Applied Machine Learning

Gradient Computation & Automatic Differentiation

Isabeau Prémont-Schwarz

McGill School of Computer Science

COMP 551 (Fall 2023)

Learning objectives

using the chain rule to calculate the gradients

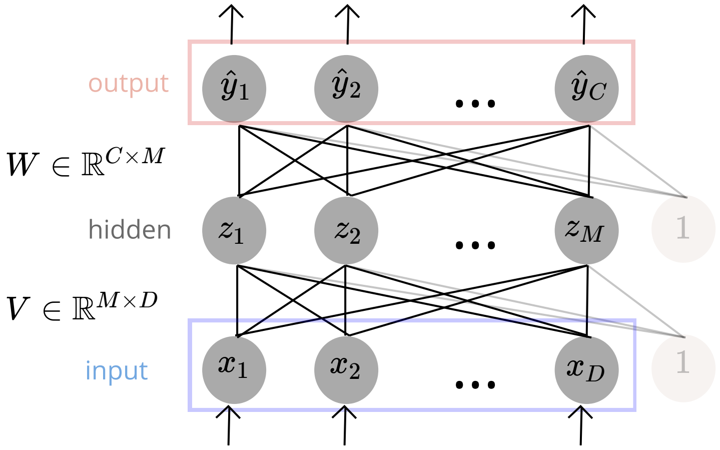
automatic differentiation

- forward mode
- reverse mode (backpropagation)

Landscape of the cost function

model two layer MLP

$$f(x; W, V) = g\big(W h(V x)\big)$$

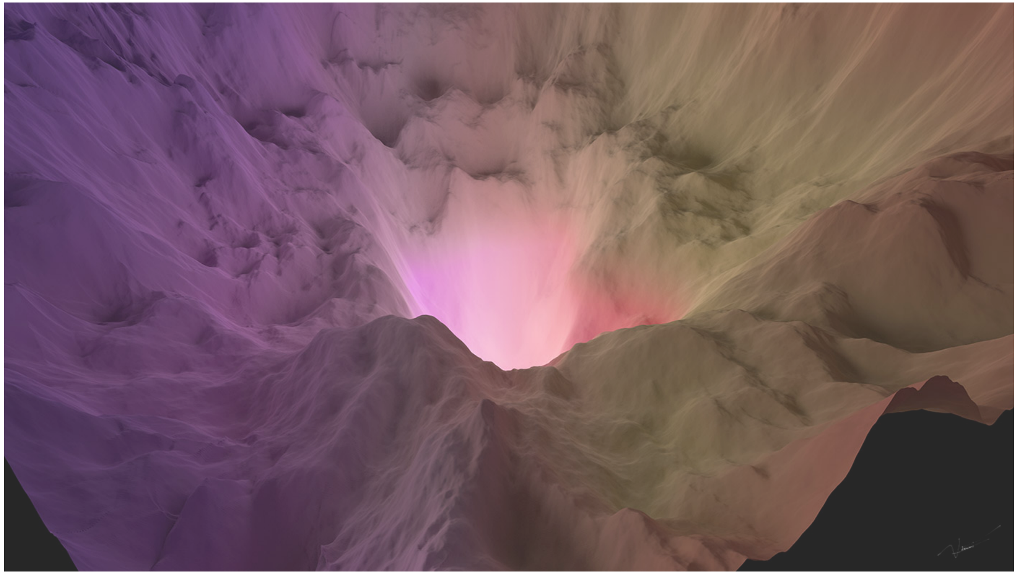


for simplicity we drop the bias terms

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



https://losslandscape.com/gallery/

Landscape of the cost function

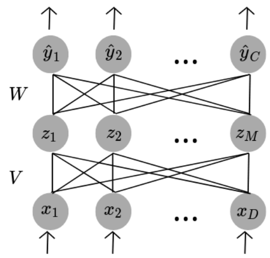
model two layer MLP

$$f(x; W, V) = g\big(W h(V x)\big)$$

there are **exponentially many** optima

given one optimum V*, W* we can create many more with the same cost:

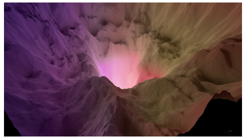
- permute hidden units in each layer (M!) weight space symmetry
- for symmetric activations: negate input/ouput of a unit
- for ReLU: rescale input/output weights attached to a unit



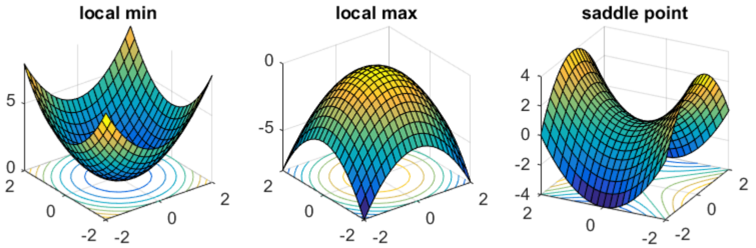
objective $\min_{W,V} \sum_n L(y^{(n)}, f(x^{(n)}; W, V))$

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

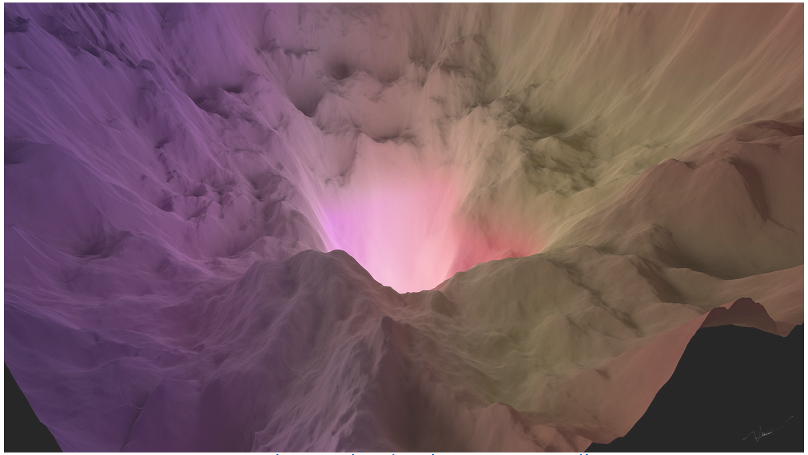
Landscape of the cost function

there are **exponentially many** optima

given one optimum V*, W* we can create many more with the same cost:

- permute hidden units in each layer (M!)
- for symmetric activations: negate input/ouput of a unit
- for ReLU: rescale input/output weights attached to a unit

this is a non-convex optimization problem



https://losslandscape.com/gallery/

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)

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

$\frac{\partial}{\partial w_1} f(w)$

$\nabla_w f_1(w)$

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

note that we use J also for cost function

$$J_{ij} = \frac{\partial f_i(w)}{\partial w_j}$$

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 into a vector for these calculations

Examples

$f(x) = x^2 \Rightarrow \frac{d}{dw} f(w) = 2x$

$f(x, y) = x^2 + y^2 \Rightarrow$
$\nabla_{w=[x,y]} f(w) = [2x, 2y]$

$f(x, y) = [x^2, y^2, 2xy] \Rightarrow$

$$J = \begin{bmatrix} 2x, & 0 \\ 0, & 2y \\ 2y, & 2x \end{bmatrix}$$

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

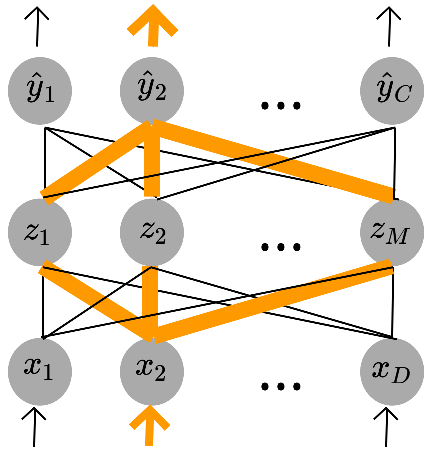
more generally $x \in \mathbb{R}^D, z \in \mathbb{R}^M, y \in \mathbb{R}^C$

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

C x D Jacobian

M x D Jacobian

C x M Jacobian



$$\frac{\partial y_c}{\partial x_d} = \sum_{m=1}^{M} \frac{\partial y_c}{\partial z_m}\frac{\partial z_m}{\partial x_d}$$

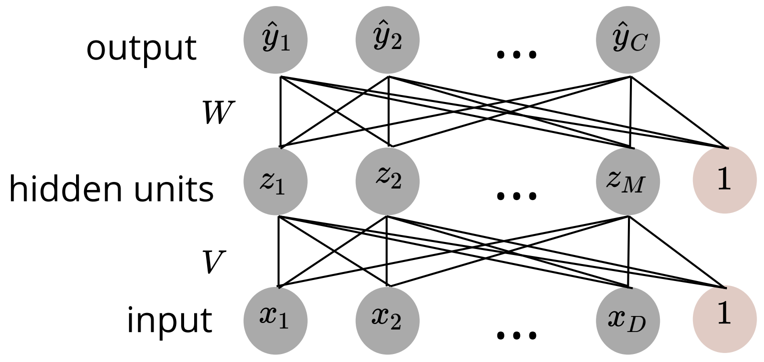
we are looking at all the "paths" through which change in x_d changes y_c and add their contribution

Training a two layer network

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

Cost function we want to minimize

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



for simplicity we drop the bias terms

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

$$\frac{\partial}{\partial W} J = \sum_{n=1}^{N} \frac{\partial}{\partial W} L(y^{(n)}, \hat{y}^{(n)}) \quad \text{and} \quad \frac{\partial}{\partial V} J = \sum_{n=1}^{N} \frac{\partial}{\partial V} L(y^{(n)}, \hat{y}^{(n)})$$

😃 ☺

Gradient calculation

using the chain rule

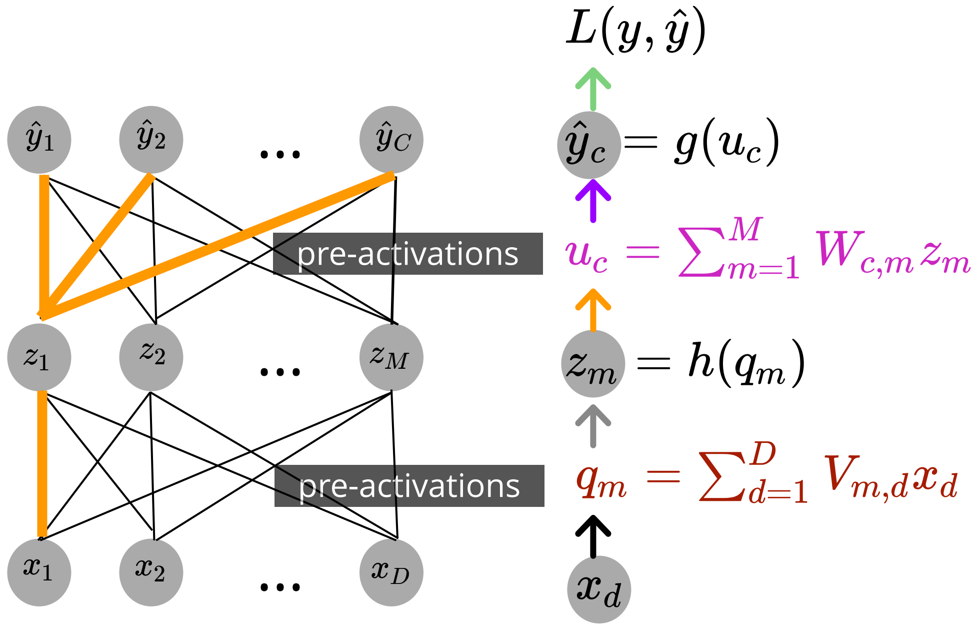
☺ $$\frac{\partial}{\partial W_{c,m}} L = \frac{\partial L}{\partial \hat{y}_c} \frac{\partial \hat{y}_c}{\partial u_c} \frac{\partial u_c}{\partial W_{c,m}}$$

- $\frac{\partial L}{\partial \hat{y}_c}$: depends on the loss function
- $\frac{\partial \hat{y}_c}{\partial u_c}$: depends on the activation function
- $\frac{\partial u_c}{\partial W_{c,m}}$: z_m

similarly for V

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

- $\frac{\partial L}{\partial \hat{y}_c}$: depends on the loss function
- $\frac{\partial \hat{y}_c}{\partial u_c}$: depends on the activation function
- $\frac{\partial u_c}{\partial z_m}$: $W_{c,m}$
- $\frac{\partial z_m}{\partial q_m}$: depends on the middle layer activation
- $\frac{\partial q_m}{\partial V_{m,d}}$: 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

C=1

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

$$\hat{y} = g(u) = u$$

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

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

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

$z_m = h(q_m)$

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

x_d

combining the three terms above

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

we have seen this in linear regression lecture!

more generally:

$$\frac{\partial}{\partial W_{c,m}} L = (\hat{y}_c - y_c) z_m$$

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

binary classification

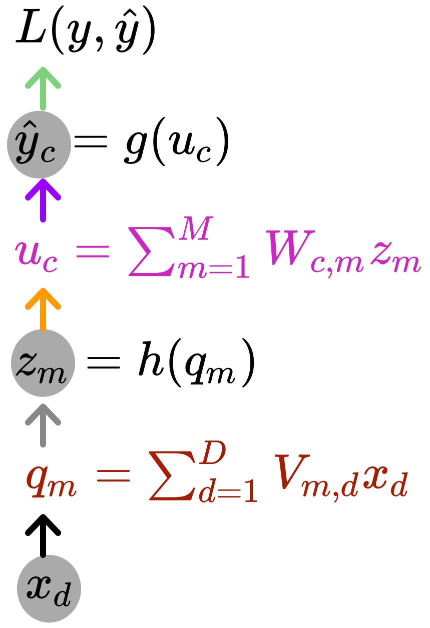
scalar output C=1

$$\hat{y} = g(u) = (1 + e^{-u})^{-1}$$

$$L(y, \hat{y}) = -y \log \hat{y} - (1-y) \log(1-\hat{y})$$

$$\frac{\partial \hat{y}}{\partial u} = \hat{y}(1-\hat{y})$$

$$\frac{\partial}{\partial \hat{y}} L(y, \hat{y}) = -\frac{y}{\hat{y}} + \frac{(1-y)}{(1-\hat{y})}$$



combining the three terms above

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

looks familiar?

we had seen this in the logistic regression lecture

Gradient calculation

$$L(y, \hat{y})$$
$$\hat{y}_c = g(u_c)$$
$$u_c = \sum_{m=1}^{M} W_{c,m} z_m$$
$$z_m = h(q_m)$$
$$q_m = \sum_{d=1}^{D} V_{m,d} x_d$$
$$x_d$$

using the chain rule

$$\frac{\partial}{\partial W_{c,m}} L = \sum_{k=1}^{C} \frac{\partial L}{\partial \hat{y}_k} \frac{\partial \hat{y}_k}{\partial u_c} \frac{\partial u_c}{\partial W_{c,m}}$$

depends on the loss function

depends on the activation function

z_m

multiclass classification

C is the number of classes

$$L(y, \hat{y}) = -\sum_c y_c \log \hat{y}_c$$

$$\frac{\partial}{\partial \hat{y}_k} L = -\frac{y_k}{\hat{y}_k}$$

$$\hat{y} = g(u) = \mathrm{softmax}(u)$$

softmax takes a vector and produces a vector

$$\hat{y}_k = \frac{e^{u_k}}{\sum_i e^{u_i}}$$

need to calculate the Jacobian

$$\frac{\partial}{\partial u_c} \hat{y}_k = \begin{cases} \hat{y}_k (1 - \hat{y}_k) & k = c \\ -\hat{y}_c \hat{y}_k & k \neq c \end{cases}$$

combining the three terms above

$$\frac{\partial}{\partial W_{c,m}} L = (\hat{y}_c - y_c) z_m$$

again, this is familiar (softmax regression lecture)

Gradient calculation

gradient wrt V:

we already did this part

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

$\frac{\partial u_c}{\partial z_m} = W_{c,m}$ $\qquad \frac{\partial q_m}{\partial V_{m,d}} = x_d$

$\frac{\partial z_m}{\partial q_m}$: 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

$$\frac{\partial}{\partial V_{m,d}} L = \sum_c (\hat{y}_c - y_c) W_{c,m} \sigma(q_m)(1-\sigma(q_m)) x_d$$
$$= \sum_c (\hat{y}_c - y_c) W_{c,m} z_m (1 - z_m) x_d$$

$$\Rightarrow \frac{\partial}{\partial V_{m,d}} J = \sum_n \sum_c (\hat{y}_c^{(n)} - y_c^{(n)}) W_{c,m} z_m^{(n)} (1 - z_m^{(n)}) x_d^{(n)}$$

for biases we simply assume the input is 1. $x_0^{(n)} = 1$ $\qquad \frac{\partial}{\partial b_m^1} L = \sum_c (\hat{y}_c - y_c) W_{c,m} \sigma(q_m)(1-\sigma(q_m))$

Gradient calculation

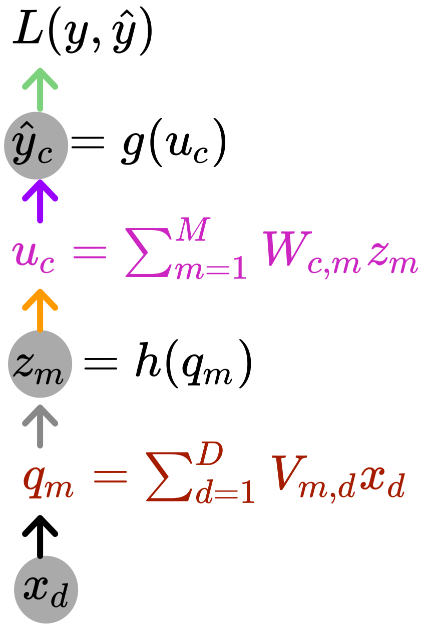
a common pattern

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

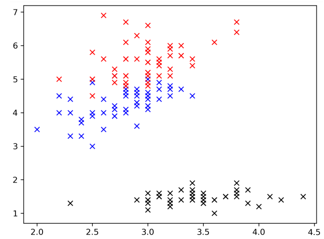
error from above $\frac{\partial L}{\partial u_c}$ input from below z_m

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

error from above $\frac{\partial L}{\partial q_m}$ input from below x_d



Example: classification



Iris dataset (D=2 features + 1 bias)

M = 16 hidden units

C=3 classes

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

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

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

$$z_m = \sigma(q_m)$$

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

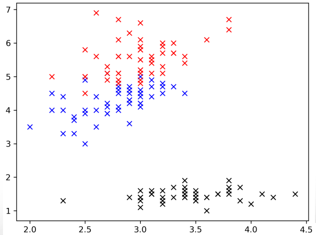
$$x_d$$

cost is softmax-cross-entropy

```
def cost(x, #N x D
         y, #N x C
         w, #M x C
         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 C
  yh = softmax(u)
  nll = - np.mean(np.sum(u*y, 1) - logsumexp(u))
  return nll
```

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

Example: classification



Iris dataset (D=2 features + 1 bias)

M = 16 hidden units

C=3 classes

```
def gradient(x,#N x D
             y,#N x C
             w,#M x C
             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 C
    dy = yh - y #N x C
    dw= np.dot(z.T, dy)/N #M x C
    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
```

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

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

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

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

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

$$z_m = \sigma(q_m)$$

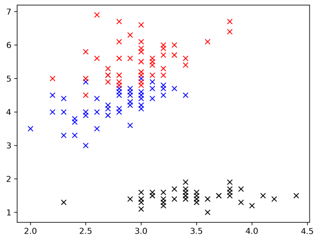
$$q_m = \sum_{d=1}^{D} V_{m,d} x_d$$

$$x_d$$

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)

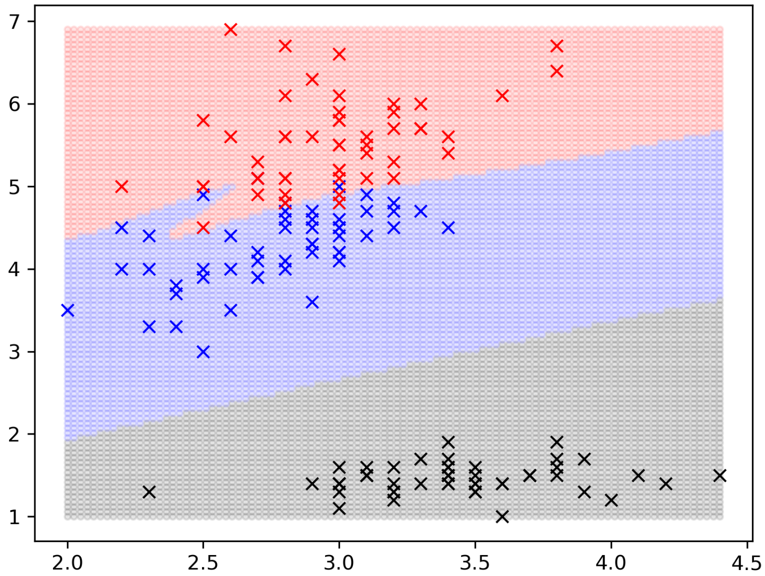
M = 16 hidden units

C=3 classes

using GD for optimization

```
while Condition:
    dw, dv = gradient(x, y, w, v)
    w = w - lr*dw
    v = v - lr*dv
```

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 $*, \sin, \frac{1}{x} \ldots$

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

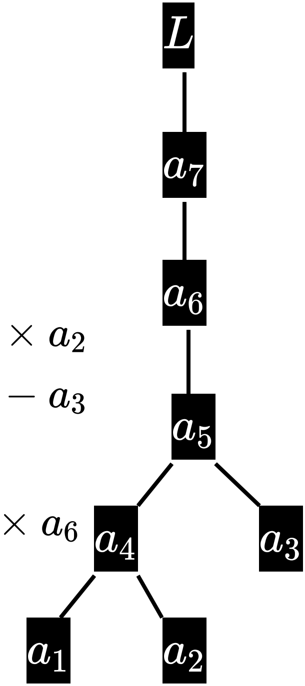
step 1 break down to atomic operations

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

$$L = \frac{1}{2}(wx - y)^2 \rightarrow$$

$$\begin{aligned} a_1 &= w \\ a_2 &= x \\ a_3 &= y \\ a_4 &= a_1 \times a_2 \\ a_5 &= a_4 - a_3 \\ a_6 &= a_5^2 \\ a_7 &= .5 \times a_6 \end{aligned}$$



forward mode: start from the leafs and propagate derivatives upward

reverse mode:

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

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

optional

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

optional

Forward mode: computational graph

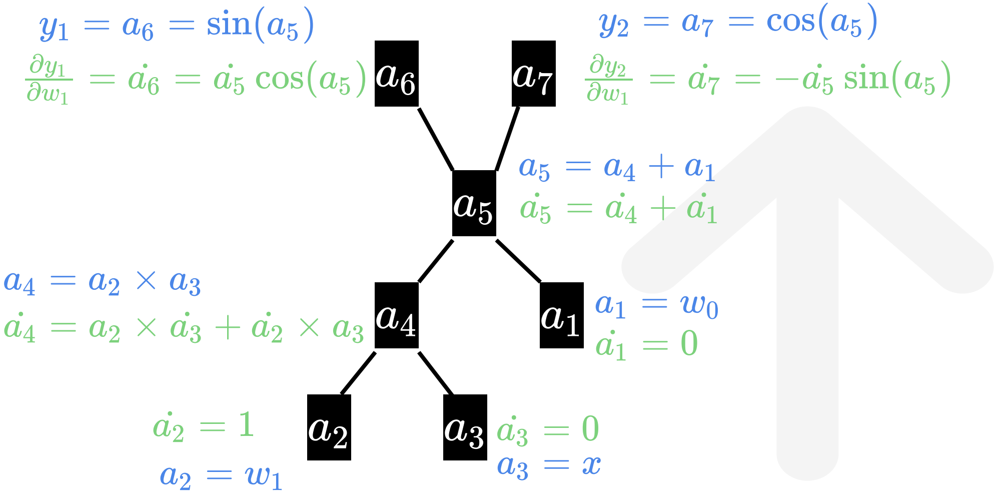
suppose we want the derivative $\frac{\partial y_1}{\partial w_1}$ where $\begin{cases} y_1 = \sin(w_1 x + w_0) \\ y_2 = \cos(w_1 x + w_0) \end{cases}$

we can represent this computation using a graph

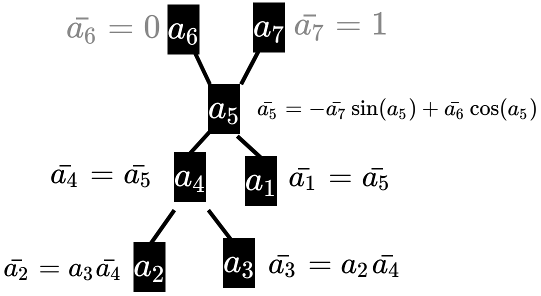
once the nodes up stream calculate their values and derivatives we may discard a node

- *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
$a_1 = w_0$	$\dot{a}_1 = 0$
$a_2 = w_1$	$\dot{a}_2 = 1$
$a_3 = x$	$\dot{a}_3 = 0$
$a_4 = a_2 \times a_3$	$\dot{a}_4 = a_2 \times \dot{a}_3 + \dot{a}_2 \times a_3$
$a_5 = a_4 + a_1$	$\dot{a}_5 = \dot{a}_4 + \dot{a}_1$
$y_1 = a_6 = \sin(a_5)$	$\dot{a}_6 = \dot{a}_5 \cos(a_5)$
$y_2 = a_7 = \cos(a_5)$	$\dot{a}_7 = -\dot{a}_5 \sin(a_5)$



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

$$\frac{\partial y_2}{\partial a_7} = 1 \qquad \bar{a}_7 = 1$$
$$\frac{\partial y_2}{\partial a_6} = 0 \qquad \bar{a}_6 = 0$$

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

$$\frac{\partial y_2}{\partial a_5} = \frac{\partial y_2}{\partial a_7}\frac{\partial a_7}{\partial a_5} + \frac{\partial y_2}{\partial a_6}\frac{\partial a_6}{\partial a_5} = -\sin(w_1 x + w_0) \qquad \bar{a}_5 = -\bar{a}_7 \sin(a_5) + \bar{a}_6 \cos(a_5)$$
$$\frac{\partial y_2}{\partial a_4} = \frac{\partial y_2}{\partial a_5}\frac{\partial a_5}{\partial a_4} = -\sin(w_1 x + w_0) \qquad \bar{a}_4 = \bar{a}_5$$
$$\frac{\partial y_2}{\partial x} = \frac{\partial y_2}{\partial a_3} = \frac{\partial y_2}{\partial a_4}\frac{\partial a_4}{\partial a_3} = -w_1 \sin(w_1 x + w_0) \qquad \bar{a}_3 = \bar{a}_4 a_2$$
$$\frac{\partial y_2}{\partial w_1} = \frac{\partial y_2}{\partial a_2} = \frac{\partial y_2}{\partial a_4}\frac{\partial a_4}{\partial a_2} = -x \sin(w_1 x + w_0) \qquad \bar{a}_2 = \bar{a}_4 a_3$$
$$\frac{\partial y_2}{\partial w_0} = \frac{\partial y_2}{\partial a_1} = \frac{\partial y_2}{\partial a_5}\frac{\partial a_5}{\partial a_1} = -\sin(w_1 x + w_0) \qquad \bar{a}_1 = \bar{a}_5$$

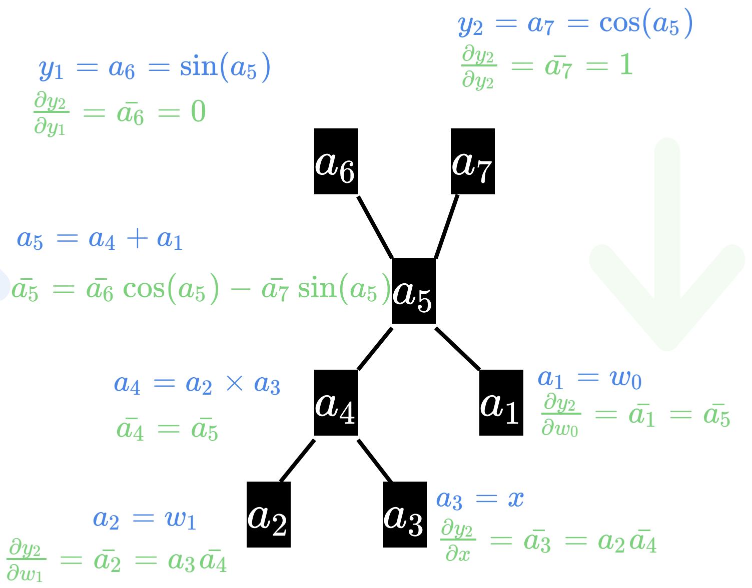
we get all partial derivatives $\frac{\partial y_2}{\partial \square}$ in one backward pass

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

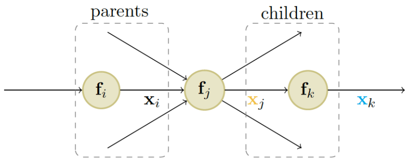


1) evaluation

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

2) partial derivatives

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



$$\frac{\partial o}{\partial x_j} = \sum_{k \in chi(j)} \frac{\partial o}{\partial x_k} \frac{\partial x_k}{\partial x_j}$$

$$\bar{a}_j = \sum_{k \in chi(j)} \bar{a}_k \frac{\partial x_k}{\partial x_j}$$

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`

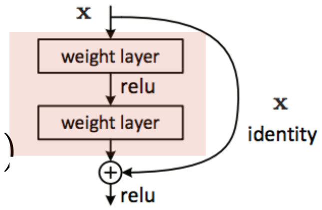
Improving optimization in deep learning

Initialization of parameters:

- random initialization (uniform or Gaussian) with small variance
  - break the symmetry of hidden units
- small positive values for bias (so that input to ReLU is >0)

models that are simpler to optimize:

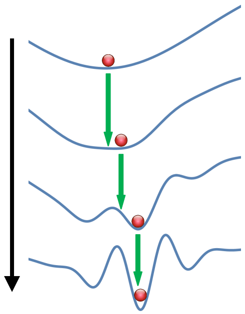
this block is correcting for the residual error in the predictions of the previous layers



- using ReLU activation
- using **skip-connection** $x^{\{\ell+l\}} = \mathrm{ReLU}(W^{\{\ell+l\}}\mathrm{ReLU}(\ldots\mathrm{ReLU}(W^{\{\ell\}}x^{\{\ell\}})\ldots) + x^{\{\ell\}})$
- using **batch-normalization** (next)

Pretrain a (simpler) model on a (simpler) task and

fine-tune on a more difficult target setting (has many forms)



continuation methods in optimization

- gradually increase the difficulty of the optimization problem
- good initialization for the next iteration

curriculum learning (similar idea)

- increase the number of "difficult" examples over time
- similar to the way humans learn

image credit: Mobahi'16

optional

Batch Normalization

original motivation

- gradient descent: parameters in all layers are updated
- distribution of inputs to layer ℓ changes
- each layer has to re-adjust
- inefficient for very deep networks

idea normalize the input to each unit (m) of a layer ℓ

activation for the instance (n) at layer ℓ

$$\hat{x}_m^{\{\ell\},(n)} = \frac{x_m^{\{\ell\},(n)} - \mu_m^{\{\ell\}}}{\sigma_m^{\{\ell\}}}$$

unit m

alternatively: apply the batch-norm to $W^{\{\ell\}} x^{\{\ell\}}$

each unit is unnecessarily constrained to have zero-mean and std=1 (we only need to fix the distribution)

introduce learnable parameters $\mathrm{ReLU}(\gamma^{\{\ell\}} \mathrm{BN}(W^{\{\ell\}} x^{\{\ell\}}) + \beta^{\{\ell\}})$

- mean and std per unit is calculated for the minibatch during the forward pass
- we backpropagate through this normalization
- at test time use the mean and std. from the whole training set
- BN regularizes the model

recent observations the change in distribution of activations is not a big issue empirically

BN works so well because it makes the loss function smooth

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.

Better optimization in deep learning:

- better initialization
- models that are easier to optimize (using skip-connection, batch-norm, ReLU)
- pre-training and curriculum learning