Applied Machine Learning

Gradient Computation & Automatic Differentiation

Reihaneh Rabbany

McGill School of Computer Science

COMP 551 (winter 2020)

Learning objectives

using the chain rule to calculate the gradients

automatic differentiation

- forward mode
- reverse mode (backpropagation)

Landscape of the cost function

model two layer MLP

$$f(x; W, V) = g(Wh(Vx))$$

there are **exponentially many** global optima:
given one global optimum we can

- permute hidden units in each layer
- for symmetric activations: negate input/ouput of a unit
- for rectifiers: rescale input/output of a unit

general beliefs
supported by empirical and theoretical results in a special settings

many more saddle points than local minima

number of local minima increases for lower costs

therefore most local optima are close to global optima

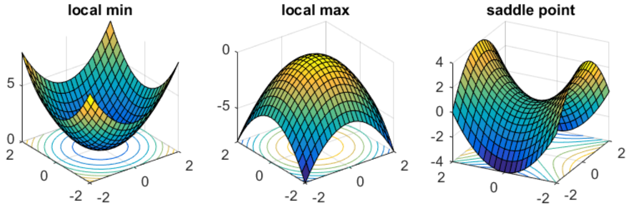
strategy use gradient descent methods (covered earlier in the course)

objective $\min_{W,V} \sum_n L(y^{(n)}, f(x^{(n)}; W, V))$

loss function depends on the task

this is a non-convex optimization problem

many critical points (points where gradient is zero)



these are not stable and SGD can escape

image credit: https://www.offconvex.org

Jacobian matrix

$f : \mathbb{R} \to \mathbb{R}$ we have the derivative $\frac{d}{dw} f(w) \in \mathbb{R}$

$f : \mathbb{R}^D \to \mathbb{R}$ **gradient** is the vector of all partial derivatives

$$\nabla_w f(w) = [\frac{\partial}{\partial w_1} f(w), \ldots, \frac{\partial}{\partial w_D} f(w)]^\top \in \mathbb{R}^D$$

$f : \mathbb{R}^D \to \mathbb{R}^M$ the **Jacobian matrix** of all partial derivatives

$$J = \begin{array}{cc} & \begin{matrix} \frac{\partial}{\partial w_1} f(w) & & \phantom{\frac{\partial f_1(w)}{\partial w_D}} \end{matrix} \\ \begin{matrix} \nabla_w f_1(w) \\ \\ \\ \end{matrix} & \begin{bmatrix} \frac{\partial f_1(w)}{\partial w_1}, & \ldots, & \frac{\partial f_1(w)}{\partial w_D} \\ \vdots & \ddots & \vdots \\ \frac{\partial f_M(w)}{\partial w_1}, & \ldots, & \frac{\partial f_M(w)}{\partial w_D} \end{bmatrix} \end{array} \in \mathbb{R}^{M \times D}$$

note that we use J also for cost function

for all three case we may simply write $\frac{\partial}{\partial w} f(w)$, where M,D will be clear from the context

what if W is a matrix? we assume it is reshaped it into a vector for these calculations

Chain rule

for $f : x \mapsto z$ and $h : z \mapsto y$ where $x, y, z \in \mathbb{R}$

$$\frac{dy}{dx} = \frac{dy}{dz}\frac{dz}{dx}$$

speed of change in z as we change x

speed of change in y as we change z

speed of change in y as we change x

more generally $x \in \mathbb{R}^D, z \in \mathbb{R}^M, y \in \mathbb{R}^C$ $\frac{\partial y_c}{\partial x_d} = \sum_{m=1}^{M} \frac{\partial y_c}{\partial z_m}\frac{\partial z_m}{\partial x_d}$

we are looking at all the "paths" through which change in x_d changes y_c and add their contribution

in matrix form $\frac{\partial y}{\partial x} = \frac{\partial y}{\partial z}\frac{\partial z}{\partial x}$

C x D Jacobian

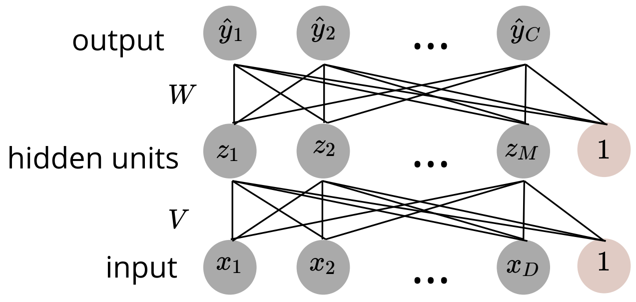
M x D Jacobian

C x M Jacobian

Training a two layer MLP

suppose we have

- D inputs $x_1, \ldots, x_D$
- C outputs $\hat{y}_1, \ldots, \hat{y}_C$
- M hidden *units* $z_1, \ldots, z_M$



for simplicity we drop the bias terms

model $\hat{y} = g\big(W\, h(V\, x)\big)$

Cost function we want to minimize

$$J(W, V) = \sum_n L(y^{(n)}, g\,(\,W\;\; h\,(\,V\;\; x^{(n)}\,)\,))$$

need gradient wrt W and V: $\frac{\partial}{\partial W} J,\; \frac{\partial}{\partial V} J$

simpler to write this for one instance (n)

so we will calculate $\frac{\partial}{\partial W} L,\; \frac{\partial}{\partial V} L$ and recover $\frac{\partial}{\partial W} J = \sum_{n=1}^{N} \frac{\partial}{\partial W} L(y^{(n)}, \hat{y}^{(n)})$ and $\frac{\partial}{\partial V} J = \sum_{n=1}^{N} \frac{\partial}{\partial V} L(y^{(n)}, \hat{y}^{(n)})$

Gradient calculation

using the chain rule

$$\frac{\partial}{\partial W_{c,m}} L = \frac{\partial L}{\partial \hat{y}_c} \frac{\partial \hat{y}_c}{\partial u_c} \frac{\partial u_c}{\partial W_{c,m}}$$

depends on the loss function

depends on the activation function

z_m

similarly for V

$$\frac{\partial}{\partial V_{m,d}} L = \sum_c \frac{\partial L}{\partial \hat{y}_c} \frac{\partial \hat{y}_c}{\partial u_c} \frac{\partial u_c}{\partial z_m} \frac{\partial z_m}{\partial q_m} \frac{\partial q_m}{\partial V_{m,d}}$$

depends on the loss function

$W_{c,m}$

x_d

depends on the activation function

depends on the middle layer activation

$$L(y, \hat{y})$$

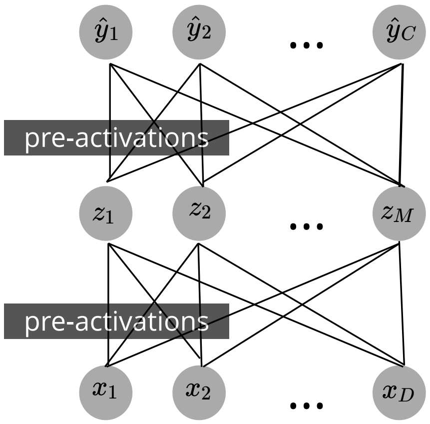
$$\hat{y}_c = g(u_c)$$

$$u_c = \sum_{m=1}^{M} W_{c,m} z_m$$

$$z_m = h(q_m)$$

$$q_m = \sum_{d=1}^{D} V_{m,d} x_d$$

$$x_d$$



Gradient calculation

using the chain rule

$$\frac{\partial}{\partial W_{c,m}} L = \frac{\partial L}{\partial \hat{y}_c} \frac{\partial \hat{y}_c}{\partial u_c} \frac{\partial u_c}{\partial W_{c,m}}$$

depends on the loss function

depends on the activation function

z_m

regression

$$\begin{cases} \hat{y} = g(u) = u = Wz \\ L(y, \hat{y}) = \frac{1}{2}||y - \hat{y}||_2^2 \end{cases}$$

substituting

$$L(y, z) = \frac{1}{2}||y - Wz||_2^2$$

taking derivative

$$\frac{\partial}{\partial W_{c,m}} L = (\hat{y}_c - y_c) z_m$$ we have seen this in linear regression lecture

Gradient calculation

using the chain rule

$$\frac{\partial}{\partial W_{c,m}} L = \frac{\partial L}{\partial \hat{y}_c} \frac{\partial \hat{y}_c}{\partial u_c} \frac{\partial u_c}{\partial W_{c,m}}$$

depends on the loss function

depends on the activation function

z_m

binary classification

scalar output C=1

$$\begin{cases} \hat{y} = g(u) = \left(1 + e^{-u}\right)^{-1} \\ L(y, \hat{y}) = y \log \hat{y} + (1-y) \log(1-\hat{y}) \end{cases}$$

substituting and simplifying (see logistic regression lecture)

$$\begin{cases} L(y, u) = y \log 1 + e^{-u} + (1-y) \log(1 + e^{u}) \\ u = \sum_m W_m z_m \end{cases}$$

substituting u in L and taking derivative $\frac{\partial}{\partial W_m} L = (\hat{y} - y) z_m$

Gradient calculation

using the chain rule

$$\frac{\partial}{\partial W_{c,m}} L = \frac{\partial L}{\partial \hat{y}_c} \frac{\partial \hat{y}_c}{\partial u_c} \frac{\partial u_c}{\partial W_{c,m}}$$

depends on the loss function

depends on the activation function

z_m

multiclass classification

C is the number of classes

$$\begin{cases} y = g(u) = \text{softmax}(u) \\ L(y, \hat{y}) = \sum_k y_k \log \hat{y}_k \end{cases}$$

substituting and simplifying (see logistic regression lecture)

$$\begin{cases} L(y, u) = -y^\top u + \log \sum_c e^u \\ u_c = \sum_m W_{c,m} z_m \end{cases}$$

substituting u in L and taking derivative $\frac{\partial}{\partial W_{c,m}} L = (\hat{y}_c - y_c) z_m$

Gradient calculation

gradient wrt V:

we already did this part

$$\frac{\partial}{\partial V_{m,d}} L = \sum_c \frac{\partial L}{\partial \hat{y}_c} \frac{\partial \hat{y}_c}{\partial u_m} \frac{\partial u_m}{\partial z_m} \frac{\partial z_m}{\partial q_m} \frac{\partial q_m}{\partial V_{m,d}}$$

$W_{k,m}$ x_d

depends on the middle layer activation

logistic function	$\sigma(q_m)(1-\sigma(q_m))$
hyperbolic tan.	$1-\tanh(q_m)^2$
ReLU	$\begin{cases} 0 & q_m \leq 0 \\ 1 & q_m > 0 \end{cases}$

$L(y, \hat{y})$

$\hat{y}_c = g(u_c)$

$u_c = \sum_{m=1}^{M} W_{c,m} z_m$

$z_m = h(q_m)$

$q_m = \sum_{d=1}^{D} V_{m,d} x_d$

x_d

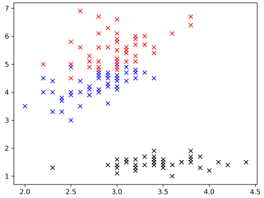
example logistic sigmoid

$$\frac{\partial}{\partial V_{m,d}} J = \sum_n \sum_k (\hat{y}_k^{(n)} - y_k^{(n)}) W_{k,m} \sigma(q_m^{(n)})(1-\sigma(q_m^{(n)})) x_d^{(n)}$$

$$= \sum_n \sum_k (\hat{y}_k^{(n)} - y_k^{(n)}) W_{k,m} z_m^{(n)} (1 - z_m^{(n)}) x_d^{(n)}$$

to get gradient for biases we simply assume the input is 1. $x_0^{(n)} = 1$

Example: classification



Iris dataset (D=2 features + 1 bias)
M = 16 hidden units
C=3 classes

$$L(y, \hat{y})$$

$$\hat{y} = \text{softmax}(u)$$

$$u_c = \sum_{m=1}^{M} W_{c,m} z_m$$

$$z_m = \sigma(q_m)$$

$$q_m = \sum_{d=1}^{D} V_{m,d} x_d$$

$$x_d$$

cost is softmax-cross-entropy

```python
def cost(X, #N x D
         Y, #N x K
         W, #M x K
         V, #D x M
        ):
    Q = np.dot(X, V) #N x M
    Z = logistic(Q) #N x M
    U = np.dot(Z, W) #N x K
    Yh = softmax(U)
    nll = - np.mean(np.sum(U*Y, 1) - logsumexp(U))
    return nll
```

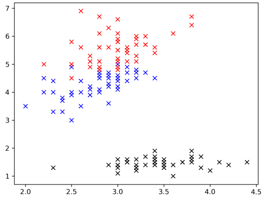
$$J = -\sum_{n=1}^{N} y^{(n)\top} u^{(n)} + \log \sum_c e^{u_c^{(n)}}$$

helper functions

```python
def logsumexp(
    Z,# NxK
):
    Zmax = np.max(Z,axis=1)[:, None]
    lse = Zmax + np.log(np.sum(np.exp(Z - Zmax), axis=1))[:, None]
    return lse #N

  def softmax(
    u, # N x K
):
    u_exp = np.exp(u - np.max(u, 1)[:, None])
    return u_exp / np.sum(u_exp, axis=-1)[:, None]
```

Example: classification



Iris dataset (D=2 features + 1 bias)
M = 16 hidden units
C=3 classes

```
def gradients(X,#N x D
              Y,#N x K
              W,#M x K
              V,#D x M
             ):
    Z = logistic(np.dot(X, V))#N x M
    N,D = X.shape
    Yh = softmax(np.dot(Z, W))#N x K
    dY = Yh - Y #N x K
    dW= np.dot(Z.T, dY)/N #M x K
    dZ = np.dot(dY, W.T) #N x M
    dV = np.dot(X.T, dZ * Z * (1 - Z))/N #D x M
    return dW, dV
```

$$\frac{\partial}{\partial W_m} L = (\hat{y} - y) z_m$$

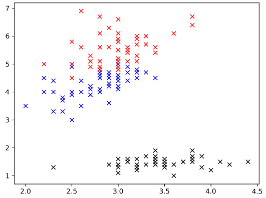
$$\frac{\partial}{\partial V_{m,d}} L = (\hat{y} - y) W_m z_m (1 - z_m) x_d$$

$$L(y, \hat{y})$$
$$\uparrow$$
$$\hat{y} = \mathrm{softmax}(u)$$
$$\uparrow$$
$$u_c = \sum_{m=1}^{M} W_{c,m} z_m$$
$$\uparrow$$
$$z_m = \sigma(q_m)$$
$$\uparrow$$
$$q_m = \sum_{d=1}^{D} V_{m,d} x_d$$
$$\uparrow$$
$$x_d$$

check your gradient function using **finite difference** approximation that uses the *cost function*

```
scipy.optimize.check_grad
```

Example: classification



Iris dataset (D=2 features + 1 bias)

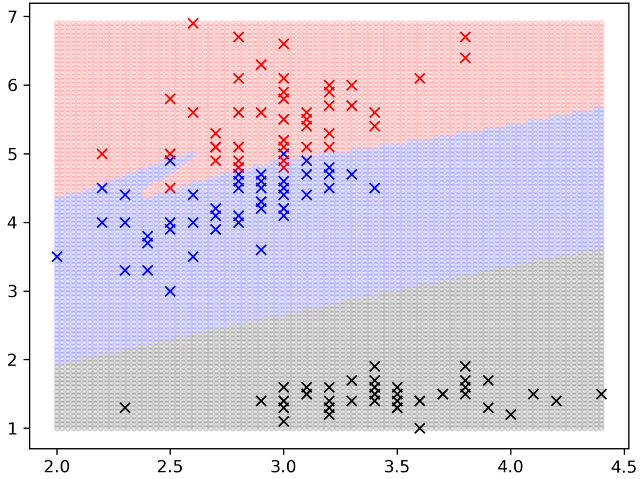
M = 16 hidden units

C=3 classes

using GD for optimization

```
def GD(X, Y, M, lr=.1, eps=1e-9, max_iters=100000):
    N, D = X.shape
    N,K = Y.shape
    W = np.random.randn(M, K)*.01
    V = np.random.randn(D, M)*.01
    dW = np.inf*np.ones_like(W)
    t = 0
    while np.linalg.norm(dW) > eps and t < max_iters:
        dW, dV = gradients(X, Y, W, V)
        W = W - lr*dW
        V = V - lr*dV
        t += 1
    return W, V
```

the resulting decision boundaries



Automating gradient computation

gradient computation is tedious and mechanical.
can we automate it?

using **numerical differentiation**?

approximates partial derivatives using finite difference $\frac{\partial f}{\partial w} \approx \frac{f(w+\epsilon)-f(w)}{\epsilon}$
needs multiple forward passes (for each input output pair)
can be slow and inaccurate
useful for black-box cost functions or checking the correctness of gradient functions

symbolic differentiation: symbolic calculation of derivatives

does not identify the computational procedure and reuse of values

automatic / algorithmic differentiation is what we want

write code that calculates various functions, *e.g., the cost function*
automatically produce (partial) derivatives *e.g., gradients used in learning*

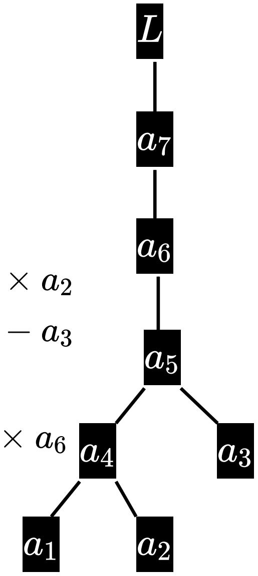
Automatic differentiation

idea use the chain rule + derivative of simple operations $*, \sin, \frac{1}{x} \dots$

use a computational graph as a data structure (for storing the result of computation)

step 1 break down to atomic operations

$$L = \frac{1}{2}(y - wx)^2 \rightarrow \begin{aligned} a_1 &= w \\ a_2 &= x \\ a_3 &= y \\ a_4 &= a_1 \times a_2 \\ a_5 &= a_4 - a_3 \\ a_6 &= a_5^2 \\ a_7 &= .5 \times a_6 \end{aligned}$$



step 2 build a graph with operations as internal nodes and input variables as leaf nodes

step 3 there are two ways to use the computational graph to calculate derivatives

forward mode: start from the leafs and propagate derivatives upward

reverse mode:

1. first in a bottom-up (forward) pass calculate the values $a_1, \dots, a_4$
2. in a top-down (backward) pass calculate the derivatives

this second procedure is called **backpropagation** when applied to neuran networks

Forward mode

suppose we want the derivative $\frac{\partial y_1}{\partial w_1}$ where $\begin{cases} y_1 = \sin(w_1 x + w_0) \\ y_2 = \cos(w_1 x + w_0) \end{cases}$

we can calculate both y_1, y_2 and derivatives $\frac{\partial y_1}{\partial w_1}$ $\frac{\partial y_2}{\partial w_1}$ in a single forward pass

	evaluation	**partial derivatives**	
	$a_1 = w_0$	$\dot{a}_1 = 0$	we initialize these to identify which derivative we want
	$a_2 = w_1$	$\dot{a}_2 = 1$	this means $\dot{\square} = \frac{\partial \square}{\partial w_1}$
	$a_3 = x$	$\dot{a}_3 = 0$	
$w_1 x$	$a_4 = a_2 \times a_3$	$\dot{a}_4 = a_2 \times \dot{a}_3 + \dot{a}_2 \times a_3$	x
$w_1 x + w_0$	$a_5 = a_4 + a_1$	$\dot{a}_5 = \dot{a}_4 + \dot{a}_1$	x
$y_1 = \sin(w_1 x + w_0)$	$a_6 = \sin(a_5)$	$\dot{a}_6 = \dot{a}_5 \cos(a_5)$	$x \cos(w_1 x + w_0) = \frac{\partial y_1}{\partial w_1}$
$y_2 = \cos(w_1 x + w_0)$	$a_7 = \cos(a_5)$	$\dot{a}_7 = -\dot{a}_5 \sin(a_5)$	$-x \sin(w_1 x + w_0) = \frac{\partial y_2}{\partial w_1}$

note that we get all partial derivatives $\frac{\partial \square}{\partial w_1}$ in one forward pass

Forward mode: computational graph

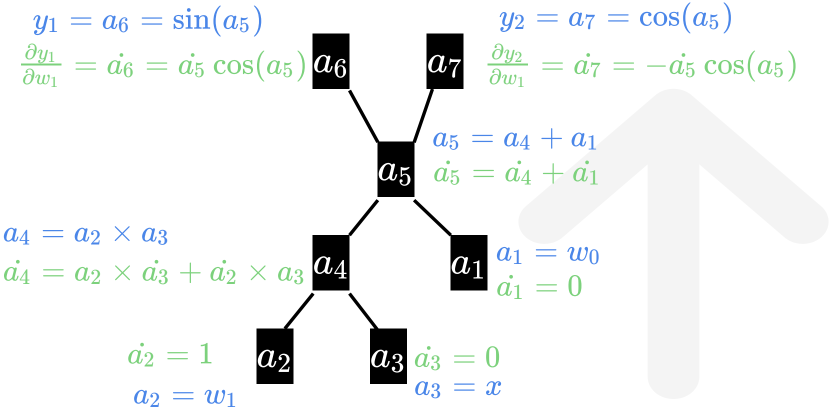
suppose we want the derivative $\frac{\partial y_1}{\partial w_1}$ where $\begin{cases} y_1 = \sin(w_1 x + w_0) \\ y_2 = \cos(w_1 x + w_0) \end{cases}$

we can represent this computation using a graph

once the nodes up stream calculate their values and derivatives we may discard a node

- *e.g.*, once $a_5, \dot{a_5}$ are obtained we can discard the values and partial derivatives for $a_4, \dot{a_4}, a_1, \dot{a_1}$

evaluation	partial derivatives
$a_1 = w_0$	$\dot{a_1} = 0$
$a_2 = w_1$	$\dot{a_2} = 1$
$a_3 = x$	$\dot{a_3} = 0$
$a_4 = a_2 \times a_3$	$\dot{a_4} = a_2 \times \dot{a_3} + \dot{a_2} \times a_3$
$a_5 = a_4 + a_1$	$\dot{a_5} = \dot{a_4} + \dot{a_1}$
$y_1 = a_6 = \sin(a_5)$	$\dot{a_6} = \dot{a_5} \cos(a_5)$
$y_2 = a_7 = \cos(a_5)$	$\dot{a_7} = -\dot{a_5} \cos(a_5)$



Reverse mode

suppose we want the derivative $\frac{\partial y_2}{\partial w_1}$ where $y_2 = \cos(w_1 x + w_0)$

first do a forward pass for evaluation

1) evaluation

$$a_1 = w_0$$
$$a_2 = w_1$$
$$a_3 = x$$
$$w_1 x \qquad a_4 = a_2 \times a_3$$
$$w_1 x + w_0 \qquad a_5 = a_4 + a_1$$
$$y_1 = \sin(w_1 x + w_0) \qquad y_1 = a_6 = \sin(a_5)$$
$$y_2 = \cos(w_1 x + w_0) \qquad y_2 = a_7 = \cos(a_5)$$

then use these values to calculate partial derivatives in a backward pass

2) partial derivatives

$$\frac{\partial y_2}{\partial y_2} = 1 \qquad \bar{a}_7 = 1$$
$$\frac{\partial y_2}{\partial y_1} = 0 \qquad \bar{a}_6 = 0$$

$\left.\right\}$ this means $\bar{\square} = \frac{\partial y_2}{\partial \square}$

$$\frac{\partial y_2}{\partial a_5} = \frac{\partial y_2}{\partial a_7}\frac{\partial a_7}{\partial a_5} + \frac{\partial y_2}{\partial a_6}\frac{\partial a_6}{\partial a_5} = -\sin(w_1 x + w_0) \qquad \bar{a}_5 = \bar{a}_6 \cos(a_5) - \bar{a}_7 \sin(a_5)$$
$$\frac{\partial y_2}{\partial a_4} = -\sin(w_1 x + w_0) \qquad \bar{a}_4 = \bar{a}_5$$
$$\frac{\partial y_2}{\partial x} = -w_1 \sin(w_1 x + w_0) \qquad \bar{a}_3 = a_2 \bar{a}_4$$
$$\frac{\partial y_2}{\partial w_1} = -x \sin(w_1 x + w_0) \qquad \bar{a}_2 = a_3 \bar{a}_4$$
$$\frac{\partial y_2}{\partial w_0} = -\sin(w_1 x + w_0) \qquad \bar{a}_1 = \bar{a}_5$$

we get all partial derivatives $\frac{\partial y_2}{\partial \square}$ in one backward pass

Reverse mode: computational graph

suppose we want the derivative $\frac{\partial y_2}{\partial w_1}$ where $y_2 = \cos(w_1 x + w_0)$

we can represent this computation using a graph

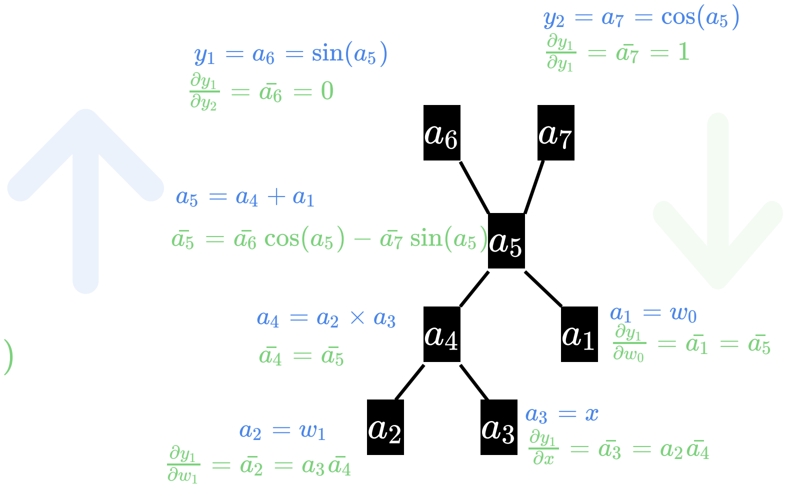
1. in a forward pass we do evaluation and **keep the values**
2. use these values in the backward pass to get partial derivatives

1) evaluation

$$a_1 = w_0$$
$$a_2 = w_1$$
$$a_3 = x$$
$$a_4 = a_2 \times a_3$$
$$a_5 = a_4 + a_1$$
$$y_1 = a_6 = \sin(a_5)$$
$$y_2 = a_7 = \cos(a_5)$$

2) partial derivatives

$$\bar{a}_7 = 1$$
$$\bar{a}_6 = 0$$
$$\bar{a}_5 = \bar{a}_6 \cos(a_5) - \bar{a}_7 \sin(a_5)$$
$$\bar{a}_4 = \bar{a}_5$$
$$\bar{a}_3 = a_2 \bar{a}_4$$
$$\bar{a}_2 = a_3 \bar{a}_4$$
$$\bar{a}_1 = \bar{a}_5$$



Forward vs Reverse mode

forward mode is more natural, easier to implement and requires less memory

a single forward pass calculates $\frac{\partial y_1}{\partial w}, \ldots, \frac{\partial y_c}{\partial w}$

however, reverse mode is more efficient in calculating gradient $\nabla_w y = [\frac{\partial y}{\partial w_1}, \ldots, \frac{\partial y}{\partial w_D}]^\top$

this is more efficient if we have single output (cost) and many variables (weights)

for this reason, in training neural networks, reverse mode is used

the backward pass in the reverse mode is called **backpropagation**

many machine learning software implement autodiff:

- `autograd (extends numpy)`
- `pytorch`
- `tensorflow`

Summary

optimization landscape in neural networks is special and not yet fully understood

- exponentially many local optima and saddle points
- most local minima are good
- calculate the gradients using backpropagation

automatic differentiation

- simplifies gradient calculation for complex models
- gradient descent becomes simpler to use
- forward mode is useful for calculating the jacobian of $f : \mathbb{R}^Q \to \mathbb{R}^P$ when $P \geq Q$
- reverse mode can be more efficient when $Q > P$
  - backpropagation is reverse mode autodiff.