CS-568 Deep Learning

Nazar Khan

PUCIT

Automatic Differentiation

Automatic Differentiation (AD)

- Set of techniques to numerically evaluate the derivative of a function *specified by a computer program*.
- Analytic or symbolic differentiation evaluates the derivative of a function *specified by a math expression*.
- Also called *algorithmic differentiation* or *computational differentiation*.
- Backpropagation is a special case of AD.

Modern machine learning frameworks (TensorFlow, Theano, PyTorch) employ AD. The programmer only needs to implement the loss function. Derivatives are handled automatically!

Types of AD

- AD is just unrolling of the chain rule.
- Can be unrolled in two ways,
  1. Forward:

$$\frac{\partial y}{\partial x} = \frac{\partial y}{\partial w_{n-1}} \frac{\partial w_{n-1}}{\partial x} = \frac{\partial y}{\partial w_{n-1}} \left(\frac{\partial w_{n-1}}{\partial w_{n-2}} \frac{\partial w_{n-2}}{\partial x} \right)$$
$$= \frac{\partial y}{\partial w_{n-1}} \left(\frac{\partial w_{n-1}}{\partial w_{n-2}} \left(\frac{\partial w_{n-2}}{\partial w_{n-3}} \frac{\partial w_{n-3}}{\partial x} \right) \right) = \cdots$$

  2. Reverse:

$$\frac{\partial y}{\partial x} = \frac{\partial y}{\partial w_1} \frac{\partial w_1}{\partial x} = \left(\frac{\partial y}{\partial w_2} \frac{\partial w_2}{\partial w_1} \right) \frac{\partial w_1}{\partial x}$$
$$= \left(\left(\frac{\partial y}{\partial w_3} \frac{\partial w_3}{\partial w_2} \right) \frac{\partial w_2}{\partial w_1} \right) \frac{\partial w_1}{\partial x} = \cdots$$

- The idea is to accumulate required derivatives.

Forward Accumulation AD

- Consider the function

$$\begin{aligned} z = f(x_1, x_2) &= x_1 x_2 + \sin x_1 \\ &= w_1 w_2 + \sin w_1 \\ &= w_3 + w_4 = w_5 \end{aligned}$$

- Consider derivatives with respect to x_1. Let $\dot{w}_i = \frac{\partial w_i}{\partial x}$.
- For computing $\frac{\partial z}{\partial x_1}$, we first compute the *seed values*

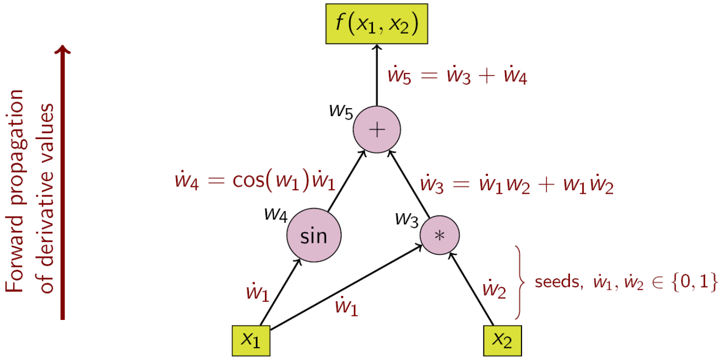
$$\begin{aligned} \dot{w}_1 &= \frac{\partial x_1}{\partial x_1} = 1 \\ \dot{w}_2 &= \frac{\partial x_2}{\partial x_1} = 0 \end{aligned}$$

- These seed values can be propagated using the *chain rule*.

Forward Accumulation AD

Operations to compute value	Operations to compute derivative
$w_1 = x_1$	$\dot{w}_1 = 1$ (seed)
$w_2 = x_2$	$\dot{w}_2 = 0$ (seed)
$w_3 = w_1 \cdot w_2$	$\dot{w}_3 = w_2 \cdot \dot{w}_1 + w_1 \cdot \dot{w}_2$
$w_4 = \sin w_1$	$\dot{w}_4 = \cos w_1 \cdot \dot{w}_1$
$w_5 = w_3 + w_4$	$\dot{w}_5 = \dot{w}_3 + \dot{w}_4$

Forward Accumulation AD



- For computing $\frac{\partial z}{\partial x_2}$, propagate *again* with seed values $\dot{w}_1 = 0$ and $\dot{w}_2 = 1$.
- Number of forward *sweeps* is equal to number of inputs.
- So forward AD is efficient when output size is much larger than input size.

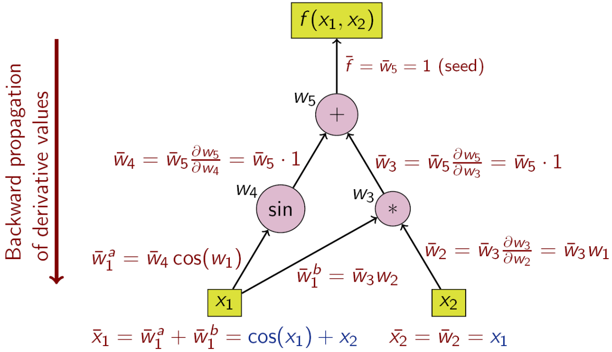
Reverse Accumulation AD

- Fix the *dependent variable* and compute the derivative w.r.t each sub-expression recursively.
  $$\frac{\partial y}{\partial x} = \frac{\partial y}{\partial w_1}\frac{\partial w_1}{\partial x} = \left(\frac{\partial y}{\partial w_2}\frac{\partial w_2}{\partial w_1}\right)\frac{\partial w_1}{\partial x} = \left(\left(\frac{\partial y}{\partial w_3}\frac{\partial w_3}{\partial w_2}\right)\frac{\partial w_2}{\partial w_1}\right)\frac{\partial w_1}{\partial x} = \cdots$$
- Define the *adjoint* $\bar{w}_i = \frac{\partial y}{\partial w_i}$ as the derivative w.r.t sub-expression w_i.
- Notice the similarity with $\delta_j = \frac{\partial L}{\partial a_j}$ in back-propagation.

Reverse Accumulation AD

Operations to compute value	Operations to compute derivative
$z_5 = w_5$	$\bar{w}_5 = 1$ (seed)
$w_5 = w_3 + w_4$	$\bar{w}_4 = \bar{w}_5$
$w_5 = w_3 + w_4$	$\bar{w}_3 = \bar{w}_5$
$w_3 = w_1 \cdot w_2$	$\bar{w}_2 = \bar{w}_3 \cdot w_1$
$w_4 = \sin w_1$ and $w_3 = w_1 \cdot w_2$	$\bar{w}_1 = \bar{w}_3 \cdot w_2 + \bar{w}_4 \cdot \cos w_1$

Reverse Accumulation AD



- Number of reverse *sweeps* is equal to number of outputs.
- So reverse AD is efficient when input size is much larger than output size. This is usually the case for classification problems.

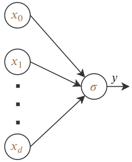
AD in Python

- A Python package called *Autograd* implements *reverse mode* automatic differentiation.
- Elementary operations such as $+, \sin, x^k$ *etc.* are *overloaded* by also computing their derivates $1, \cos, kx$ *etc.*.
- If required, user-defined complex functions and their derivative implementations can be *registered* with Autograd.

Logistic Regression via Automatic Differentiation

Binary classifier with no hidden layer

Just a perceptron with logistic sigmoid activation function. Models probability of class 1 instead of decision.



$$y = p(\mathcal{C}_1|\mathbf{x}) = \sigma(\mathbf{w}^T\mathbf{x})$$
$$1 - y = p(\mathcal{C}_2|\mathbf{x}) = 1 - p(\mathcal{C}_1|\mathbf{x})$$

Binary cross-entropy loss

$$L(\mathbf{w}) = -\sum_{n=1}^{N} t_n \ln y_n + (1 - t_n) \ln (1 - y_n)$$

What is $\ln(0)$? $\ln(1)$?
Plot $\ln(y)$ for $y > 0$?
Plot $\ln(y)$ for $0 < y \leq 1$?
When is $L(\mathbf{w}) = 0$?
Can $L(\mathbf{w})$ be negative?

Logistic Regression via Automatic Differentiation

Binary classifier with no hidden layer

```
import pylab
import sklearn.datasets
import autograd.numpy as np
from autograd import grad

# Generate the data
train_X, train_y = sklearn.datasets.make_moons(500, noise=0.1)

# Define the activation, prediction and loss functions for Logistic Regression
def activation(x):
    return 0.5*(np.tanh(x) + 1)

def predict(weights, inputs):
    return activation(np.dot(inputs, weights))

def loss(weights):
    preds = predict(weights, train_X)
    label_probabilities = np.log(preds) * train_y + np.log(1 - preds) * (1 - train_y)
    return -np.sum(label_probabilities)

# Compute the gradient of the loss function
gradient_loss = grad(loss)

# Set the initial weights
weights = np.array([1.0, 1.0])

# Steepest Descent
```

Logistic Regression via Automatic Differentiation

Binary classifier with no hidden layer

```
loss_values = []
learning_rate = 0.001
for i in range(100):
    loss_values.append(loss(weights))
    step = gradient_loss(weights)
    weights -= step * learning_rate

# Plot the decision boundary
x_min, x_max = train_X[:, 0].min() - 0.5, train_X[:, 0].max() + 0.5
y_min, y_max = train_X[:, 1].min() - 0.5, train_X[:, 1].max() + 0.5
x_mesh, y_mesh = np.meshgrid(np.arange(x_min, x_max, 0.01), np.arange(y_min, y_max, 0.01))
Z = predict(weights, np.c_[x_mesh.ravel(), y_mesh.ravel()])
Z = Z.reshape(x_mesh.shape)
cs = pylab.contourf(x_mesh, y_mesh, Z, cmap=pylab.cm.Spectral)
pylab.scatter(train_X[:, 0], train_X[:, 1], c=train_y, cmap=pylab.cm.Spectral)
pylab.colorbar(cs)

# Plot the loss over each step
pylab.figure()
pylab.plot(loss_values)
pylab.xlabel("Steps")
pylab.ylabel("Loss")
pylab.show()
```