We use the SGD optimizer provided by the optim package to zero gradient and perform gradient update.

Third version (exploit nn and built-in loss functions)

```
import torch.optim as optim
import torch.nn as nn
from torch.nn.modules.loss import MSELoss

Y = Y.reshape(-1, 1)

net = nn.Linear(2, 1)
optimizer = optim.SGD(net.parameters(), lr=0.5, momentum=0)
mse = MSELoss()
for i in range(200):
    optimizer.zero_grad()
    loss = mse(net(X), Y)
    loss.backward()
    optimizer.step()

for param in net.parameters():
    print(param)
```

- We use the nn module to define our neural net for OLS, and use the builtin MSE loss to compute loss.

More on PyTorch

Defining a general MLP

- We can use the nn module to define general MLPs. For example, if we want to replace the OLS network using a single ReLU hidden layer MLP, we can define the network as follows

```
net = nn.Sequential(nn.Linear(2, 10),
                    nn.ReLU(),
                    nn.Linear(10, 1))
```

- Exercise: try training the above neural on the toy dataset.
- The nn module also implements many other activation funtions. See the *Non-linear Activations* sections at https://pytorch.org/docs/stable/nn.html.

Using DataLoader to load mini-batches

- We often use SGD to train neural nets. This requires us to split the dataset into mini-batches and loop through them.
- This code below illustrates how to to this.

```
class DatasetWrapper(Dataset):
    def __init__(self, X, y):
        self.X, self.y = X, y

    def __len__(self):
        return len(self.X)

    def __getitem__(self, idx):
        return self.X[idx], self.y[idx]

data_loader = DataLoader(DatasetWrapper(X, y), batch_size=10,
    shuffle=True)

for i, (X, y) in enumerate(data_loader):
    print(i, X.shape, y.shape)
```

Your Turn

Which of the following statement is correct? (Multiple choice)

(a) It is generally easy to manually work out the formula for the gradient of the loss of a neural net, and implement it from scratch.

(b) Autodiff allows us to implement one, get one free (implement function evaluation code, get gradient evaluation code free).

(c) PyTorch supports tensor computing and deep neural nets.

What You Need to Know...

- Automatic gradient computation approaches
  - Numerical differentiation
  - Symbolic differentiation
  - Automatic differentiation: forward mode, reverse mode
- Several key PyTorch packages
  - `torch`, `torch.nn`, `torch.autograd`, `torch.optim`