Neural Networks

Samuel Cheng
Slide credits: Andrej Karpathy, Justin Johnson, Feifei Li

School of ECE
University of Oklahoma

Spring, 2023

Table of Contents

Review

In the last couple classes, we discussed

- Basic concepts of regression and classification
- Examples of regularization such as ridge (l_2) regression and lasso (l_1)
- Linear classifiers including logistic regression and softmax classifier

Review

In the last couple classes, we discussed

- Basic concepts of regression and classification
- Examples of regularization such as ridge (l_2) regression and lasso (l_1)
- Linear classifiers including logistic regression and softmax classifier
  - We introduced loss functions and the concept of training a classifier through minimizing the loss function

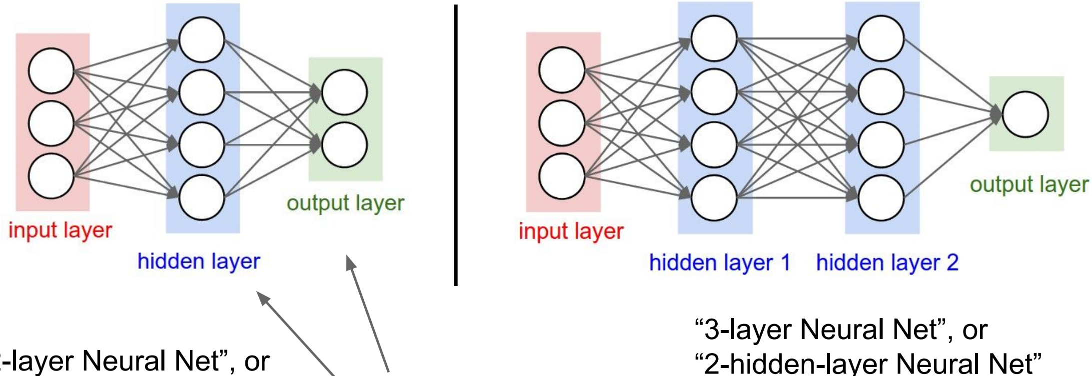
Review

In the last couple classes, we discussed

- Basic concepts of regression and classification
- Examples of regularization such as ridge (l_2) regression and lasso (l_1)
- Linear classifiers including logistic regression and softmax classifier
  - We introduced loss functions and the concept of training a classifier through minimizing the loss function
  - We described stochastic gradient descent and momentum trick for classification

Nomenclature of basic network architectures

Neural Networks: Architectures

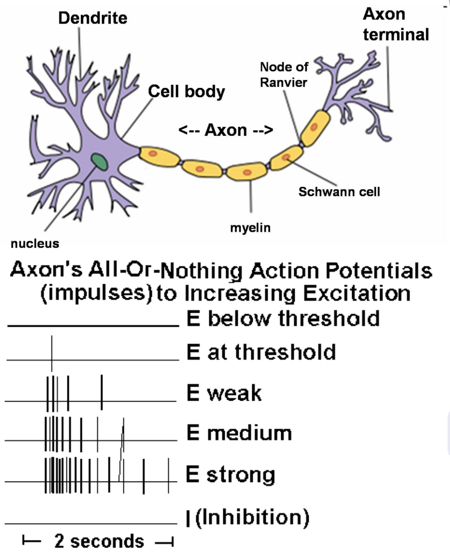


"2-layer Neural Net", or
"1-hidden-layer Neural Net"

"3-layer Neural Net", or
"2-hidden-layer Neural Net"

"Fully-connected" layers

Caveat: don't go too far for the brain analogy



Biological neurons:

- Many different types
- Dendrite can perform complex non-linear operations
- Synapses are not a single weight but a complex non-linear dynamical system
- Rate code model may not be adequate

Also see London 2005 (Slide credit: CS231n)

Back-propagation and computational graph

- As described in last lecture, training in supervised learning system often boils down to minimizing of loss function w.r.t. some parameters

Back-propagation and computational graph

- As described in last lecture, training in supervised learning system often boils down to minimizing of loss function w.r.t. some parameters
- For neural networks, it is thus necessary to find $\frac{\partial L(\mathbf{w};\mathbf{x})}{\partial w}$ for a weight in each layer

Back-propagation and computational graph

- As described in last lecture, training in supervised learning system often boils down to minimizing of loss function w.r.t. some parameters
- For neural networks, it is thus necessary to find $\frac{\partial L(\mathbf{w};\mathbf{x})}{\partial w}$ for a weight in each layer
- Back-propagation (BP) is an efficient way to find such derivation. Actually it is in fact just another way of spelling out the chain rule $\frac{\partial L}{\partial x} = \frac{\partial L}{\partial y}\frac{\partial y}{\partial x}$ in calculus

Back-propagation and computational graph

- As described in last lecture, training in supervised learning system often boils down to minimizing of loss function w.r.t. some parameters
- For neural networks, it is thus necessary to find $\frac{\partial L(\mathbf{w};\mathbf{x})}{\partial w}$ for a weight in each layer
- Back-propagation (BP) is an efficient way to find such derivation. Actually it is in fact just another way of spelling out the chain rule $\frac{\partial L}{\partial x} = \frac{\partial L}{\partial y}\frac{\partial y}{\partial x}$ in calculus
- It is often easier to explain BP in terms of computational graph

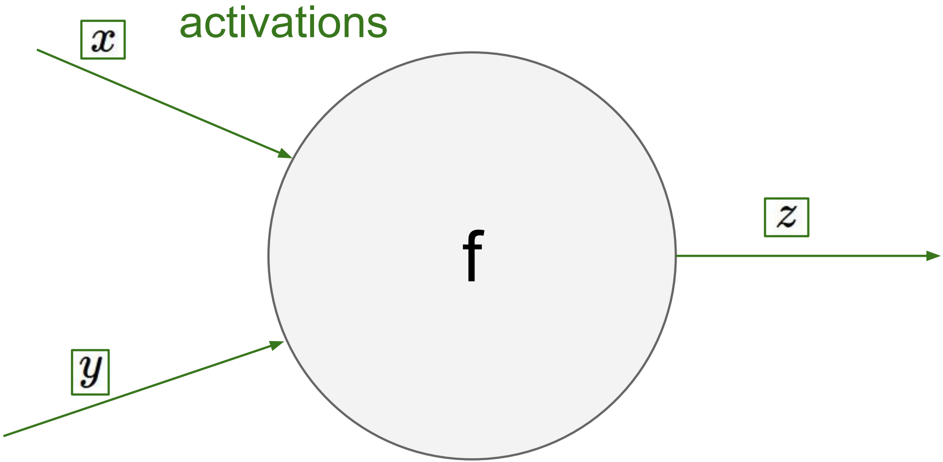
Back-propagation and computational graph

- As described in last lecture, training in supervised learning system often boils down to minimizing of loss function w.r.t. some parameters
- For neural networks, it is thus necessary to find $\frac{\partial L(\mathbf{w};\mathbf{x})}{\partial w}$ for a weight in each layer
- Back-propagation (BP) is an efficient way to find such derivation. Actually it is in fact just another way of spelling out the chain rule $\frac{\partial L}{\partial x} = \frac{\partial L}{\partial y}\frac{\partial y}{\partial x}$ in calculus
- It is often easier to explain BP in terms of computational graph
  - Computational graph can be interpreted as generalization of a neural networks
  - Neuron no longer will be restricted to summation and activation function but can be any computation as well (e.g., max)

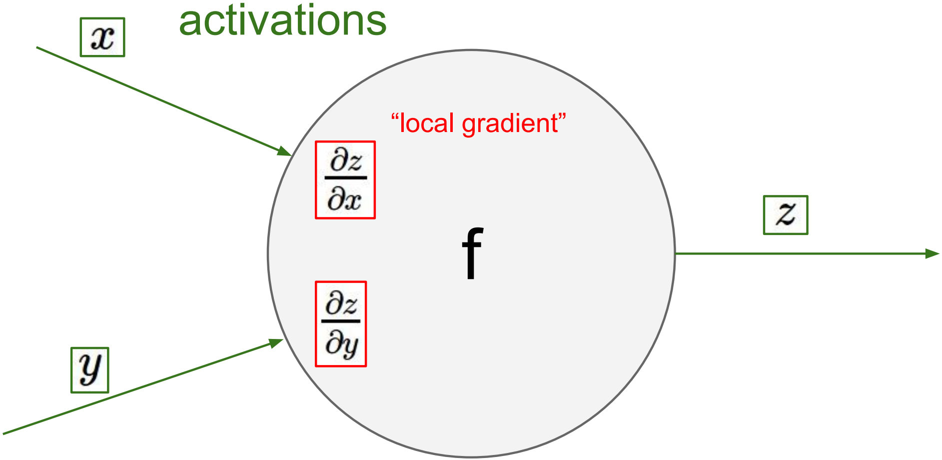
Back-propagation and computational graph

- As described in last lecture, training in supervised learning system often boils down to minimizing of loss function w.r.t. some parameters
- For neural networks, it is thus necessary to find $\frac{\partial L(\mathbf{w};\mathbf{x})}{\partial w}$ for a weight in each layer
- Back-propagation (BP) is an efficient way to find such derivation. Actually it is in fact just another way of spelling out the chain rule $\frac{\partial L}{\partial x} = \frac{\partial L}{\partial y}\frac{\partial y}{\partial x}$ in calculus
- It is often easier to explain BP in terms of computational graph
  - Computational graph can be interpreted as generalization of a neural networks
  - Neuron no longer will be restricted to summation and activation function but can be any computation as well (e.g., max)
- Let me try to explain through an example

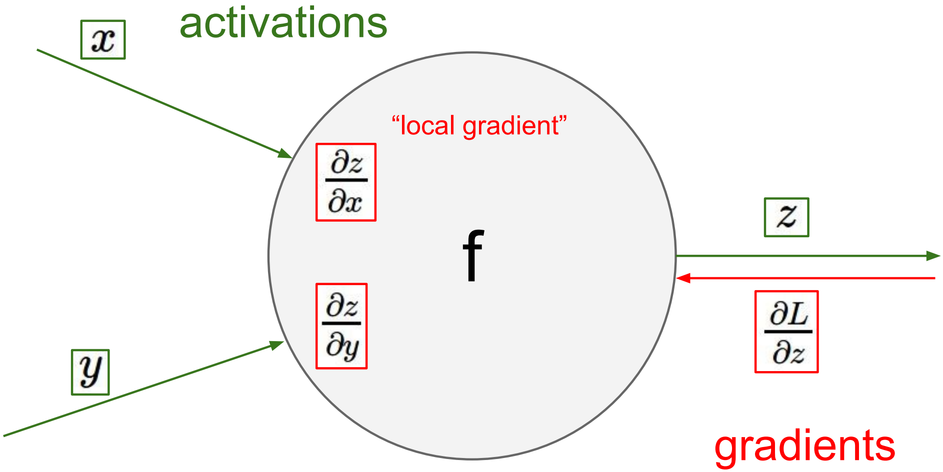
BP at one node



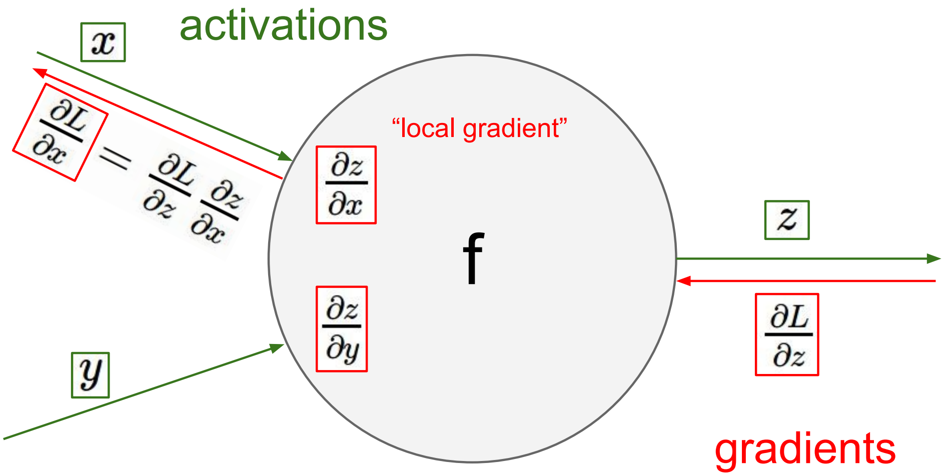
BP at one node



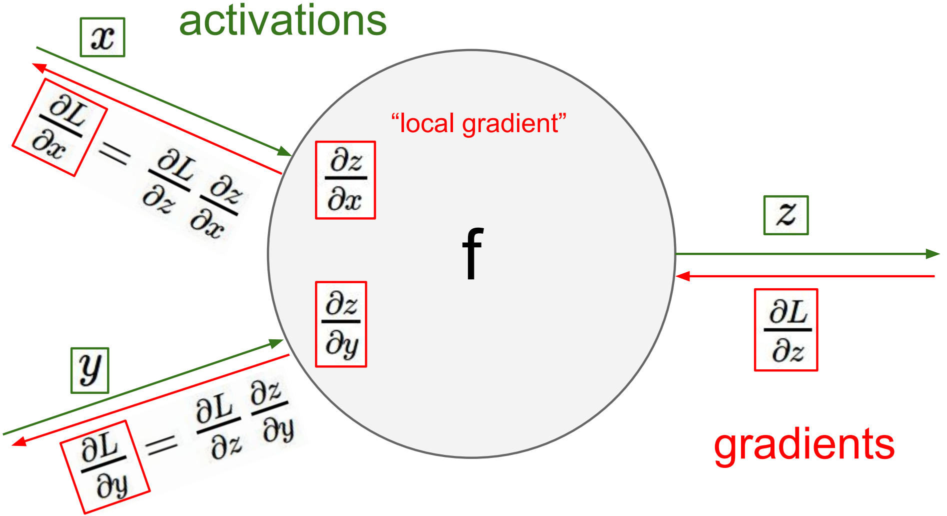
BP at one node



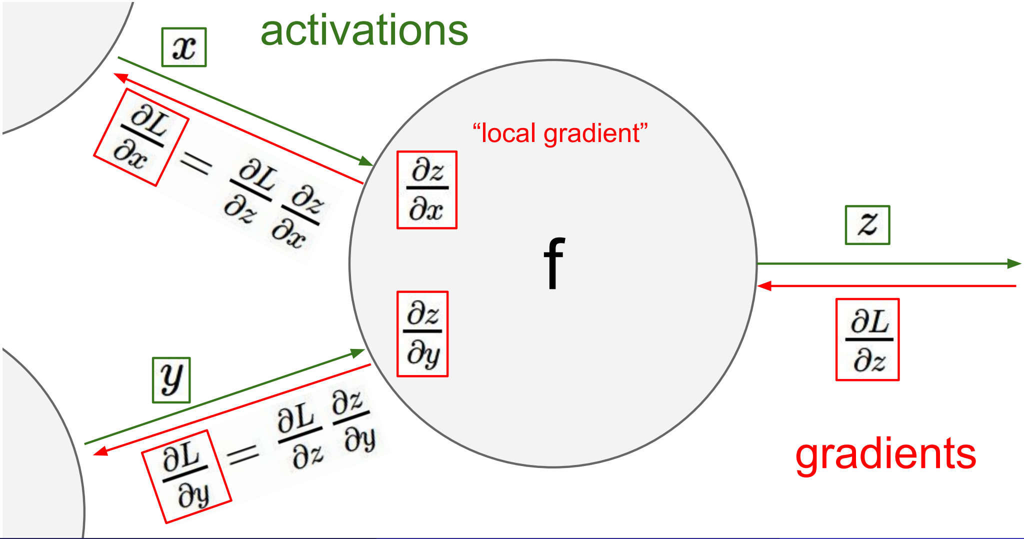
BP at one node



BP at one node



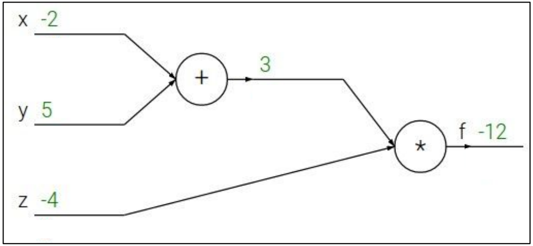
BP at one node



A simple BP example

$$f(x, y, z) = (x + y)z$$

e.g. x = -2, y = 5, z = -4



A simple BP example

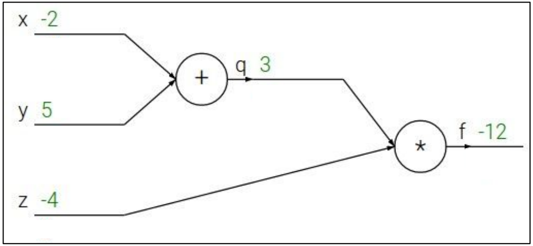
$$f(x, y, z) = (x + y)z$$

e.g. x = -2, y = 5, z = -4

$$q = x + y \qquad \frac{\partial q}{\partial x} = 1, \frac{\partial q}{\partial y} = 1$$

$$f = qz \qquad \frac{\partial f}{\partial q} = z, \frac{\partial f}{\partial z} = q$$

Want: $\frac{\partial f}{\partial x}, \frac{\partial f}{\partial y}, \frac{\partial f}{\partial z}$



A simple BP example

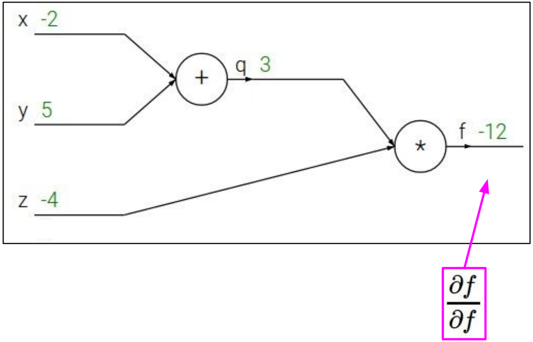
$$f(x, y, z) = (x + y)z$$

e.g. x = -2, y = 5, z = -4

$$q = x + y \qquad \frac{\partial q}{\partial x} = 1, \frac{\partial q}{\partial y} = 1$$

$$f = qz \qquad \frac{\partial f}{\partial q} = z, \frac{\partial f}{\partial z} = q$$

Want: $\frac{\partial f}{\partial x}, \frac{\partial f}{\partial y}, \frac{\partial f}{\partial z}$



A simple BP example

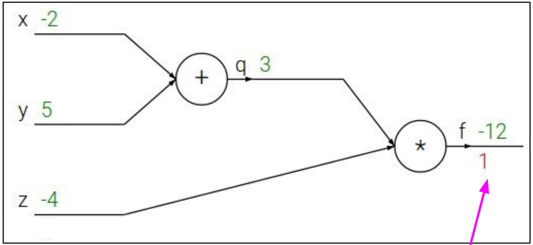
$$f(x, y, z) = (x + y)z$$

e.g. x = -2, y = 5, z = -4

$$q = x + y \qquad \frac{\partial q}{\partial x} = 1, \frac{\partial q}{\partial y} = 1$$

$$f = qz \qquad \frac{\partial f}{\partial q} = z, \frac{\partial f}{\partial z} = q$$

Want: $\frac{\partial f}{\partial x}, \frac{\partial f}{\partial y}, \frac{\partial f}{\partial z}$



$\frac{\partial f}{\partial f}$

A simple BP example

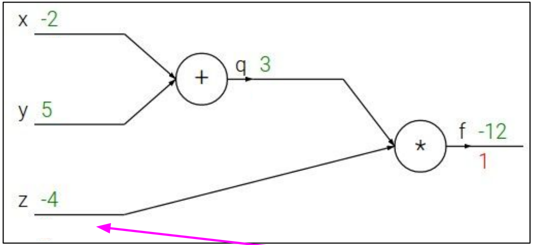
$$f(x, y, z) = (x + y)z$$

e.g. x = -2, y = 5, z = -4

$$q = x + y \qquad \frac{\partial q}{\partial x} = 1, \frac{\partial q}{\partial y} = 1$$

$$f = qz \qquad \frac{\partial f}{\partial q} = z, \frac{\partial f}{\partial z} = q$$

Want: $\frac{\partial f}{\partial x}, \frac{\partial f}{\partial y}, \frac{\partial f}{\partial z}$



$\frac{\partial f}{\partial z}$

A simple BP example

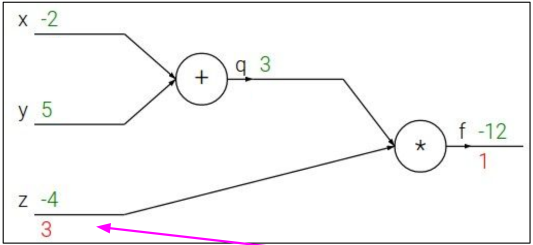
$$f(x,y,z) = (x+y)z$$

e.g. x = -2, y = 5, z = -4

$$q = x + y \qquad \frac{\partial q}{\partial x} = 1, \frac{\partial q}{\partial y} = 1$$

$$f = qz \qquad \frac{\partial f}{\partial q} = z, \frac{\partial f}{\partial z} = q$$

Want: $\frac{\partial f}{\partial x}, \frac{\partial f}{\partial y}, \frac{\partial f}{\partial z}$



$$\frac{\partial f}{\partial z}$$

A simple BP example

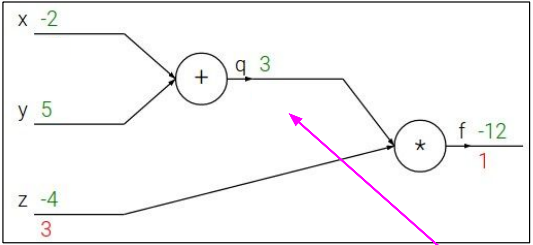
$$f(x,y,z) = (x+y)z$$

e.g. x = -2, y = 5, z = -4

$$q = x + y \qquad \frac{\partial q}{\partial x} = 1, \frac{\partial q}{\partial y} = 1$$

$$f = qz \qquad \frac{\partial f}{\partial q} = z, \frac{\partial f}{\partial z} = q$$

Want: $\frac{\partial f}{\partial x}, \frac{\partial f}{\partial y}, \frac{\partial f}{\partial z}$



$$\frac{\partial f}{\partial q}$$

A simple BP example

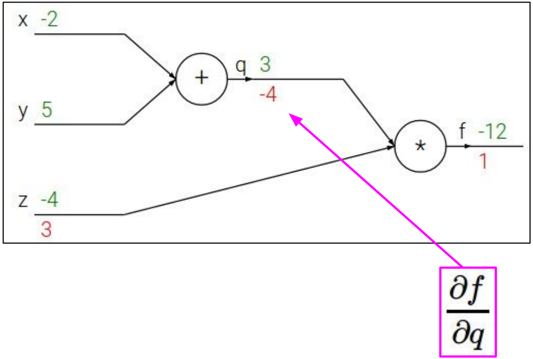
$$f(x, y, z) = (x + y)z$$

e.g. x = -2, y = 5, z = -4

$$q = x + y \qquad \frac{\partial q}{\partial x} = 1, \frac{\partial q}{\partial y} = 1$$

$$f = qz \qquad \frac{\partial f}{\partial q} = z, \frac{\partial f}{\partial z} = q$$

Want: $\frac{\partial f}{\partial x}, \frac{\partial f}{\partial y}, \frac{\partial f}{\partial z}$



A simple BP example

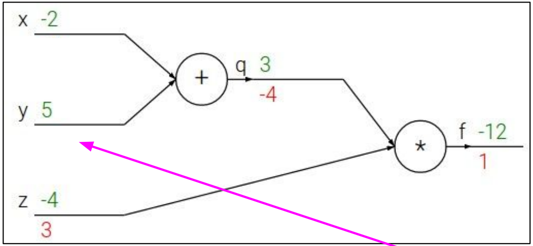
$$f(x,y,z) = (x+y)z$$

e.g. x = -2, y = 5, z = -4

$$q = x + y \qquad \frac{\partial q}{\partial x} = 1, \frac{\partial q}{\partial y} = 1$$

$$f = qz \qquad \frac{\partial f}{\partial q} = z, \frac{\partial f}{\partial z} = q$$

Want: $\frac{\partial f}{\partial x}, \frac{\partial f}{\partial y}, \frac{\partial f}{\partial z}$



$\frac{\partial f}{\partial y}$

A simple BP example

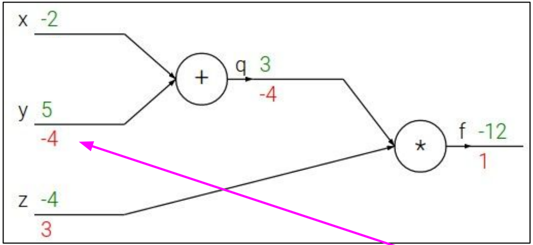
$$f(x,y,z) = (x+y)z$$

e.g. x = -2, y = 5, z = -4

$$q = x + y \qquad \frac{\partial q}{\partial x} = 1, \frac{\partial q}{\partial y} = 1$$

$$f = qz \qquad \frac{\partial f}{\partial q} = z, \frac{\partial f}{\partial z} = q$$

Want: $\frac{\partial f}{\partial x}, \frac{\partial f}{\partial y}, \frac{\partial f}{\partial z}$



$\frac{\partial f}{\partial y}$

Chain rule:

$$\frac{\partial f}{\partial y} = \frac{\partial f}{\partial q}\frac{\partial q}{\partial y}$$

A simple BP example

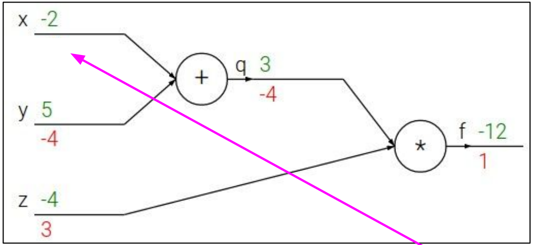
$$f(x,y,z) = (x+y)z$$

e.g. x = -2, y = 5, z = -4

$$q = x + y \qquad \frac{\partial q}{\partial x} = 1, \frac{\partial q}{\partial y} = 1$$

$$f = qz \qquad \frac{\partial f}{\partial q} = z, \frac{\partial f}{\partial z} = q$$

Want: $\frac{\partial f}{\partial x}, \frac{\partial f}{\partial y}, \frac{\partial f}{\partial z}$



$\frac{\partial f}{\partial x}$

A simple BP example

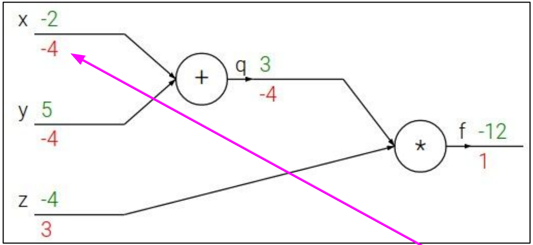
$$f(x, y, z) = (x + y)z$$

e.g. x = -2, y = 5, z = -4

$$q = x + y \qquad \frac{\partial q}{\partial x} = 1, \frac{\partial q}{\partial y} = 1$$

$$f = qz \qquad \frac{\partial f}{\partial q} = z, \frac{\partial f}{\partial z} = q$$

Want: $\frac{\partial f}{\partial x}, \frac{\partial f}{\partial y}, \frac{\partial f}{\partial z}$



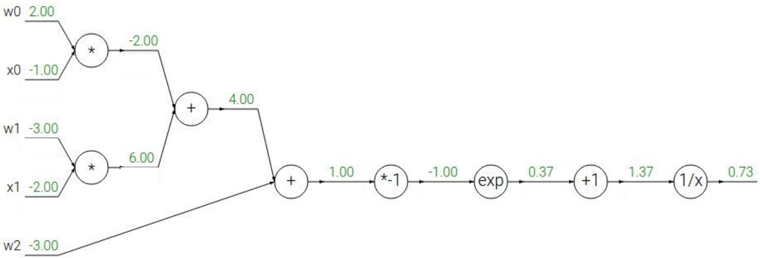
$\frac{\partial f}{\partial x}$

Chain rule:

$$\frac{\partial f}{\partial x} = \frac{\partial f}{\partial q} \frac{\partial q}{\partial x}$$

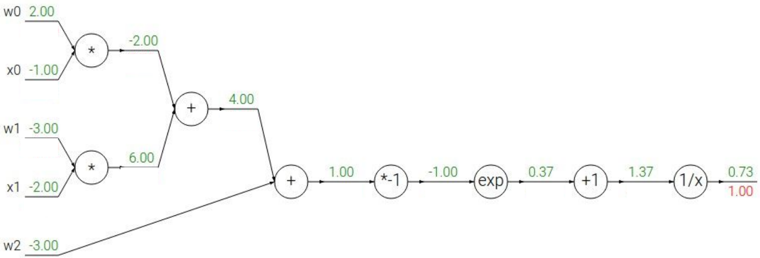
Yet another BP example

Another example: $$f(w,x) = \frac{1}{1+e^{-(w_0x_0+w_1x_1+w_2)}}$$



Yet another BP example

Another example: $$f(w,x) = \frac{1}{1+e^{-(w_0x_0+w_1x_1+w_2)}}$$

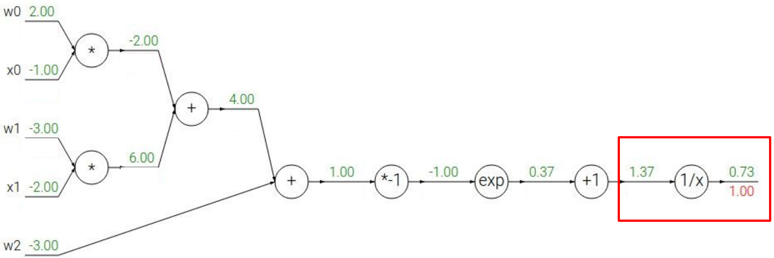


$f(x) = e^x \quad \rightarrow \quad \frac{df}{dx} = e^x$ | $f(x) = \frac{1}{x} \quad \rightarrow \quad \frac{df}{dx} = -1/x^2$

$f_a(x) = ax \quad \rightarrow \quad \frac{df}{dx} = a$ | $f_c(x) = c + x \quad \rightarrow \quad \frac{df}{dx} = 1$

Yet another BP example

Another example: $$f(w,x) = \frac{1}{1+e^{-(w_0x_0+w_1x_1+w_2)}}$$



$$f(x) = e^x \quad \rightarrow \quad \frac{df}{dx} = e^x$$

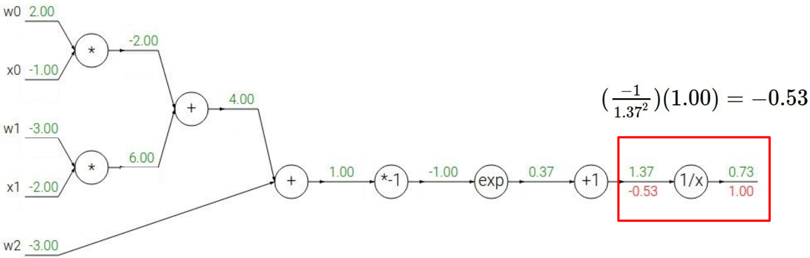
$$f_a(x) = ax \quad \rightarrow \quad \frac{df}{dx} = a$$

$$f(x) = \frac{1}{x} \quad \rightarrow \quad \frac{df}{dx} = -1/x^2$$

$$f_c(x) = c + x \quad \rightarrow \quad \frac{df}{dx} = 1$$

Yet another BP example

Another example: $$f(w,x) = \frac{1}{1+e^{-(w_0x_0+w_1x_1+w_2)}}$$



$f(x) = e^x \quad \rightarrow \quad \frac{df}{dx} = e^x$

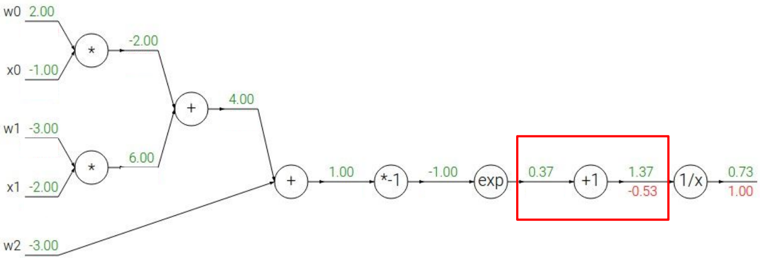
$f_a(x) = ax \quad \rightarrow \quad \frac{df}{dx} = a$

$f(x) = \frac{1}{x} \quad \rightarrow \quad \frac{df}{dx} = -1/x^2$

$f_c(x) = c + x \quad \rightarrow \quad \frac{df}{dx} = 1$

Yet another BP example

Another example: $$f(w,x) = \frac{1}{1+e^{-(w_0x_0+w_1x_1+w_2)}}$$



$f(x) = e^x \quad \rightarrow \quad \frac{df}{dx} = e^x$

$f_a(x) = ax \quad \rightarrow \quad \frac{df}{dx} = a$

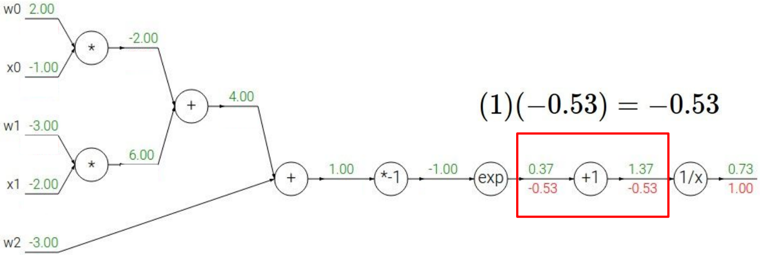
$f(x) = \frac{1}{x} \quad \rightarrow \quad \frac{df}{dx} = -1/x^2$

$f_c(x) = c + x \quad \rightarrow \quad \frac{df}{dx} = 1$

Yet another BP example

Another example: $$f(w,x) = \frac{1}{1+e^{-(w_0x_0+w_1x_1+w_2)}}$$

$$(1)(-0.53) = -0.53$$



$$f(x) = e^x \quad \rightarrow \quad \frac{df}{dx} = e^x$$

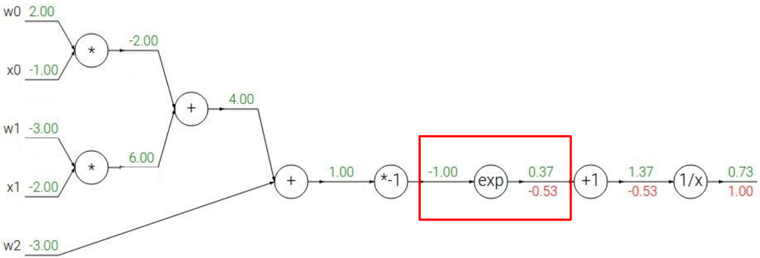
$$f_a(x) = ax \quad \rightarrow \quad \frac{df}{dx} = a$$

$$f(x) = \frac{1}{x} \quad \rightarrow \quad \frac{df}{dx} = -1/x^2$$

$$f_c(x) = c + x \quad \rightarrow \quad \frac{df}{dx} = 1$$

Yet another BP example

Another example: $f(w,x) = \frac{1}{1+e^{-(w_0x_0+w_1x_1+w_2)}}$



$f(x) = e^x \quad \rightarrow \quad \frac{df}{dx} = e^x$

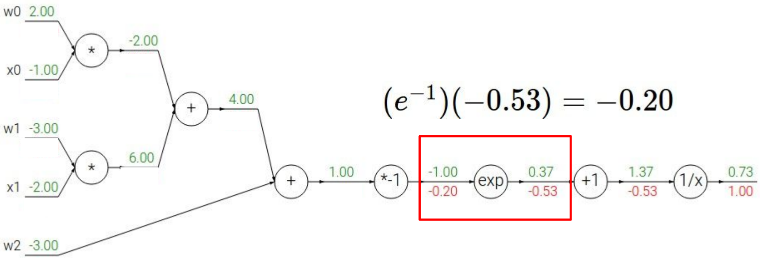
$f_a(x) = ax \quad \rightarrow \quad \frac{df}{dx} = a$

$f(x) = \frac{1}{x} \quad \rightarrow \quad \frac{df}{dx} = -1/x^2$

$f_c(x) = c + x \quad \rightarrow \quad \frac{df}{dx} = 1$

Yet another BP example

Another example: $$f(w,x) = \frac{1}{1+e^{-(w_0x_0+w_1x_1+w_2)}}$$

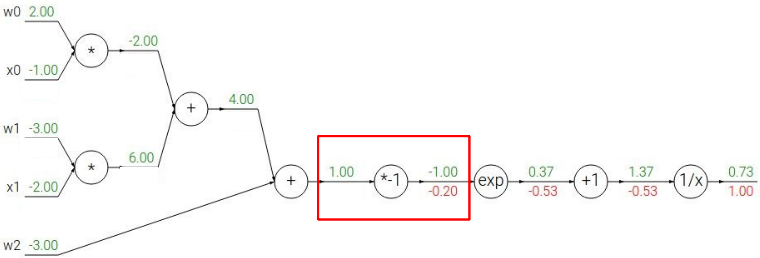


$$(e^{-1})(-0.53) = -0.20$$

$f(x) = e^x$	$\rightarrow$	$\frac{df}{dx} = e^x$	$f(x) = \frac{1}{x}$	$\rightarrow$	$\frac{df}{dx} = -1/x^2$
$f_a(x) = ax$	$\rightarrow$	$\frac{df}{dx} = a$	$f_c(x) = c + x$	$\rightarrow$	$\frac{df}{dx} = 1$

Yet another BP example

Another example: $$f(w,x) = \frac{1}{1+e^{-(w_0x_0+w_1x_1+w_2)}}$$



$f(x) = e^x \quad \rightarrow \quad \frac{df}{dx} = e^x$

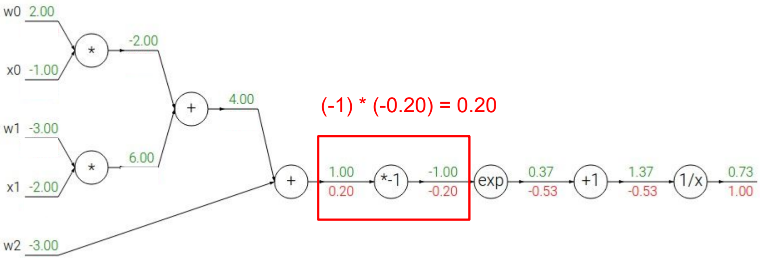
$f_a(x) = ax \quad \rightarrow \quad \frac{df}{dx} = a$

$f(x) = \frac{1}{x} \quad \rightarrow \quad \frac{df}{dx} = -1/x^2$

$f_c(x) = c + x \quad \rightarrow \quad \frac{df}{dx} = 1$

Yet another BP example

Another example: $f(w,x) = \frac{1}{1+e^{-(w_0x_0+w_1x_1+w_2)}}$



$f(x) = e^x \quad \rightarrow \quad \frac{df}{dx} = e^x$

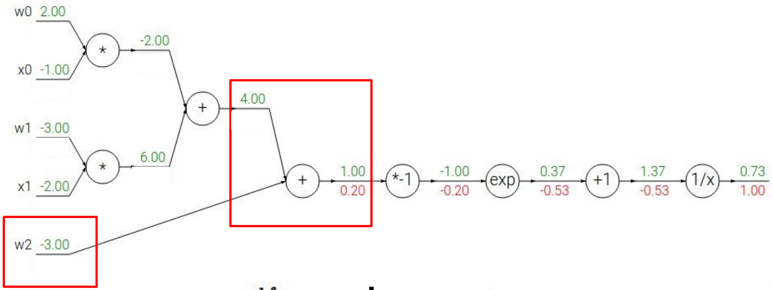
$f_a(x) = ax \quad \rightarrow \quad \frac{df}{dx} = a$

$f(x) = \frac{1}{x} \quad \rightarrow \quad \frac{df}{dx} = -1/x^2$

$f_c(x) = c + x \quad \rightarrow \quad \frac{df}{dx} = 1$

Yet another BP example

Another example: $$f(w,x) = \frac{1}{1+e^{-(w_0x_0+w_1x_1+w_2)}}$$

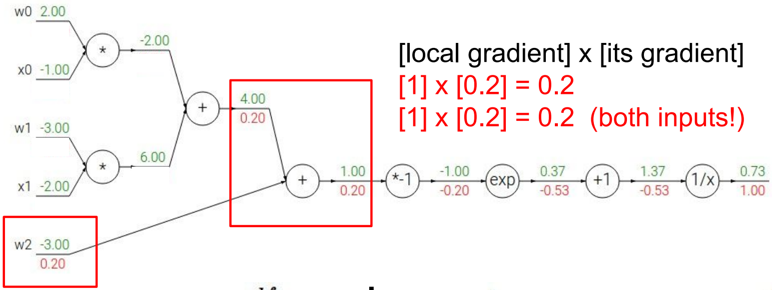


$$f(x) = e^x \quad \rightarrow \quad \frac{df}{dx} = e^x \qquad\qquad f(x) = \frac{1}{x} \quad \rightarrow \quad \frac{df}{dx} = -1/x^2$$

$$f_a(x) = ax \quad \rightarrow \quad \frac{df}{dx} = a \qquad\qquad f_c(x) = c + x \quad \rightarrow \quad \frac{df}{dx} = 1$$

Yet another BP example

Another example: $f(w,x) = \frac{1}{1+e^{-(w_0x_0+w_1x_1+w_2)}}$



[local gradient] x [its gradient]
[1] x [0.2] = 0.2
[1] x [0.2] = 0.2 (both inputs!)

$f(x) = e^x \quad \rightarrow \quad \frac{df}{dx} = e^x$

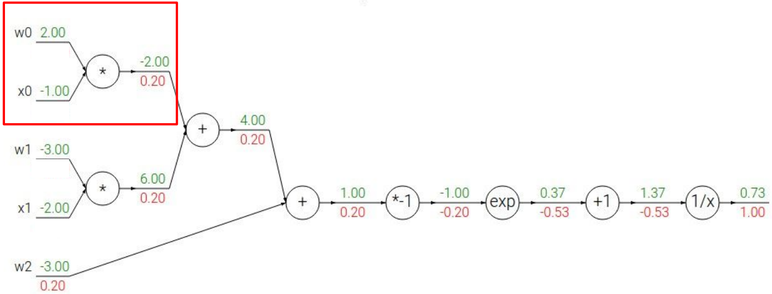
$f_a(x) = ax \quad \rightarrow \quad \frac{df}{dx} = a$

$f(x) = \frac{1}{x} \quad \rightarrow \quad \frac{df}{dx} = -1/x^2$

$f_c(x) = c + x \quad \rightarrow \quad \frac{df}{dx} = 1$

Yet another BP example

Another example: $$f(w,x) = \frac{1}{1+e^{-(w_0x_0+w_1x_1+w_2)}}$$

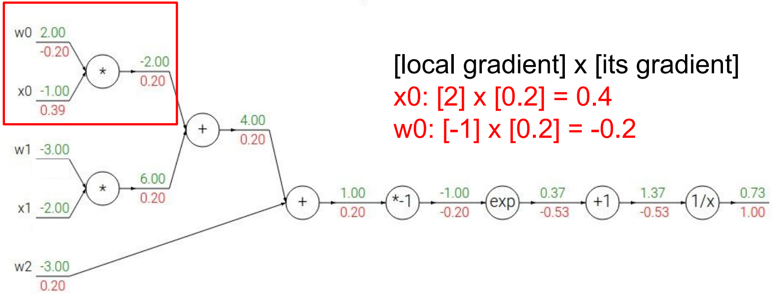


$$f(x) = e^x \quad \rightarrow \quad \frac{df}{dx} = e^x \qquad\qquad f(x) = \frac{1}{x} \quad \rightarrow \quad \frac{df}{dx} = -1/x^2$$

$$f_a(x) = ax \quad \rightarrow \quad \frac{df}{dx} = a \qquad\qquad f_c(x) = c + x \quad \rightarrow \quad \frac{df}{dx} = 1$$

Yet another BP example

Another example: $f(w,x) = \frac{1}{1+e^{-(w_0x_0+w_1x_1+w_2)}}$



[local gradient] x [its gradient]
x0: [2] x [0.2] = 0.4
w0: [-1] x [0.2] = -0.2

$f(x) = e^x \rightarrow \frac{df}{dx} = e^x$ | $f(x) = \frac{1}{x} \rightarrow \frac{df}{dx} = -1/x^2$

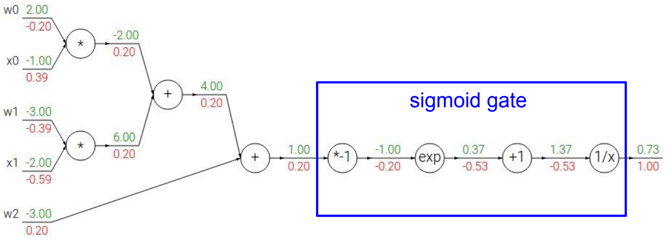
$f_a(x) = ax \rightarrow \frac{df}{dx} = a$ | $f_c(x) = c + x \rightarrow \frac{df}{dx} = 1$

Breaking down at different granularities

$$f(w,x) = \frac{1}{1+e^{-(w_0x_0+w_1x_1+w_2)}}$$

$$\boxed{\sigma(x) = \frac{1}{1+e^{-x}}}$$ sigmoid function

$$\frac{d\sigma(x)}{dx} = \frac{e^{-x}}{(1+e^{-x})^2} = \left(\frac{1+e^{-x}-1}{1+e^{-x}}\right)\left(\frac{1}{1+e^{-x}}\right) = (1-\sigma(x))\,\sigma(x)$$



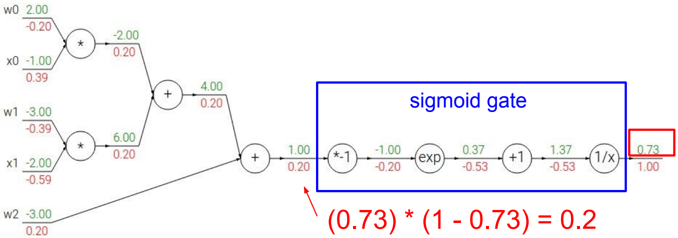
Breaking down at different granularities

$$f(w,x) = \frac{1}{1+e^{-(w_0x_0+w_1x_1+w_2)}}$$

$$\sigma(x) = \frac{1}{1+e^{-x}}$$

sigmoid function

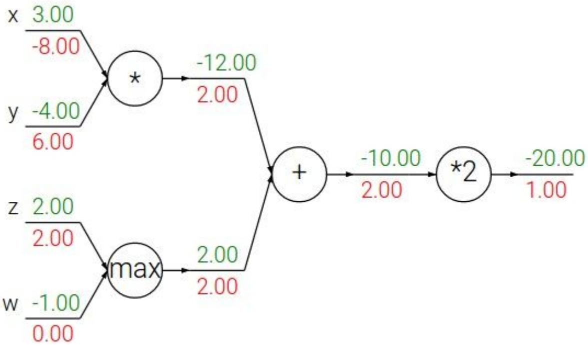
$$\frac{d\sigma(x)}{dx} = \frac{e^{-x}}{(1+e^{-x})^2} = \left(\frac{1+e^{-x}-1}{1+e^{-x}}\right)\left(\frac{1}{1+e^{-x}}\right) = (1-\sigma(x))\,\sigma(x)$$



Think, pair, share

Patterns in backward flow

add gate: gradient distributor

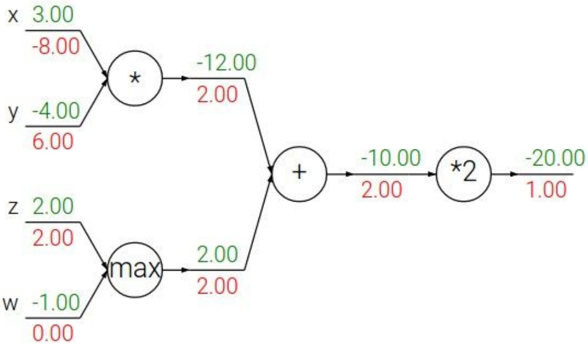


Think, pair, share

Patterns in backward flow

add gate: gradient distributor

Q: What is a **max** gate?

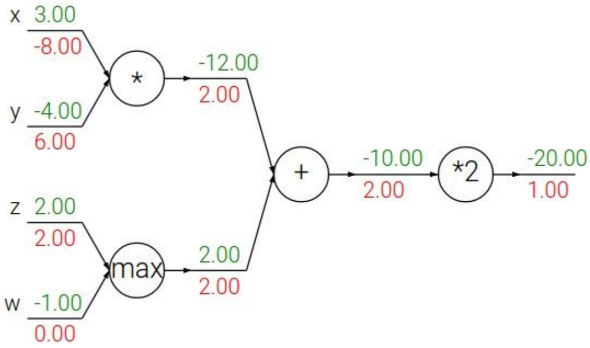


Think, pair, share

Patterns in backward flow

add gate: gradient distributor

max gate: gradient router



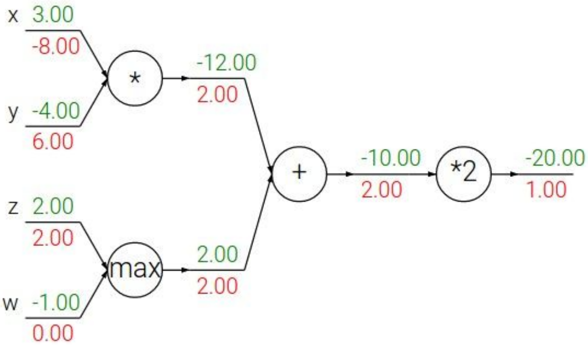
Think, pair, share

Patterns in backward flow

add gate: gradient distributor

max gate: gradient router

Q: What is a **mul** gate?



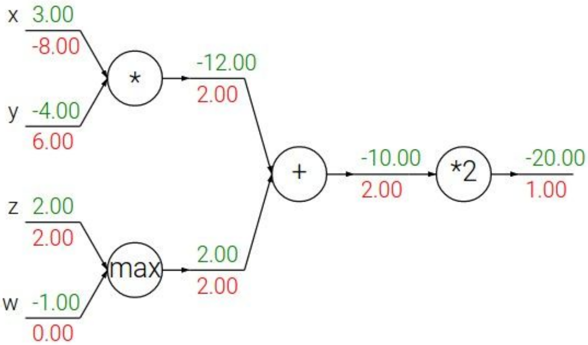
Think, pair, share

Patterns in backward flow

add gate: gradient distributor

max gate: gradient router

mul gate: gradient switcher

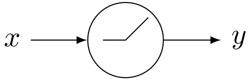


More examples: RELU

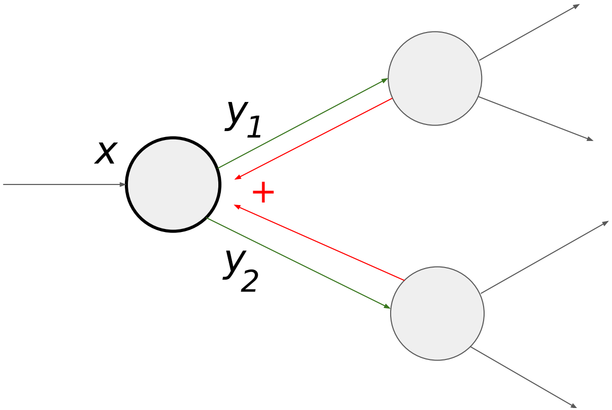
- Consider a “half-linear” function with negative side chopped off. That is,

$$f(x) = \begin{cases} x & x \geq 0 \\ 0 & \text{otherwise} \end{cases}$$

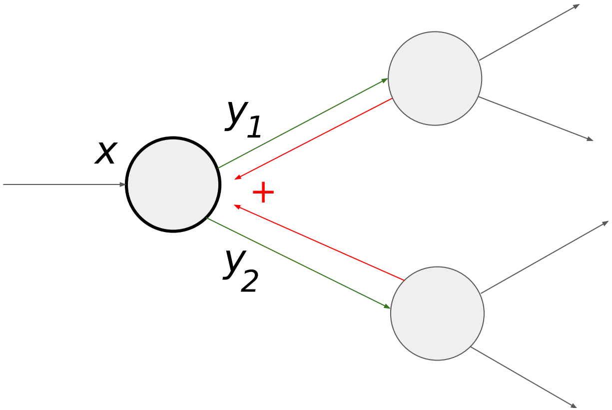
  - This is known to be the rectified linear unit (RELU)
- How should the gradient be propagated back?



Merging gradients



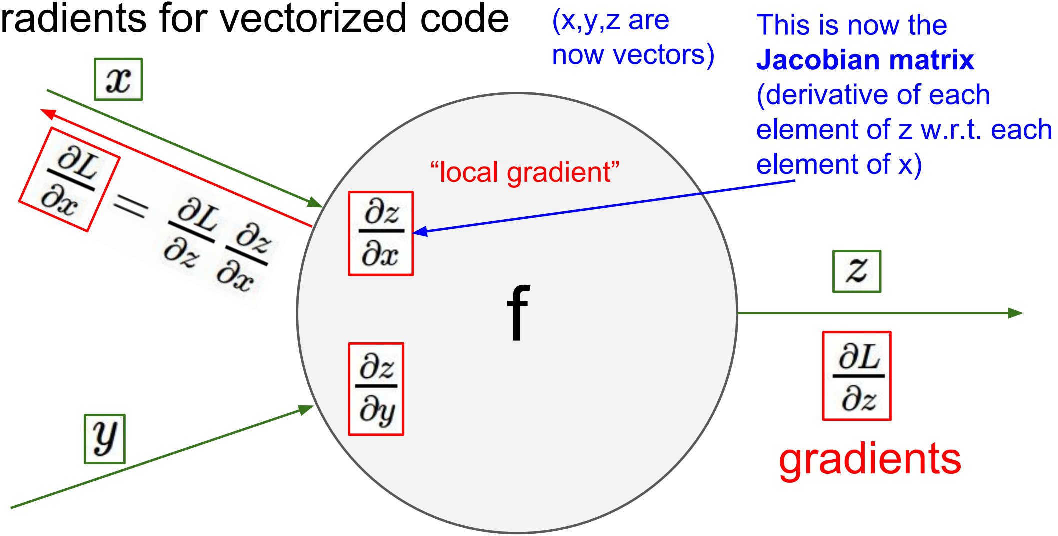
Merging gradients



$$\frac{\partial L(y_1(x), y_2(x))}{\partial x} = \frac{\partial L}{\partial y_1}\frac{\partial y_1}{\partial x_1} + \frac{\partial L}{\partial y_2}\frac{\partial y_2}{\partial x_1}$$

Handling vector variables

Gradients for vectorized code



Handling vector variables

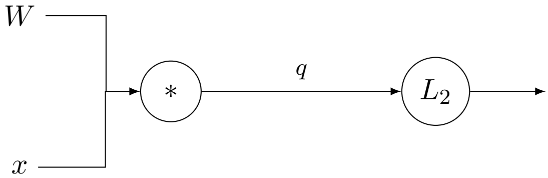
A vectorized example: $L = \|q - \tilde{q}\|^2 = \|Wx - \tilde{q}\|^2$

Handling vector variables

A vectorized example: $L = \|q - \tilde{q}\|^2 = \|Wx - \tilde{q}\|^2$

Handling vector variables

A vectorized example: $L = \|q - \tilde{q}\|^2 = \|Wx - \tilde{q}\|^2$

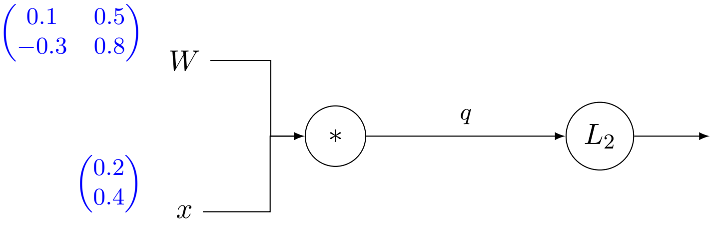


$$q = Wx = \begin{pmatrix} W_{1,1}x_1 + \cdots + W_{1,n}x_n \\ \vdots \\ W_{n,1}x_1 + \cdots + W_{n,n}x_n \end{pmatrix}$$

$$L(q) = \|q - \tilde{q}\|^2$$

Handling vector variables

A vectorized example: $L = \|q - \tilde{q}\|^2 = \|Wx - \tilde{q}\|^2$

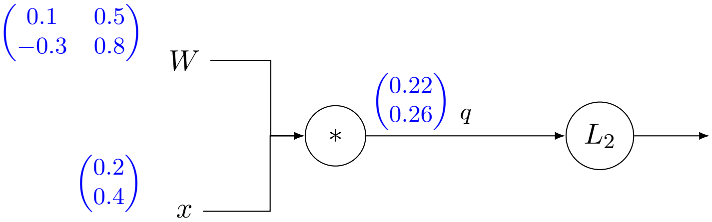


$$q = Wx = \begin{pmatrix} W_{1,1}x_1 + \cdots + W_{1,n}x_n \\ \vdots \\ W_{n,1}x_1 + \cdots + W_{n,n}x_n \end{pmatrix}$$

$$L(q) = \|q - \tilde{q}\|^2$$

Handling vector variables

A vectorized example: $L = \|q - \tilde{q}\|^2 = \|Wx - \tilde{q}\|^2$



$$q = Wx = \begin{pmatrix} W_{1,1}x_1 + \cdots + W_{1,n}x_n \\ \vdots \\ W_{n,1}x_1 + \cdots + W_{n,n}x_n \end{pmatrix}$$

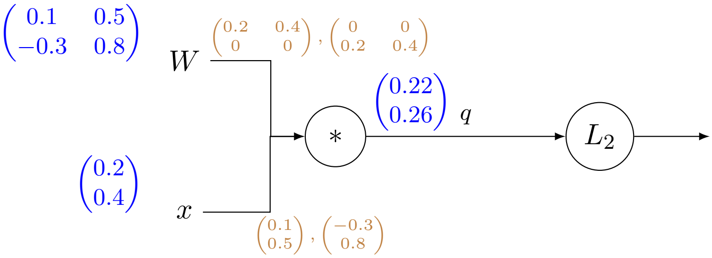
$$L(q) = \|q - \tilde{q}\|^2$$

$$\frac{\partial q_k}{\partial W_{i,j}} = \delta_{i,k} x_j$$

$$\frac{\partial q_k}{\partial x_i} = W_{k,i}$$

Handling vector variables

A vectorized example: $L = \|q - \tilde{q}\|^2 = \|Wx - \tilde{q}\|^2$



$$q = Wx = \begin{pmatrix} W_{1,1}x_1 + \cdots + W_{1,n}x_n \\ \vdots \\ W_{n,1}x_1 + \cdots + W_{n,n}x_n \end{pmatrix}$$

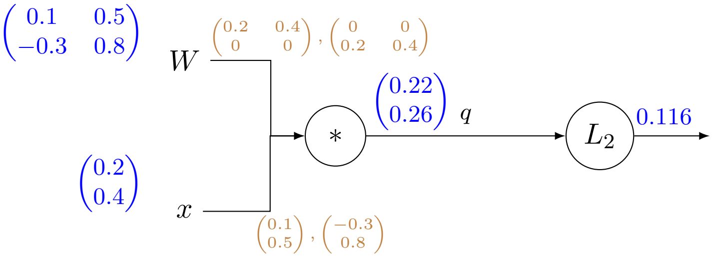
$$L(q) = \|q - \tilde{q}\|^2$$

$$\frac{\partial q_k}{\partial W_{i,j}} = \delta_{i,k} x_j$$

$$\frac{\partial q_k}{\partial x_i} = W_{k,i}$$

Handling vector variables

A vectorized example: $L = \|q - \tilde{q}\|^2 = \|Wx - \tilde{q}\|^2$



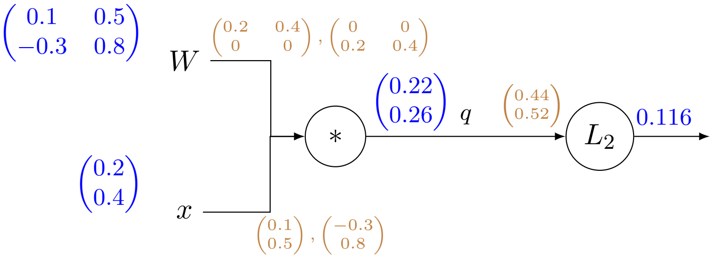
$$q = Wx = \begin{pmatrix} W_{1,1}x_1 + \cdots + W_{1,n}x_n \\ \vdots \\ W_{n,1}x_1 + \cdots + W_{n,n}x_n \end{pmatrix}$$

$$L(q) = \|q - \tilde{q}\|^2$$

$$\frac{\partial L}{\partial q_i} = 2(q_i - \tilde{q}_i)$$

Handling vector variables

A vectorized example: $L = \|q - \tilde{q}\|^2 = \|Wx - \tilde{q}\|^2$



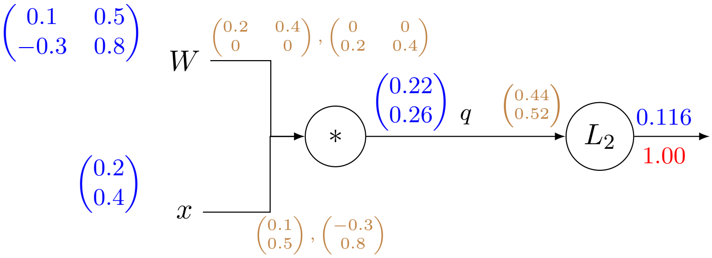
$$q = Wx = \begin{pmatrix} W_{1,1}x_1 + \cdots + W_{1,n}x_n \\ \vdots \\ W_{n,1}x_1 + \cdots + W_{n,n}x_n \end{pmatrix}$$

$$L(q) = \|q - \tilde{q}\|^2$$

$$\frac{\partial L}{\partial q_i} = 2(q_i - \tilde{q}_i)$$

Handling vector variables

A vectorized example: $L = \|q - \tilde{q}\|^2 = \|Wx - \tilde{q}\|^2$



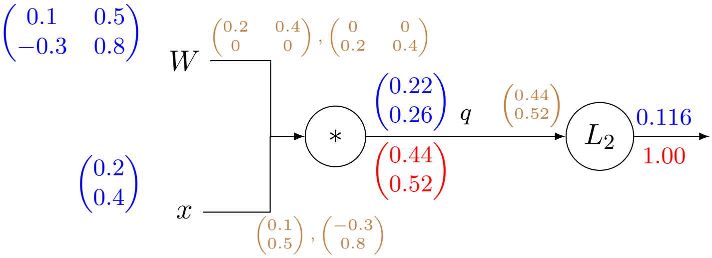
$$q = Wx = \begin{pmatrix} W_{1,1}x_1 + \cdots + W_{1,n}x_n \\ \vdots \\ W_{n,1}x_1 + \cdots + W_{n,n}x_n \end{pmatrix}$$

$$L(q) = \|q - \tilde{q}\|^2$$

$$\frac{\partial L}{\partial q_i} = 1.00 \cdot \frac{\partial L}{\partial q_i}$$

Handling vector variables

A vectorized example: $L = \|q - \tilde{q}\|^2 = \|Wx - \tilde{q}\|^2$



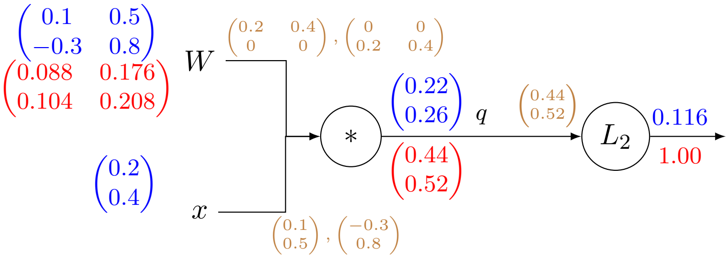
$$q = Wx = \begin{pmatrix} W_{1,1}x_1 + \cdots + W_{1,n}x_n \\ \vdots \\ W_{n,1}x_1 + \cdots + W_{n,n}x_n \end{pmatrix}$$

$$L(q) = \|q - \tilde{q}\|^2$$

$$\frac{\partial L}{\partial q_i} = 1.00 \cdot \frac{\partial L}{\partial q_i}$$

Handling vector variables

A vectorized example: $L = \|q - \tilde{q}\|^2 = \|Wx - \tilde{q}\|^2$



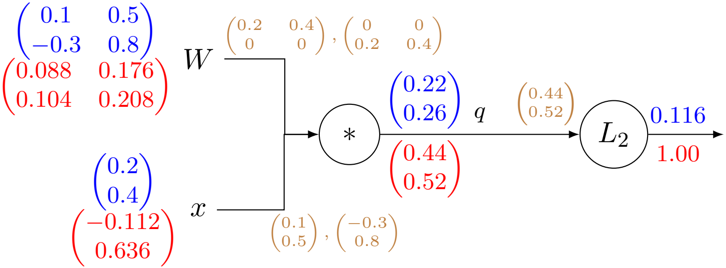
$$q = Wx = \begin{pmatrix} W_{1,1}x_1 + \cdots + W_{1,n}x_n \\ \vdots \\ W_{n,1}x_1 + \cdots + W_{n,n}x_n \end{pmatrix}$$

$$L(q) = \|q - \tilde{q}\|^2$$

$$\frac{\partial L}{\partial W_{i,j}} = \frac{\partial L}{\partial q_1}\frac{\partial q_1}{\partial W_{i,j}} + \frac{\partial L}{\partial q_2}\frac{\partial q_2}{\partial W_{i,j}}$$

Handling vector variables

A vectorized example: $L = \|q - \tilde{q}\|^2 = \|Wx - \tilde{q}\|^2$



$$q = Wx = \begin{pmatrix} W_{1,1}x_1 + \cdots + W_{1,n}x_n \\ \vdots \\ W_{n,1}x_1 + \cdots + W_{n,n}x_n \end{pmatrix}$$

$$L(q) = \|q - \tilde{q}\|^2$$

$$\frac{\partial L}{\partial x_i} = \frac{\partial L}{\partial q_1}\frac{\partial q_1}{\partial x_i} + \frac{\partial L}{\partial q_2}\frac{\partial q_2}{\partial x_i}$$

Example: Softmax

- $\sigma_l(o) = \frac{\exp(o_l)}{\sum_k \exp(o_k)}$
- $\frac{\partial \sigma_i(o)}{\partial o_j} = -\frac{\exp(o_i)}{\left(\sum_k \exp(o_k)\right)^2} \exp(o_j) = -\sigma_i(o)\sigma_j(o)$
- $\frac{\partial \sigma_i(o)}{\partial o_i} = \frac{\exp(o_i)}{\sum_k \exp(o_k)} - \frac{\exp(o_i)}{\left(\sum_k \exp(o_k)\right)^2} \exp(o_j) = \sigma_i(o)(1 - \sigma_j(o))$

Example: Softmax + Cross-entropy

- $L = -\sum_l q_l \log \sigma_l(o)$
- $\frac{\partial L}{\partial \sigma_l} = -\frac{q_l}{\sigma_l}$
- $\frac{\partial L}{\partial o_i} = \sum_l -\frac{q_l}{\sigma_l}\frac{\partial \sigma_l}{\partial o_i} = \sum_{l \neq i} \frac{q_l}{\sigma_l}\sigma_i(o)\sigma_l(o) - \frac{q_i}{\sigma_i}\sigma_i(o)(1-\sigma_i(o))$
  $= \sigma_i(1-q_i) - q_i(1-\sigma_i) = \sigma_i - q_i$
- Makes lot of sense!

Example: IoU (reference)

- Interception over union is commonly used to quantify segmentation quality for image segmentation
- For pixel v, X_v is the estimated mask and $Y_v \in \{0, 1\}$ is the ground truth

Example: IoU (reference)

- Interception over union is commonly used to quantify segmentation quality for image segmentation
- For pixel v, X_v is the estimated mask and $Y_v \in \{0,1\}$ is the ground truth
- $IoU(X) = \frac{I(X)}{U(X)}$, where $I(X) \approx \sum_v X_v Y_v$ and $U(X) \approx \sum_v (X_v + Y_v - X_v Y_v)$
- $\frac{\partial IoU(X)}{\partial X_v}$

Example: IoU (reference)

- Interception over union is commonly used to quantify segmentation quality for image segmentation
- For pixel v, X_v is the estimated mask and $Y_v \in \{0, 1\}$ is the ground truth
- $IoU(X) = \frac{I(X)}{U(X)}$, where $I(X) \approx \sum_v X_v Y_v$ and $U(X) \approx \sum_v (X_v + Y_v - X_v Y_v)$
- $\frac{\partial IoU(X)}{\partial X_v} = \frac{U(X)\frac{\partial I(X)}{\partial X_v} - I(X)\frac{\partial U(X)}{\partial X_v}}{U^2(X)}$

Example: IoU (reference)

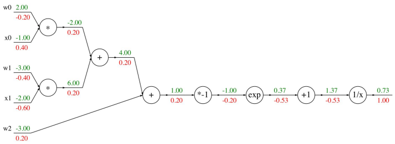
- Interception over union is commonly used to quantify segmentation quality for image segmentation
- For pixel v, X_v is the estimated mask and $Y_v \in \{0, 1\}$ is the ground truth
- $IoU(X) = \frac{I(X)}{U(X)}$, where $I(X) \approx \sum_v X_v Y_v$ and $U(X) \approx \sum_v (X_v + Y_v - X_v Y_v)$
- $\frac{\partial IoU(X)}{\partial X_v} = \frac{U(X)\frac{\partial I(X)}{\partial X_v} - I(X)\frac{\partial U(X)}{\partial X_v}}{U^2(X)} = \frac{U(X)Y_v - I(X)(1-Y_v)}{U(X)^2}$

Example: IoU (reference)

- Interception over union is commonly used to quantify segmentation quality for image segmentation
- For pixel v, X_v is the estimated mask and $Y_v \in \{0,1\}$ is the ground truth
- $IoU(X) = \frac{I(X)}{U(X)}$, where $I(X) \approx \sum_v X_v Y_v$ and $U(X) \approx \sum_v (X_v + Y_v - X_v Y_v)$
- $\frac{\partial IoU(X)}{\partial X_v} = \frac{U(X)\frac{\partial I(X)}{\partial X_v} - I(X)\frac{\partial U(X)}{\partial X_v}}{U^2(X)} = \frac{U(X)Y_v - I(X)(1-Y_v)}{U(X)^2} \Rightarrow \frac{\partial IoU(X)}{\partial X_v} = \begin{cases} \frac{1}{U(X)} & Y_v = 1 \\ -\frac{I(X)}{U(X)^2} & Y_v = 0 \end{cases}$

Implementation

Modularized implementation: forward / backward API

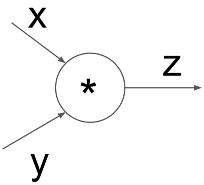


Graph (or Net) object *(rough psuedo code)*

```
class ComputationalGraph(object):
    #...
    def forward(inputs):
        # 1. [pass inputs to input gates...]
        # 2. forward the computational graph:
        for gate in self.graph.nodes_topologically_sorted():
            gate.forward()
        return loss # the final gate in the graph outputs the loss
    def backward():
        for gate in reversed(self.graph.nodes_topologically_sorted()):
            gate.backward() # little piece of backprop (chain rule applied)
        return inputs_gradients
```

Implementation

Modularized implementation: forward / backward API



(x,y,z are scalars)

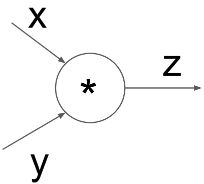
```
class MultiplyGate(object):
    def forward(x,y):
        z = x*y
        return z
    def backward(dz):
        # dx = ... #todo
        # dy = ... #todo
        return [dx, dy]
```

$$\frac{\partial L}{\partial z}$$

$$\frac{\partial L}{\partial x}$$

Implementation

Modularized implementation: forward / backward API



(x,y,z are scalars)

```
class MultiplyGate(object):
    def forward(x,y):
        z = x*y
        self.x = x # must keep these around!
        self.y = y
        return z
    def backward(dz):
        dx = self.y * dz # [dz/dx * dL/dz]
        dy = self.x * dz # [dz/dy * dL/dz]
        return [dx, dy]
```

Remark of BP

- During the forward pass, each computing unit will evaluate the output and also the corresponding local derivatives of the output w.r.t. the inputs

Remark of BP

- During the forward pass, each computing unit will evaluate the output and also the corresponding local derivatives of the output w.r.t. the inputs
- During the backward pass, the local derivatives and the evaluated outputs will be "consumed" to compute the overall derivatives

Remark of BP

- During the forward pass, each computing unit will evaluate the output and also the corresponding local derivatives of the output w.r.t. the inputs
- During the backward pass, the local derivatives and the evaluated outputs will be "consumed" to compute the overall derivatives
    - For a large network, there can be a large spike of memory consumption during the forward pass

Remark of BP

- During the forward pass, each computing unit will evaluate the output and also the corresponding local derivatives of the output w.r.t. the inputs
- During the backward pass, the local derivatives and the evaluated outputs will be "consumed" to compute the overall derivatives
  - For a large network, there can be a large spike of memory consumption during the forward pass
- Note that BP only computes the gradients. It does not perform the optimization. Sometimes you may hear people said that they trained their networks with BP. What they said was not literally right. We will discuss more on optimizer later today

Remark of BP

- During the forward pass, each computing unit will evaluate the output and also the corresponding local derivatives of the output w.r.t. the inputs
- During the backward pass, the local derivatives and the evaluated outputs will be "consumed" to compute the overall derivatives
  - For a large network, there can be a large spike of memory consumption during the forward pass
- Note that BP only computes the gradients. It does not perform the optimization. Sometimes you may hear people said that they trained their networks with BP. What they said was not literally right. We will discuss more on optimizer later today
- With BP in place, why we still can't train deep networks?

Gradient vanishing and exploding problems

- As each training step is nothing more than going approximately downhill along the negative gradient

Gradient vanishing and exploding problems

- As each training step is nothing more than going approximately downhill along the negative gradient
    - Gradient vanishing: no training can continue as gradient goes to zero

Gradient vanishing and exploding problems

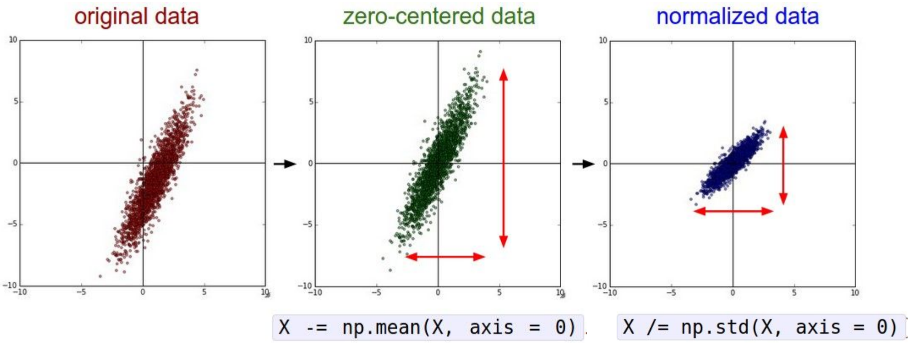
- As each training step is nothing more than going approximately downhill along the negative gradient
  - Gradient vanishing: no training can continue as gradient goes to zero
  - Gradient exploding: training dies as gradients goes overflow and usually resulting in NaN

Gradient vanishing and exploding problems

- As each training step is nothing more than going approximately downhill along the negative gradient
  - Gradient vanishing: no training can continue as gradient goes to zero
  - Gradient exploding: training dies as gradients goes overflow and usually resulting in NaN
- As layers stack up, these problems become more and more likely to happen
  - These make training deep ANN challenging

Input preprocessing

Step 1: Preprocess the data

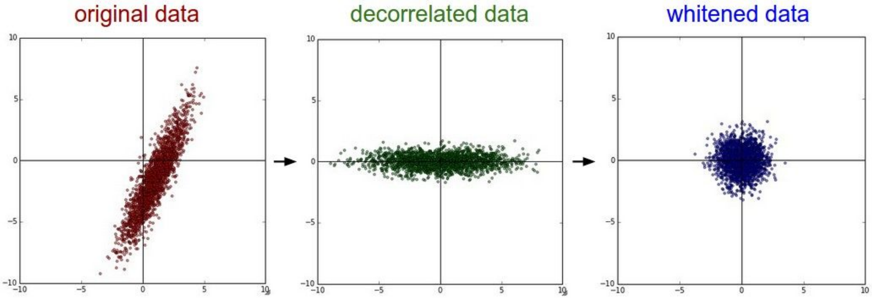


(Assume X [NxD] is data matrix, each example in a row)

Input preprocessing

Step 1: Preprocess the data

In practice, you may also see **PCA** and **Whitening** of the data



(data has diagonal covariance matrix)

(covariance matrix is the identity matrix)

Input preprocessing

TLDR: In practice for Images: center only

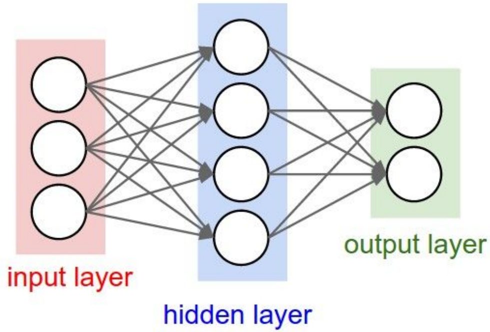
e.g. consider CIFAR-10 example with [32,32,3] images

- Subtract the mean image (e.g. AlexNet)
  (mean image = [32,32,3] array)
- Subtract per-channel mean (e.g. VGGNet)
  (mean along each channel = 3 numbers)

Not common to normalize variance, to do PCA or whitening

Weight initialization

- Q: what happens when W=0 init is used?



Weight initialization

- First idea: **Small random numbers** (gaussian with zero mean and 1e-2 standard deviation)

```
W = 0.01* np.random.randn(D,H)
```

Weight initialization

- First idea: **Small random numbers**
(gaussian with zero mean and 1e-2 standard deviation)

```
W = 0.01* np.random.randn(D,H)
```

Works ~okay for small networks, but can lead to non-homogeneous distributions of activations across the layers of a network.

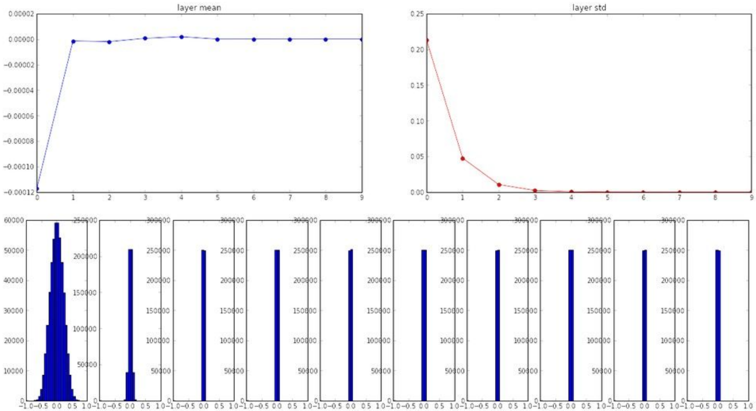
Weight initialization

Let's look at some activation statistics

- 10 layers
- 500 neurons per layer
- $tanh(\cdot)$ for activation
- $W = 0.01 * \text{np.random.randn(fan\_in, fan\_out)}$ as described in the last slide

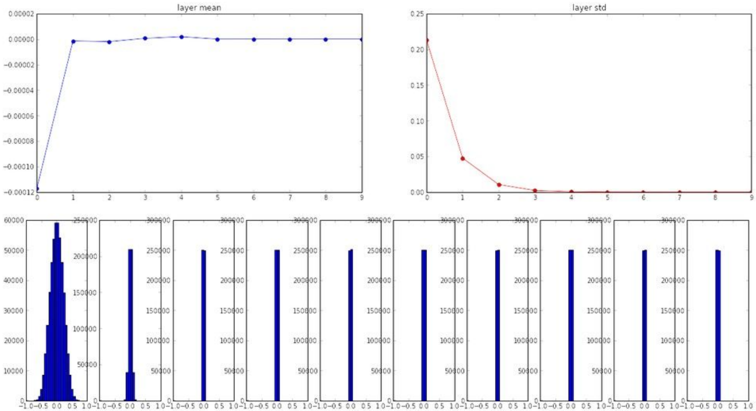
Weight initialization

```
input layer had mean 0.000927 and std 0.998388
hidden layer 1 had mean -0.000117 and std 0.213081
hidden layer 2 had mean -0.000001 and std 0.047551
hidden layer 3 had mean -0.000002 and std 0.010630
hidden layer 4 had mean 0.000001 and std 0.002378
hidden layer 5 had mean 0.000002 and std 0.000532
hidden layer 6 had mean -0.000000 and std 0.000119
hidden layer 7 had mean 0.000000 and std 0.000026
hidden layer 8 had mean -0.000000 and std 0.000006
hidden layer 9 had mean 0.000000 and std 0.000001
hidden layer 10 had mean -0.000000 and std 0.000000
```



Weight initialization

```
input layer had mean 0.000927 and std 0.998388
hidden layer 1 had mean -0.000117 and std 0.213081
hidden layer 2 had mean -0.000001 and std 0.047551
hidden layer 3 had mean -0.000002 and std 0.010630
hidden layer 4 had mean 0.000001 and std 0.002378
hidden layer 5 had mean 0.000002 and std 0.000532
hidden layer 6 had mean -0.000000 and std 0.000119
hidden layer 7 had mean 0.000000 and std 0.000026
hidden layer 8 had mean -0.000000 and std 0.000006
hidden layer 9 had mean 0.000000 and std 0.000001
hidden layer 10 had mean -0.000000 and std 0.000000
```



All activations become zero!

Q: think about the backward pass. What do the gradients look like?

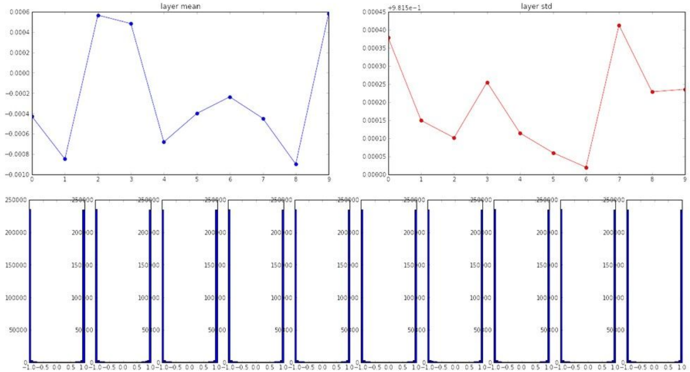
Hint: think about backward pass for a W*X gate.

Weight initialization

```
W = np.random.randn(fan_in, fan_out) * 1.0 # layer initialization
```

```
input layer had mean 0.001800 and std 1.001311
hidden layer 1 had mean -0.000430 and std 0.981879
hidden layer 2 had mean -0.000849 and std 0.981649
hidden layer 3 had mean 0.000566 and std 0.981601
hidden layer 4 had mean 0.000483 and std 0.981755
hidden layer 5 had mean -0.000682 and std 0.981614
hidden layer 6 had mean -0.000401 and std 0.981560
hidden layer 7 had mean -0.000237 and std 0.981520
hidden layer 8 had mean -0.000448 and std 0.981913
hidden layer 9 had mean -0.000899 and std 0.981728
hidden layer 10 had mean 0.000584 and std 0.981736
```

*1.0 instead of *0.01



Almost all neurons completely saturated, either -1 and 1. Gradients will be all zero.

Variance calibration for linear layer

Assume linear activation and zero-mean weights and inputs. And number of inputs is n. Then,

$$\mathrm{Var}(y) = \mathrm{Var}\left(\sum_{i}^{n} w_i x_i\right) = \sum_{i}^{n} \mathrm{Var}(w_i x_i)$$

$Var(XY) = E[X]^2 Var(Y) + E[Y]^2 Var(X) + Var(X)Var(Y)$

$$Var(XY) = E[(XY)^2] - E[XY]^2$$

$Var(XY) = E[X]^2 Var(Y) + E[Y]^2 Var(X) + Var(X)Var(Y)$

$$
\begin{aligned}
Var(XY) &= E[(XY)^2] - E[XY]^2 \\
&= E[X^2]E[Y^2] - E[X]^2 E[Y]^2
\end{aligned}
$$

$Var(XY) = E[X]^2 Var(Y) + E[Y]^2 Var(X) + Var(X)Var(Y)$

$$
\begin{aligned}
Var(XY) &= E[(XY)^2] - E[XY]^2 \\
&= E[X^2]E[Y^2] - E[X]^2E[Y]^2
\end{aligned}
$$

$$
\begin{aligned}
&Var(X)Var(Y) \\
&= (E[X^2] - E[X]^2)(E[Y^2] - E[Y]^2)
\end{aligned}
$$

$Var(XY) = E[X]^2 Var(Y) + E[Y]^2 Var(X) + Var(X)Var(Y)$

$$
\begin{aligned}
Var(XY) &= E[(XY)^2] - E[XY]^2 \\
&= E[X^2]E[Y^2] - E[X]^2 E[Y]^2
\end{aligned}
$$

$$
\begin{aligned}
& Var(X)Var(Y) \\
&= (E[X^2] - E[X]^2)(E[Y^2] - E[Y]^2) \\
&= E[X^2]E[Y^2] - E[X]^2 E[Y^2] - E[X^2]E[Y]^2 + E[X]^2 E[Y]^2
\end{aligned}
$$

$Var(XY) = E[X]^2 Var(Y) + E[Y]^2 Var(X) + Var(X)Var(Y)$

$$
\begin{aligned}
Var(XY) &= E[(XY)^2] - E[XY]^2 \\
&= E[X^2]E[Y^2] - E[X]^2 E[Y]^2
\end{aligned}
$$

$$
\begin{aligned}
& Var(X)Var(Y) \\
=& (E[X^2] - E[X]^2)(E[Y^2] - E[Y]^2) \\
=& E[X^2]E[Y^2] - E[X]^2 E[Y^2] - E[X^2]E[Y]^2 + E[X]^2 E[Y]^2 \\
=& E[X^2]E[Y^2] - E[X]^2 (E[Y^2] - E[Y]^2) \\
& E[Y]^2 (E[X^2] - E[X]^2) - E[X]^2 E[Y]^2
\end{aligned}
$$

$Var(XY) = E[X]^2 Var(Y) + E[Y]^2 Var(X) + Var(X)Var(Y)$

$$
\begin{aligned}
Var(XY) &= E[(XY)^2] - E[XY]^2 \\
&= E[X^2]E[Y^2] - E[X]^2E[Y]^2
\end{aligned}
$$

$$
\begin{aligned}
& Var(X)Var(Y) \\
=& (E[X^2] - E[X]^2)(E[Y^2] - E[Y]^2) \\
=& E[X^2]E[Y^2] - E[X]^2E[Y^2] - E[X^2]E[Y]^2 + E[X]^2E[Y]^2 \\
=& E[X^2]E[Y^2] - E[X]^2(E[Y^2] - E[Y]^2) \\
& E[Y]^2(E[X^2] - E[X]^2) - E[X]^2E[Y]^2 \\
=& Var(XY) - E[X]^2Var(Y) - E[Y]^2Var(X)
\end{aligned}
$$

Variance calibration for linear layer

Assume linear activation and zero-mean weights and inputs. And number of inputs is n. Then,

$$\begin{aligned}\mathrm{Var}(y) &= \mathrm{Var}\left(\sum_i^n w_i x_i\right) = \sum_i^n \mathrm{Var}(w_i x_i)\\ &= \sum_i^n E[w_i]^2 \mathrm{Var}(x_i) + E[x_i]^2 \mathrm{Var}(w_i) + \mathrm{Var}(x_i)\mathrm{Var}(w_i)\end{aligned}$$

Variance calibration for linear layer

Assume linear activation and zero-mean weights and inputs. And number of inputs is n. Then,

$$
\begin{aligned}
\mathrm{Var}(y) &= \mathrm{Var}\left(\sum_i^n w_i x_i\right) = \sum_i^n \mathrm{Var}(w_i x_i) \\
&= \sum_i^n E[w_i]^2 \mathrm{Var}(x_i) + E[x_i]^2 \mathrm{Var}(w_i) + \mathrm{Var}(x_i)\mathrm{Var}(w_i) \\
&= \sum_i^n \mathrm{Var}(x_i)\mathrm{Var}(w_i) \\
&= (n\mathrm{Var}(w))\,\mathrm{Var}(x)
\end{aligned}
$$

Variance calibration for linear layer

Assume linear activation and zero-mean weights and inputs. And number of inputs is n. Then,

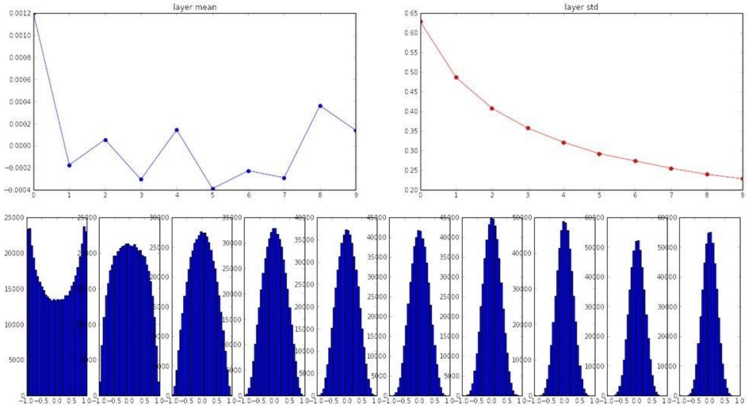
$$\begin{aligned}
\mathrm{Var}(y) = \mathrm{Var}\left(\sum_i^n w_i x_i\right) &= \sum_i^n \mathrm{Var}(w_i x_i) \\
&= \sum_i^n E[w_i]^2 \mathrm{Var}(x_i) + E[x_i]^2 \mathrm{Var}(w_i) + \mathrm{Var}(x_i)\mathrm{Var}(w_i) \\
&= \sum_i^n \mathrm{Var}(x_i)\mathrm{Var}(w_i) \\
&= (n\mathrm{Var}(w))\,\mathrm{Var}(x)
\end{aligned}$$

Thus, output will have same variance as input if $n\mathrm{Var}(w) = 1$

Weight initialization

```
input layer had mean 0.001800 and std 1.001311
hidden layer 1 had mean 0.001198 and std 0.627953
hidden layer 2 had mean -0.000175 and std 0.486051
hidden layer 3 had mean 0.000055 and std 0.407723
hidden layer 4 had mean -0.000306 and std 0.357108
hidden layer 5 had mean 0.000142 and std 0.320917
hidden layer 6 had mean -0.000389 and std 0.292116
hidden layer 7 had mean -0.000228 and std 0.273387
hidden layer 8 had mean -0.000291 and std 0.254935
hidden layer 9 had mean 0.000361 and std 0.239266
hidden layer 10 had mean 0.000139 and std 0.228008
```

```
W = np.random.randn(fan_in, fan_out) / np.sqrt(fan_in) # layer initialization
```



Reasonable initialization. (Mathematical derivation assumes linear activation

Xavier weight initialization

- By the same argument, if we want the variance of the backprop gradient does not change, we want $mVar(w) = 1$, where m is the number of outputs
- To account for both directions, one may initialize the weight with variance $\frac{2}{n+m}$
    - This is known as Xavier weight initialization
    - torch.nn.init.xavier_uniform_/torch.nn.init.xavier_normal_

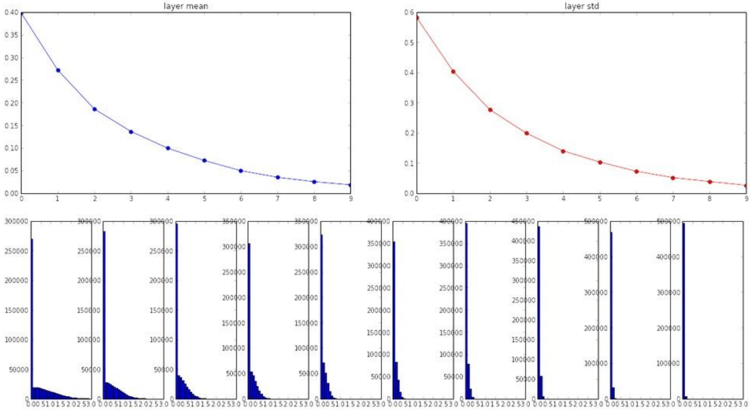
```
layer=torch.nn.Linear(10,20)
nn.init.xavier_normal_(layer.weight)

w=torch.empty(10,20)  # tensor without initialization
nn.init.xavier_normal_(w)
```

Weight initialization

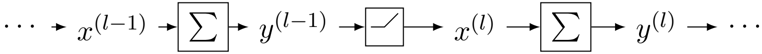
```
input layer had mean 0.000501 and std 0.999444
hidden layer 1 had mean 0.398623 and std 0.582273
hidden layer 2 had mean 0.272352 and std 0.403795
hidden layer 3 had mean 0.186076 and std 0.276912
hidden layer 4 had mean 0.136442 and std 0.198685
hidden layer 5 had mean 0.099568 and std 0.140299
hidden layer 6 had mean 0.072234 and std 0.103280
hidden layer 7 had mean 0.049775 and std 0.072748
hidden layer 8 had mean 0.035138 and std 0.051572
hidden layer 9 had mean 0.025404 and std 0.038583
hidden layer 10 had mean 0.018408 and std 0.026076
```

```
W = np.random.randn(fan_in, fan_out) / np.sqrt(fan_in) # layer initialization
```

but when using the ReLU nonlinearity it breaks.



Variance calibration for ReLU

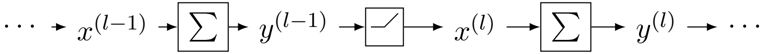


Note that it doesn't work when the activation layer is ReLU. But...[1]

$$\mathrm{Var}(y^{(l)}) = \mathrm{Var}\left(\sum_i^n w_i^{(l)} x_i^{(l)}\right)$$

[1]Note that $y^{(l)}$ now denotes the sum of input before going through the activation function.

Variance calibration for ReLU

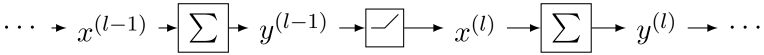


Note that it doesn't work when the activation layer is ReLU. But...[1]

$$\mathrm{Var}(y^{(l)}) = \mathrm{Var}\left(\sum_i^n w_i^{(l)} x_i^{(l)}\right) = \sum_i^n \mathrm{Var}(w_i^{(l)} x_i^{(l)}) = n\mathrm{Var}(w^{(l)} x^{(l)})$$

[1] Note that $y^{(l)}$ now denotes the sum of input before going through the activation function.

Variance calibration for ReLU

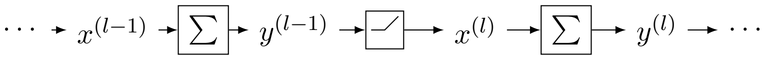


Note that it doesn't work when the activation layer is ReLU. But...[1]

$$\mathrm{Var}(y^{(l)}) = \mathrm{Var}\left(\sum_i^n w_i^{(l)} x_i^{(l)}\right) = \sum_i^n \mathrm{Var}(w_i^{(l)} x_i^{(l)}) = n\mathrm{Var}(w^{(l)} x^{(l)})$$
$$= nE[w^{(l)}]^2 \mathrm{Var}(x^{(l)}) + nE[x^{(l)}]^2 \mathrm{Var}(w^{(l)}) + \mathrm{nVar}(x^{(l)})\mathrm{Var}(w^{(l)})$$

[1]Note that $y^{(l)}$ now denotes the sum of input before going through the activation function.

Variance calibration for ReLU

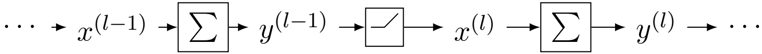


Note that it doesn't work when the activation layer is ReLU. But...[1]

$$
\begin{aligned}
\mathrm{Var}(y^{(l)}) &= \mathrm{Var}\left(\sum_i^n w_i^{(l)} x_i^{(l)}\right) = \sum_i^n \mathrm{Var}(w_i^{(l)} x_i^{(l)}) = n\mathrm{Var}(w^{(l)} x^{(l)}) \\
&= nE[w^{(l)}]^2 \mathrm{Var}(x^{(l)}) + nE[x^{(l)}]^2 \mathrm{Var}(w^{(l)}) + \mathrm{nVar}(x^{(l)})\mathrm{Var}(w^{(l)}) \\
&= nE[x^{(l)}]^2 \mathrm{Var}(w^{(l)}) + n\mathrm{Var}(x^{(l)})\mathrm{Var}(w^{(l)})
\end{aligned}
$$

[1] Note that $y^{(l)}$ now denotes the sum of input before going through the activation function.

Variance calibration for ReLU

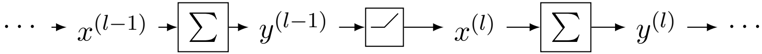


Note that it doesn't work when the activation layer is ReLU. But...[1]

$$
\begin{aligned}
\mathrm{Var}(y^{(l)}) &= \mathrm{Var}\left(\sum_i^n w_i^{(l)} x_i^{(l)}\right) = \sum_i^n \mathrm{Var}(w_i^{(l)} x_i^{(l)}) = n\mathrm{Var}(w^{(l)} x^{(l)}) \\
&= nE[w^{(l)}]^2 \mathrm{Var}(x^{(l)}) + nE[x^{(l)}]^2 \mathrm{Var}(w^{(l)}) + \mathrm{nVar}(x^{(l)})\mathrm{Var}(w^{(l)}) \\
&= nE[x^{(l)}]^2 \mathrm{Var}(w^{(l)}) + n\mathrm{Var}(x^{(l)})\mathrm{Var}(w^{(l)}) \\
&= nE[(x^{(l)})^2]\mathrm{Var}(w^{(l)})
\end{aligned}
$$

[1] Note that $y^{(l)}$ now denotes the sum of input before going through the activation function.

Variance calibration for ReLU

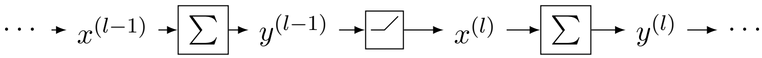


Note that it doesn't work when the activation layer is ReLU. But...[1]

$$
\begin{aligned}
\mathrm{Var}(y^{(l)}) = \mathrm{Var}\left(\sum_i^n w_i^{(l)} x_i^{(l)}\right) &= \sum_i^n \mathrm{Var}(w_i^{(l)} x_i^{(l)}) = n\mathrm{Var}(w^{(l)} x^{(l)}) \\
&= nE[w^{(l)}]^2 \mathrm{Var}(x^{(l)}) + nE[x^{(l)}]^2 \mathrm{Var}(w^{(l)}) + \mathrm{nVar}(x^{(l)})\mathrm{Var}(w^{(l)}) \\
&= nE[x^{(l)}]^2 \mathrm{Var}(w^{(l)}) + n\mathrm{Var}(x^{(l)})\mathrm{Var}(w^{(l)}) \\
&= nE[(x^{(l)})^2]\mathrm{Var}(w^{(l)}) \\
&= n(\mathrm{Var}(y^{(l-1)})/2)\mathrm{Var}(w^{(l)}) = \left(\frac{n}{2}\mathrm{Var}(w^{(l)})\right)\mathrm{Var}(y^{(l-1)})
\end{aligned}
$$

[1] Note that $y^{(l)}$ now denotes the sum of input before going through the activation function.

Variance calibration for ReLU



Note that it doesn't work when the activation layer is ReLU. But...[1]

$$\begin{aligned}
\text{Var}(y^{(l)}) &= \text{Var}\left(\sum_i^n w_i^{(l)} x_i^{(l)}\right) = \sum_i^n \text{Var}(w_i^{(l)} x_i^{(l)}) = n\text{Var}(w^{(l)} x^{(l)}) \\
&= nE[w^{(l)}]^2 \text{Var}(x^{(l)}) + nE[x^{(l)}]^2 \text{Var}(w^{(l)}) + \text{nVar}(x^{(l)})\text{Var}(w^{(l)}) \\
&= nE[x^{(l)}]^2 \text{Var}(w^{(l)}) + n\text{Var}(x^{(l)})\text{Var}(w^{(l)}) \\
&= nE[(x^{(l)})^2]\text{Var}(w^{(l)}) \\
&= n(\text{Var}(y^{(l-1)})/2)\text{Var}(w^{(l)}) = \left(\frac{n}{2}\text{Var}(w^{(l)})\right)\text{Var}(y^{(l-1)})
\end{aligned}$$

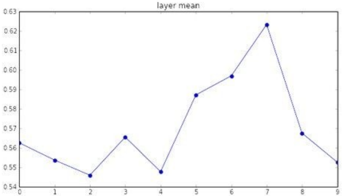
Variance of y conserved across a layer if $\frac{n}{2}\text{Var}(w) = 1$

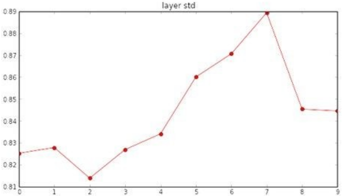
[1] Note that $y^{(l)}$ now denotes the sum of input before going through the activation function.

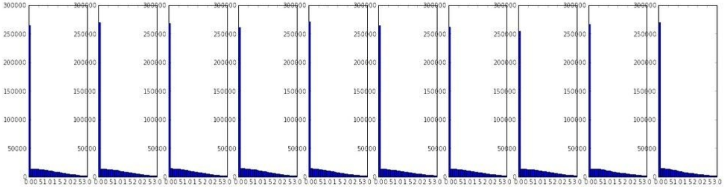
Weight initialization

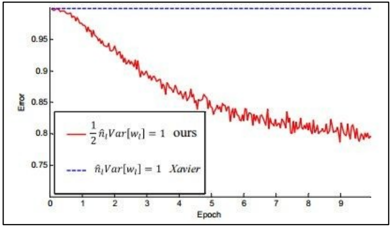
```
W = np.random.randn(fan_in, fan_out) / np.sqrt(fan_in/2) # layer initialization
```

```
input layer had mean 0.000501 and std 0.999444
hidden layer 1 had mean 0.562488 and std 0.825232
hidden layer 2 had mean 0.553614 and std 0.827835
hidden layer 3 had mean 0.545867 and std 0.813855
hidden layer 4 had mean 0.565396 and std 0.826902
hidden layer 5 had mean 0.547678 and std 0.834092
hidden layer 6 had mean 0.587103 and std 0.860035
hidden layer 7 had mean 0.596867 and std 0.870610
hidden layer 8 had mean 0.623214 and std 0.889348
hidden layer 9 had mean 0.567498 and std 0.845357
hidden layer 10 had mean 0.552531 and std 0.844523
```









Kaiming weight initialization

- The ReLU adjustment was first proposed by Kaiming He and his coauthors in an ICCV 2015 paper. The initialization method is adopted and popularized by ResNet
  - This is known as Kaiming weight initialization
  - Unlike Xavier initialization, only fan-in is considered $\Rightarrow Var(w) = \frac{2}{n}$
  - torch.nn.init.kaiming_uniform_/torch.nn.init.kaiming_normal_

```
layer=torch.nn.Linear(10,20)
nn.init.kaiming_normal_(layer.weight)
```

Batch normalization

Batch Normalization

[Ioffe and Szegedy, 2015]

"you want unit gaussian activations? just make them so."

consider a batch of activations at some layer.
To make each dimension unit gaussian, apply:

$$\widehat{x}^{(k)} = \frac{x^{(k)} - \mathrm{E}[x^{(k)}]}{\sqrt{\mathrm{Var}[x^{(k)}]}}$$

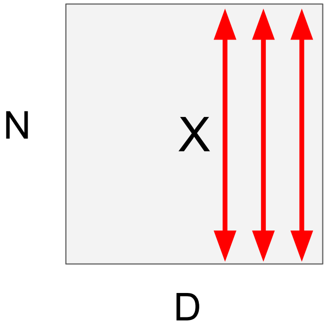
this is a vanilla differentiable function...

Batch normalization

Batch Normalization

[Ioffe and Szegedy, 2015]

“you want unit gaussian activations?
just make them so.”



1. compute the empirical mean and variance independently for each dimension.

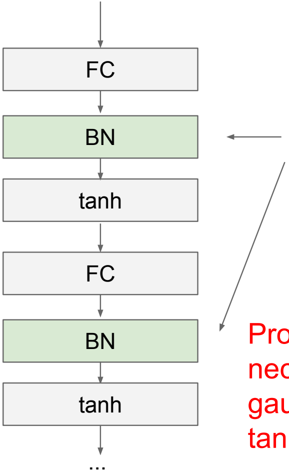
2. Normalize

$$\widehat{x}^{(k)} = \frac{x^{(k)} - \mathrm{E}[x^{(k)}]}{\sqrt{\mathrm{Var}[x^{(k)}]}}$$

Batch normalization

Batch Normalization

[Ioffe and Szegedy, 2015]



Usually inserted after Fully Connected / (or Convolutional, as we'll see soon) layers, and before nonlinearity.

Problem: do we necessarily want a unit gaussian input to a tanh layer?

$$\widehat{x}^{(k)} = \frac{x^{(k)} - \mathrm{E}[x^{(k)}]}{\sqrt{\mathrm{Var}[x^{(k)}]}}$$

Batch normalization

Batch Normalization

[Ioffe and Szegedy, 2015]

Normalize:

$$\widehat{x}^{(k)} = \frac{x^{(k)} - \mathrm{E}[x^{(k)}]}{\sqrt{\mathrm{Var}[x^{(k)}]}}$$

And then allow the network to squash the range if it wants to:

$$y^{(k)} = \gamma^{(k)}\widehat{x}^{(k)} + \beta^{(k)}$$

Note, the network can learn:

$$\gamma^{(k)} = \sqrt{\mathrm{Var}[x^{(k)}]}$$

$$\beta^{(k)} = \mathrm{E}[x^{(k)}]$$

to recover the identity mapping.

Batch normalization

Batch Normalization

[Ioffe and Szegedy, 2015]

Input: Values of x over a mini-batch: $\mathcal{B} = \{x_{1\ldots m}\}$;
Parameters to be learned: γ, β

Output: $\{y_i = \mathrm{BN}_{\gamma,\beta}(x_i)\}$

$$\mu_{\mathcal{B}} \leftarrow \frac{1}{m}\sum_{i=1}^{m} x_i \qquad \text{// mini-batch mean}$$

$$\sigma_{\mathcal{B}}^2 \leftarrow \frac{1}{m}\sum_{i=1}^{m} (x_i - \mu_{\mathcal{B}})^2 \qquad \text{// mini-batch variance}$$

$$\widehat{x}_i \leftarrow \frac{x_i - \mu_{\mathcal{B}}}{\sqrt{\sigma_{\mathcal{B}}^2 + \epsilon}} \qquad \text{// normalize}$$

$$y_i \leftarrow \gamma \widehat{x}_i + \beta \equiv \mathrm{BN}_{\gamma,\beta}(x_i) \qquad \text{// scale and shift}$$

- Improves gradient flow through the network
- Allows higher learning rates
- Reduces the strong dependence on initialization
- Acts as a form of regularization in a funny way, and slightly reduces the need for dropout, maybe

Batch normalization

Batch Normalization

[Ioffe and Szegedy, 2015]

Input: Values of x over a mini-batch: $\mathcal{B} = \{x_{1...m}\}$;
Parameters to be learned: γ, β

Output: $\{y_i = \text{BN}_{\gamma,\beta}(x_i)\}$

$$\mu_\mathcal{B} \leftarrow \frac{1}{m}\sum_{i=1}^{m} x_i \qquad \text{// mini-batch mean}$$

$$\sigma_\mathcal{B}^2 \leftarrow \frac{1}{m}\sum_{i=1}^{m}(x_i - \mu_\mathcal{B})^2 \qquad \text{// mini-batch variance}$$

$$\widehat{x}_i \leftarrow \frac{x_i - \mu_\mathcal{B}}{\sqrt{\sigma_\mathcal{B}^2 + \epsilon}} \qquad \text{// normalize}$$

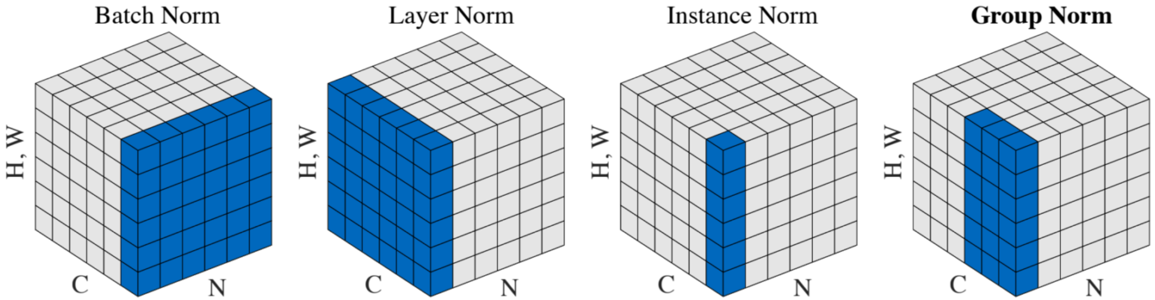
$$y_i \leftarrow \gamma\widehat{x}_i + \beta \equiv \text{BN}_{\gamma,\beta}(x_i) \qquad \text{// scale and shift}$$

Note: at test time BatchNorm layer functions differently:

The mean/std are not computed based on the batch. Instead, a single fixed empirical mean of activations during training is used.

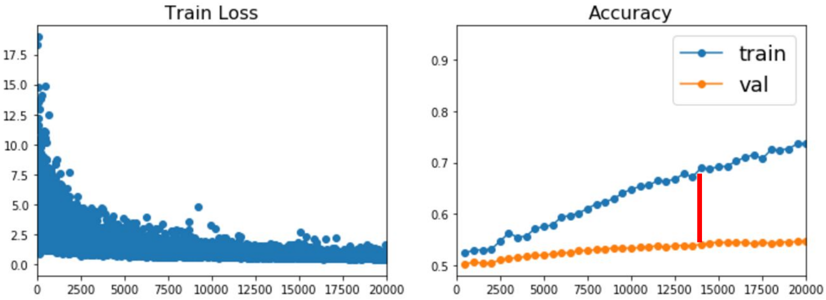
(e.g. can be estimated during training with running averages)

Other normalization techniques



Reducing testing error

How to improve single-model performance?



Ensemble trick

1. Train multiple independent models
2. At test time average their results

Enjoy 2% extra performance

Ensemble trick

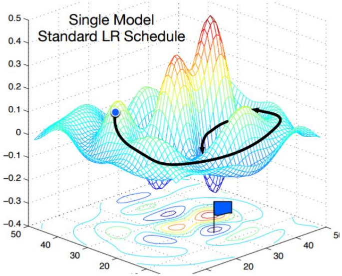
Fun Tips/Tricks:

- can also get a small boost from averaging multiple model checkpoints of a single model.

Ensemble trick

Model Ensembles: Tips and Tricks

Instead of training independent models, use multiple snapshots of a single model during training!

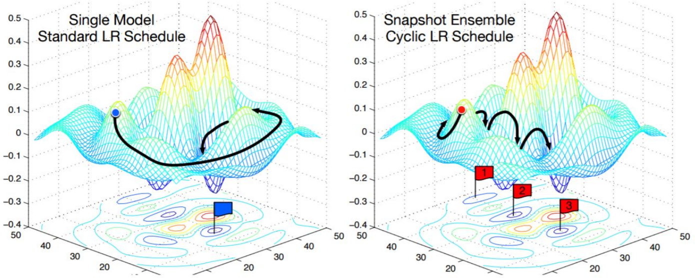


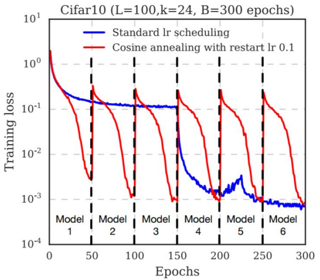
Loshchilov and Hutter, "SGDR: Stochastic gradient descent with restarts", arXiv 2016
Huang et al, "Snapshot ensembles: train 1, get M for free", ICLR 2017

Ensemble trick

Model Ensembles: Tips and Tricks

Instead of training independent models, use multiple snapshots of a single model during training!





Loshchilov and Hutter, "SGDR: Stochastic gradient descent with restarts", arXiv 2016
Huang et al, "Snapshot ensembles: train 1, get M for free", ICLR 2017
Figures copyright Yixuan Li and Geoff Pleiss, 2017. Reproduced with permission.

Cyclic learning rate schedules can make this work even better!

Ensemble trick

Model Ensembles: Tips and Tricks

Instead of using actual parameter vector, keep a moving average of the parameter vector and use that at test time (Polyak averaging)

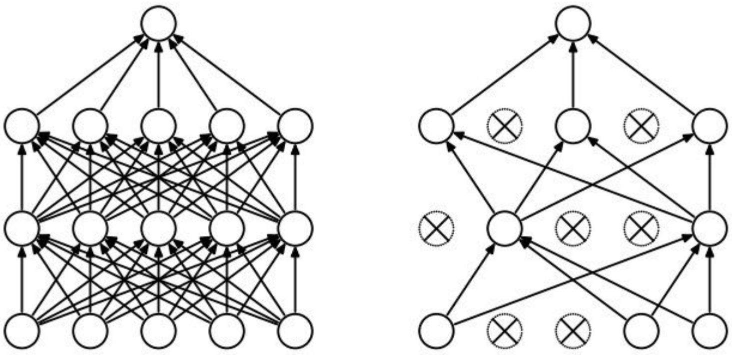
```
while True:
  data_batch = dataset.sample_data_batch()
  loss = network.forward(data_batch)
  dx = network.backward()
  x += - learning_rate * dx
  x_test = 0.995*x_test + 0.005*x # use for test set
```

Polyak and Juditsky, “Acceleration of stochastic approximation by averaging”, SIAM Journal on Control and Optimization, 1992.

Dropout

Regularization: **Dropout**

"randomly set some neurons to zero in the forward pass"



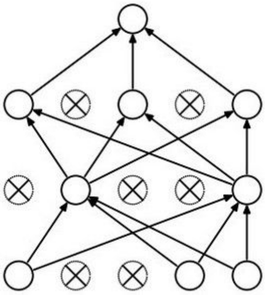
(a) Standard Neural Net (b) After applying dropout.

[Srivastava et al., 2014]

Dropout

```python
p = 0.5 # probability of keeping a unit active. higher = less dropout

def train_step(X):
  """ X contains the data """

  # forward pass for example 3-layer neural network
  H1 = np.maximum(0, np.dot(W1, X) + b1)
  U1 = np.random.rand(*H1.shape) < p # first dropout mask
  H1 *= U1 # drop!
  H2 = np.maximum(0, np.dot(W2, H1) + b2)
  U2 = np.random.rand(*H2.shape) < p # second dropout mask
  H2 *= U2 # drop!
  out = np.dot(W3, H2) + b3

  # backward pass: compute gradients... (not shown)
  # perform parameter update... (not shown)
```

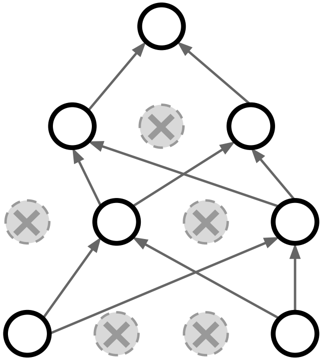
Example forward pass with a 3-layer network using dropout



Dropout

Regularization: Dropout

How can this possibly be a good idea?

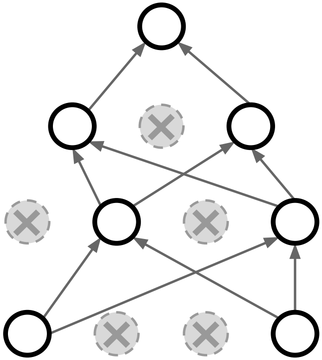


Forces the network to have a redundant representation;
Prevents co-adaptation of features

Dropout

Regularization: Dropout

How can this possibly be a good idea?



Another interpretation:

Dropout is training a large **ensemble** of models (that share parameters).

Each binary mask is one model

An FC layer with 4096 units has $2^{4096} \sim 10^{1233}$ possible masks!
Only $\sim 10^{82}$ atoms in the universe...

Dropout

Dropout: Test time

Dropout makes our output random!

Output (label) Input (image)

$$y = f_W(x, z)$$

Random mask

Want to "average out" the randomness at test-time

$$y = f(x) = E_z\left[f(x, z)\right] = \int p(z) f(x, z) dz$$

But this integral seems hard …

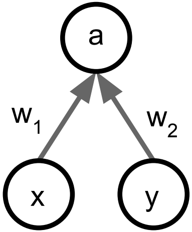
Dropout

Dropout: Test time

Want to approximate the integral

$$y = f(x) = E_z\left[f(x, z)\right] = \int p(z) f(x, z) dz$$

Consider a single neuron.

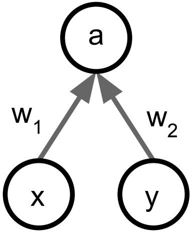


Dropout

Dropout: Test time

Want to approximate the integral

$$y = f(x) = E_z\big[f(x,z)\big] = \int p(z)f(x,z)dz$$



Consider a single neuron.

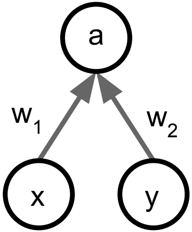
At test time we have: $E\big[a\big] = w_1x + w_2y$

Dropout

Dropout: Test time

Want to approximate the integral

$$y = f(x) = E_z\big[f(x,z)\big] = \int p(z) f(x,z)\,dz$$



Consider a single neuron.

At test time we have: $E\big[a\big] = w_1 x + w_2 y$

During training we have:

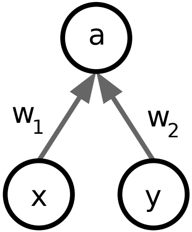
$$\begin{aligned} E\big[a\big] =& \frac{1}{4}(w_1x + w_2y) + \frac{1}{4}(w_1x + 0y) \\ &+ \frac{1}{4}(0x + 0y) + \frac{1}{4}(0x + w_2y) \\ =& \frac{1}{2}(w_1x + w_2y) \end{aligned}$$

Dropout

Dropout: Test time

Want to approximate the integral

$$y = f(x) = E_z\big[f(x,z)\big] = \int p(z) f(x,z) dz$$



Consider a single neuron.

At test time we have: $E\big[a\big] = w_1 x + w_2 y$

During training we have:

$$E\big[a\big] = \frac{1}{4}(w_1 x + w_2 y) + \frac{1}{4}(w_1 x + 0y) + \frac{1}{4}(0x + 0y) + \frac{1}{4}(0x + w_2 y) = \frac{1}{2}(w_1 x + w_2 y)$$

At test time, multiply by probability p

Dropout

Dropout: Test time

```
def predict(X):
  # ensembled forward pass
  H1 = np.maximum(0, np.dot(W1, X) + b1) * p # NOTE: scale the activations
  H2 = np.maximum(0, np.dot(W2, H1) + b2) * p # NOTE: scale the activations
  out = np.dot(W3, H2) + b3
```

At test time all neurons are active always
=> We must scale the activations so that for each neuron:
output at test time = expected output at training time

Dropout

Dropout Summary

```
""" Vanilla Dropout: Not recommended implementation (see notes below) """

p = 0.5 # probability of keeping a unit active. higher = less dropout

def train_step(X):
  """ X contains the data """

  # forward pass for example 3-layer neural network
  H1 = np.maximum(0, np.dot(W1, X) + b1)
  U1 = np.random.rand(*H1.shape) < p # first dropout mask
  H1 *= U1 # drop!
  H2 = np.maximum(0, np.dot(W2, H1) + b2)
  U2 = np.random.rand(*H2.shape) < p # second dropout mask
  H2 *= U2 # drop!
  out = np.dot(W3, H2) + b3

  # backward pass: compute gradients... (not shown)
  # perform parameter update... (not shown)

def predict(X):
  # ensembled forward pass
  H1 = np.maximum(0, np.dot(W1, X) + b1) * p # NOTE: scale the activations
  H2 = np.maximum(0, np.dot(W2, H1) + b2) * p # NOTE: scale the activations
  out = np.dot(W3, H2) + b3
```

drop in forward pass

scale at test time

Dropout

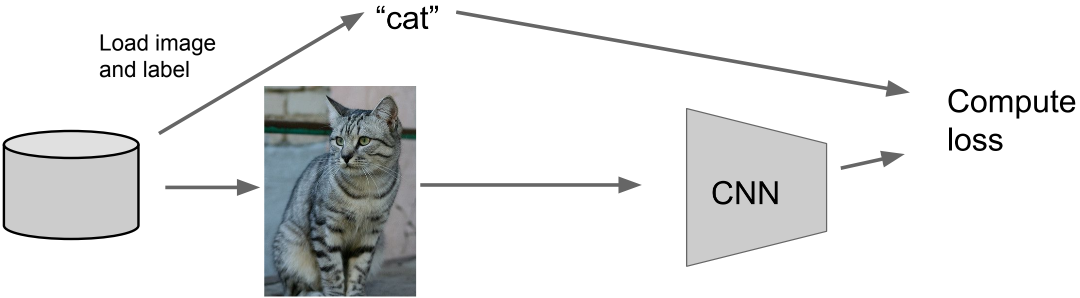
More common: “Inverted dropout”

```
p = 0.5 # probability of keeping a unit active. higher = less dropout

def train_step(X):
  # forward pass for example 3-layer neural network
  H1 = np.maximum(0, np.dot(W1, X) + b1)
  U1 = (np.random.rand(*H1.shape) < p) / p # first dropout mask. Notice /p!
  H1 *= U1 # drop!
  H2 = np.maximum(0, np.dot(W2, H1) + b2)
  U2 = (np.random.rand(*H2.shape) < p) / p # second dropout mask. Notice /p!
  H2 *= U2 # drop!
  out = np.dot(W3, H2) + b3

  # backward pass: compute gradients... (not shown)
  # perform parameter update... (not shown)

def predict(X):
  # ensembled forward pass
  H1 = np.maximum(0, np.dot(W1, X) + b1) # no scaling necessary
  H2 = np.maximum(0, np.dot(W2, H1) + b2)
  out = np.dot(W3, H2) + b3
```

test time is unchanged!

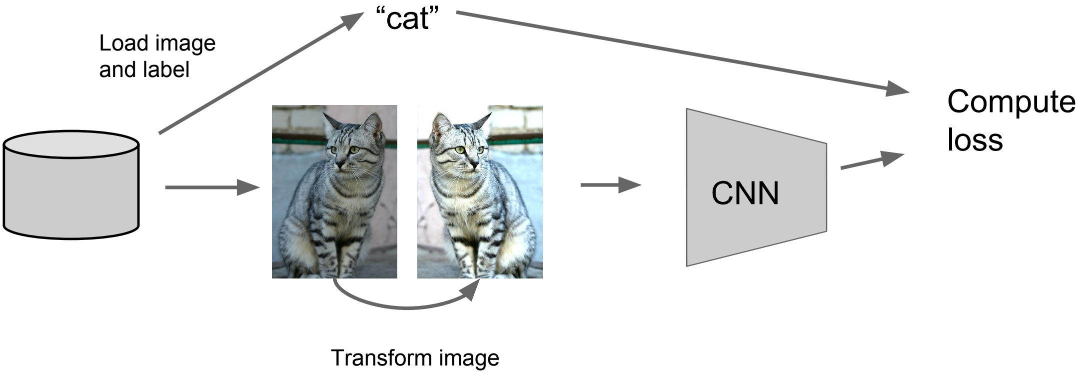
Data augmentation

Regularization: Data Augmentation



Data augmentation

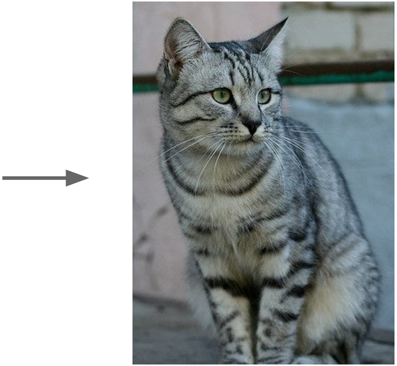
Regularization: Data Augmentation



Data augmentation

Data Augmentation

Horizontal Flips



Data augmentation

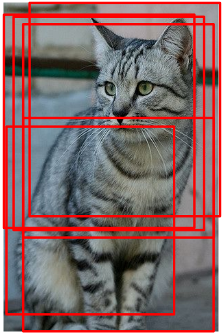
Data Augmentation

Random crops and scales

Training: sample random crops / scales

ResNet:

1. Pick random L in range [256, 480]
2. Resize training image, short side = L
3. Sample random 224 x 224 patch



Data augmentation

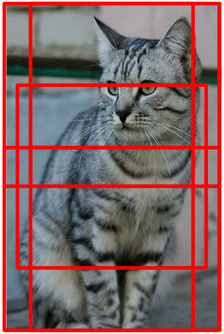
Data Augmentation

Random crops and scales

Training: sample random crops / scales

ResNet:

1. Pick random L in range [256, 480]
2. Resize training image, short side = L
3. Sample random 224 x 224 patch



Testing: average a fixed set of crops

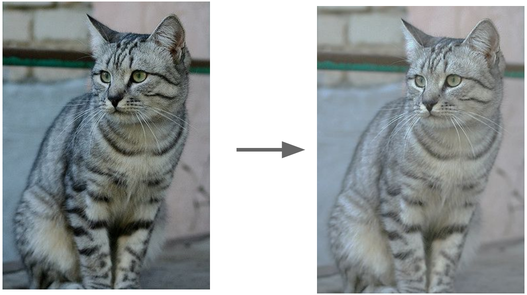
ResNet:

1. Resize image at 5 scales: {224, 256, 384, 480, 640}
2. For each size, use 10 224 x 224 crops: 4 corners + center, + flips

Data augmentation

Data Augmentation
Color Jitter

Simple: Randomize
contrast and brightness



Data augmentation

Data Augmentation
Color Jitter

Simple: Randomize contrast and brightness

More Complex:

1. Apply PCA to all [R, G, B] pixels in training set
2. Sample a "color offset" along principal component directions
3. Add offset to all pixels of a training image

(As seen in *[Krizhevsky et al. 2012]*, ResNet, etc)

Data augmentation

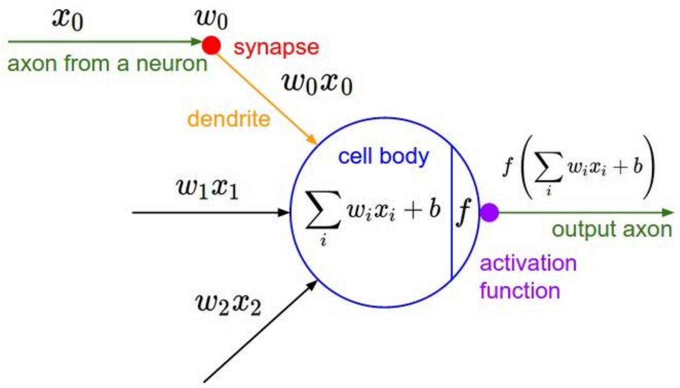
Data Augmentation

Get creative for your problem!

Random mix/combinations of :

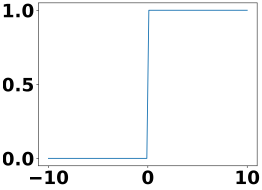
- translation
- rotation
- stretching
- shearing,
- lens distortions, … (go crazy)

Activation functions

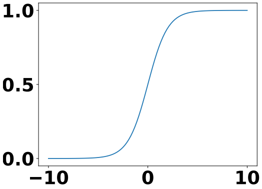


Threshold-based activation

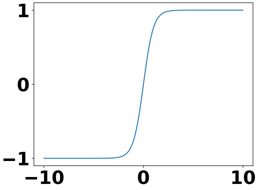
- Step function: earliest, used in perceptron



- Sigmoid (logistic) function: $\frac{1}{1+\exp(-x)}$



- Tanh: $\frac{\exp(x)-\exp(-x)}{\exp(x)+\exp(-x)}$



Threshold-based activation

- Historically very popular since they model well a saturated neuron

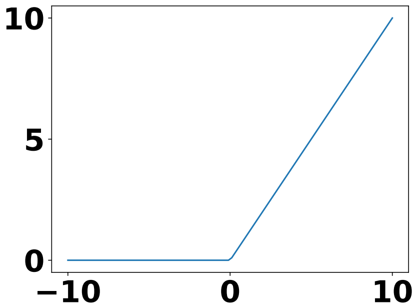
Threshold-based activation

- Historically very popular since they model well a saturated neuron
- However,
  - Saturated neurons lead to vanishing gradient
  - exp is a bit compute expensive
  - some concerns that sigmoid is not zero-centered (tanh solved the problem)

Threshold-based activation

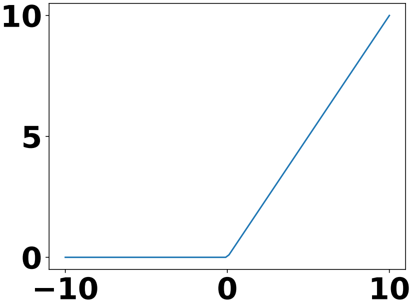
- Historically very popular since they model well a saturated neuron
- However,
  - Saturated neurons lead to vanishing gradient
  - exp is a bit compute expensive
  - some concerns that sigmoid is not zero-centered (tanh solved the problem)
- In most hidden layers, sigmoid and tanh should be avoided because of the gradient vanishing problem

ReLU



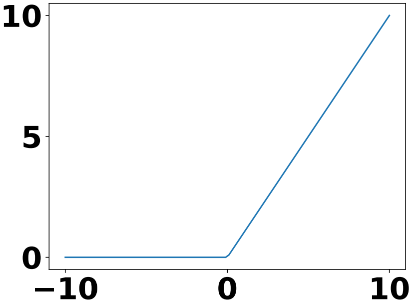
- Rectified linear unit: $f(x) = \max(x, 0)$
- Introduced by Nair and Hinton in 2010 and popularized by Alexnet in 2012

ReLU



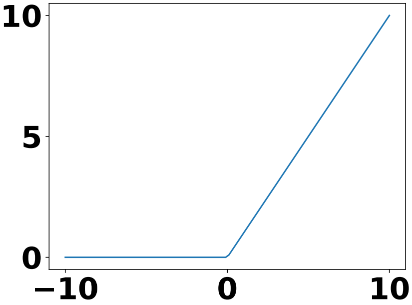
- Rectified linear unit: $f(x) = \max(x, 0)$
- Introduced by Nair and Hinton in 2010 and popularized by Alexnet in 2012
- Pros
  - No gradient vanishing problem
  - Computationally efficient
  - Converges much faster than sigmoid/tanh

ReLU



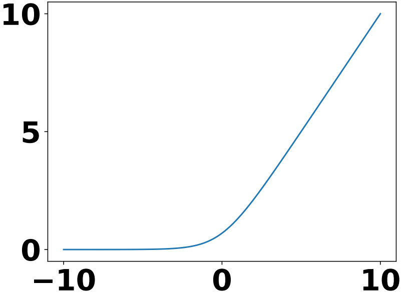
- Rectified linear unit: $f(x) = \max(x, 0)$
- Introduced by Nair and Hinton in 2010 and popularized by Alexnet in 2012
- Pros
  - No gradient vanishing problem
  - Computationally efficient
  - Converges much faster than sigmoid/tanh
- Cons
  - Not zero-centered and output always positive
  - Not differentiable at 0 (doesn't seem to be a problem in practice)

ReLU



- Rectified linear unit: $f(x) = \max(x, 0)$
- Introduced by Nair and Hinton in 2010 and popularized by Alexnet in 2012
- Pros
  - No gradient vanishing problem
  - Computationally efficient
  - Converges much faster than sigmoid/tanh
- Cons
  - Not zero-centered and output always positive
  - Not differentiable at 0 (doesn't seem to be a problem in practice)
- Bottom line, just use ReLU when in doubt

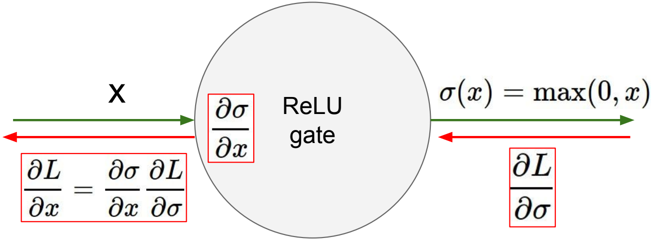
"Softplus"

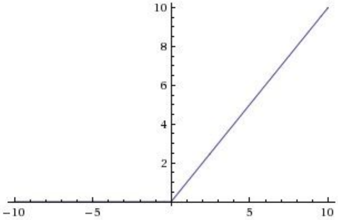


- $f(x) = \frac{1}{\beta} \log(1 + \exp(\beta x))$
- Act as a smooth version of ReLU
- In practice, it doesn't seem to work so well

  The use of softplus is generally discouraged. ... one might expect it to have advantage over the rectifier due to being differentiable everywhere or due to saturating less completely, but empirically it does not –Deep Learning book

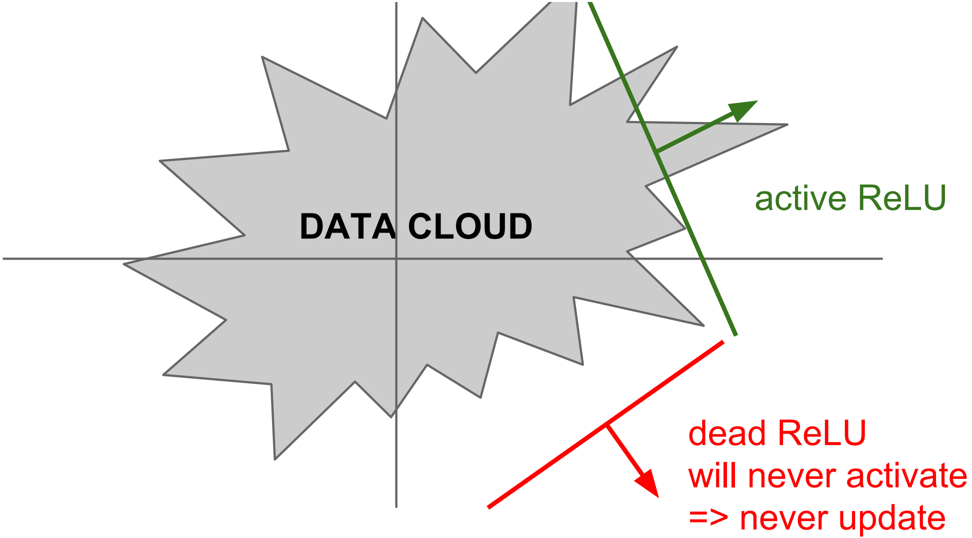
ReLU



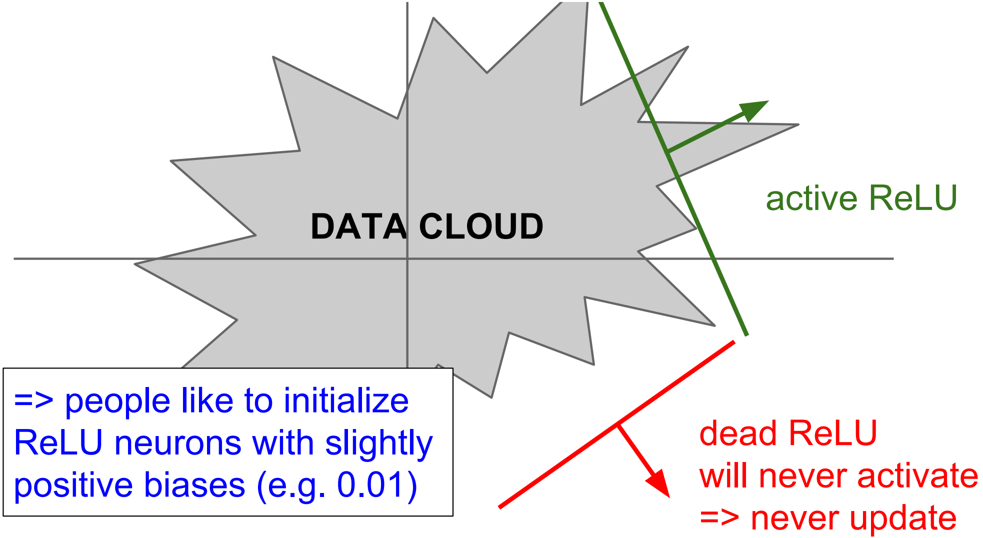


What happens when x = -10?
What happens when x = 0?
What happens when x = 10?

Dead ReLU neurons



Dead ReLU neurons



=> people like to initialize ReLU neurons with slightly positive biases (e.g. 0.01)

Sparsity of ReLU

Ian Goodfellow

Lead author of the Deep Learning textbook: http://www.deeplearningbook.org · Upvoted by Rahul Bohare, M.S. Machine Learning & Computer Vision, Technical University of Munich (2019) and Viresh Ranjan, PhD Student in Machine LearningAuthor has **212** answers and **3.4M** answer views ...

Related **Where is Sparsity important in Deep Learning?**

The main thing that's important is sparsity of *connections*: each unit should usually be connected to relatively few other units. In the human brain, estimates of the number of neurons vary, but it something like 1e10–1e11 neurons. Each neuron is only connected to about 1e4 other neurons on average though. In machine learning, we see this in convolutional networks. Each neuron receives input only from a very small patch in the layer below.

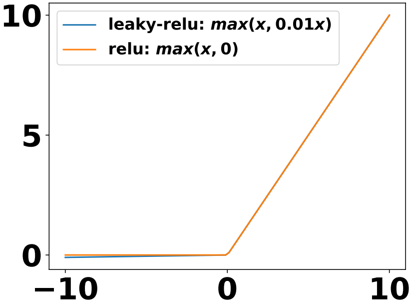
Sparsity of connections can be seen as resembling sparsity of weights, because it's equivalent (in terms of the function it represents) to having a fully connected network with zero weights in most places. Sparsity of connections is better though, because you don't pay the computational cost of explicitly multiplying each input by zero and adding up all those zeros.

So far, learning weights that are sparse hasn't really paid off, at least not in the context of neural nets. Statisticians often learn sparse models in order to understand which variables are most important, but that's an analysis technique, not a strategy for making better predictions.

Learning activations that are sparse doesn't really seem to matter either. Five years ago, people thought that part of why relus worked well was that they were sparse, but it turns out that all that matters is that they are piecewise linear. Maxout can beat relus in some contexts and performs about the same as relus in other contexts, and it's not sparse at all: http://jmlr.org/proceedings/papers/v28/goodfellow13.pdf

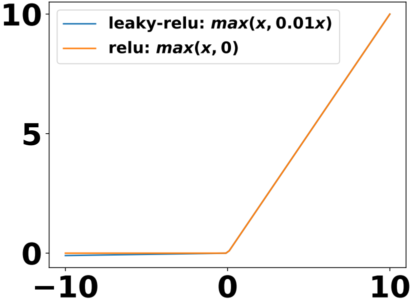
- Theoretically ReLU promotes sparsity
  - many zeros in trained model
- But it is controversial if that is a dominant factor

Leaky ReLU



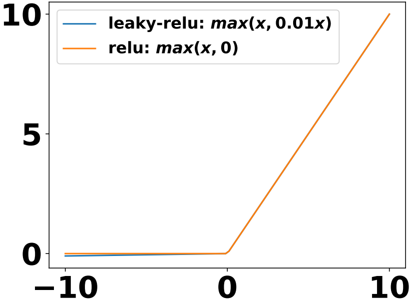
- $f(x) = \max(x, 0.01x)$

Leaky ReLU



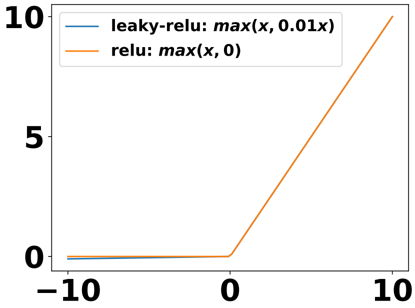
- $f(x) = \max(x, 0.01x)$
- Does not saturate

Leaky ReLU



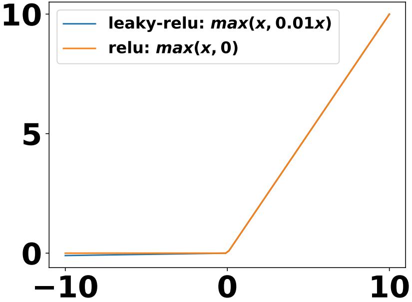
- $f(x) = \max(x, 0.01x)$
- Does not saturate
- Computationally efficient

Leaky ReLU



- $f(x) = \max(x, 0.01x)$
- Does not saturate
- Computationally efficient
- Seem to work better than ReLU (see experiments here and here)

Leaky ReLU



- $f(x) = \max(x, 0.01x)$
- Does not saturate
- Computationally efficient
- Seem to work better than ReLU (see experiments here and here)
- Generalize to Parametric Rectifier (PReLU)
  - Replace 0.01 with a learnable α. i.e., $f(x) = \max(x, \alpha x)$

Maxout [Goodfellow et al., 2013]

- Try to generalize ReLU and leaky ReLU

$$\max(\mathbf{w}_1^T\mathbf{x} + b_1, \mathbf{w}_2^T\mathbf{x} + b_2)$$

Maxout [Goodfellow et al., 2013]

- Try to generalize ReLU and leaky ReLU

$$\max(\mathbf{w}_1^T\mathbf{x} + b_1, \mathbf{w}_2^T\mathbf{x} + b_2)$$

Pros

- Linear regime
- Does not saturate
- Does not die

Maxout [Goodfellow et al., 2013]

- Try to generalize ReLU and leaky ReLU

$$\max(\mathbf{w}_1^T\mathbf{x} + b_1, \mathbf{w}_2^T\mathbf{x} + b_2)$$

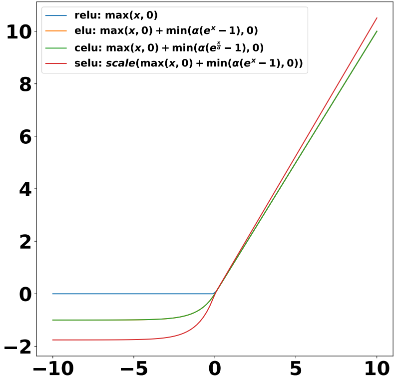
Pros

- Linear regime
- Does not saturate
- Does not die

Cons

- Double amount of parameters

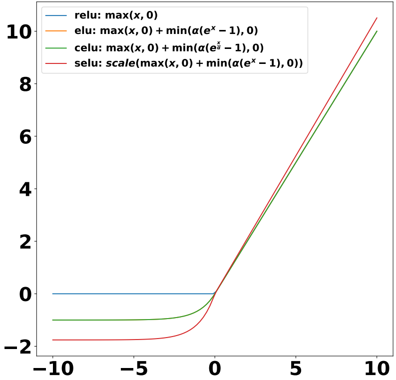
ELU



- Exponential linear unit:

$$\mathrm{ELU}(x,\alpha)=\begin{cases} x & \text{if } x>0 \\ \alpha(e^x-1) & \text{otherwise}\end{cases}$$

ELU

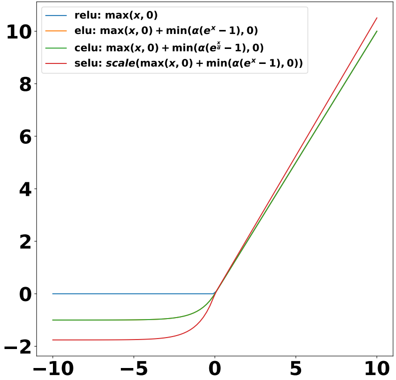


- Exponential linear unit:

$$\mathrm{ELU}(x,\alpha) = \begin{cases} x & \text{if } x > 0 \\ \alpha(e^x - 1) & \text{otherwise} \end{cases}$$

- Closer to zero mean

ELU

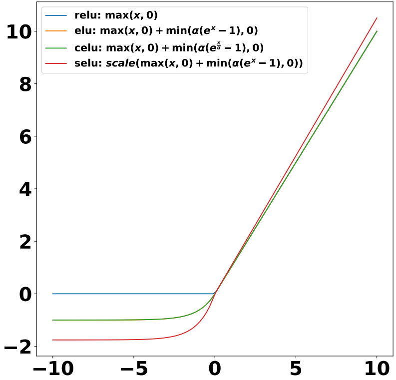


- Exponential linear unit:

$$\mathrm{ELU}(x,\alpha) = \begin{cases} x & \text{if } x > 0 \\ \alpha(e^x - 1) & \text{otherwise} \end{cases}$$

- Closer to zero mean
- Work better than ReLU according to this

ELU

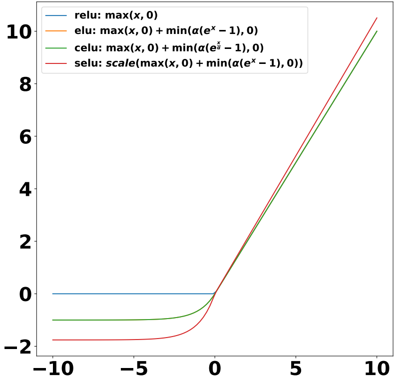


- Exponential linear unit:

$$\mathrm{ELU}(x,\alpha) = \begin{cases} x & \text{if } x > 0 \\ \alpha(e^x - 1) & \text{otherwise} \end{cases}$$

- Closer to zero mean
- Work better than ReLU according to this
- CELU: $f(x) = \begin{cases} x & \text{if } x > 0 \\ \alpha(e^{x/\alpha} - 1) & \text{otherwise} \end{cases}$
  - $x \to x/\alpha$ to make function differentiable at 0

ELU



- Exponential linear unit:

$$\mathrm{ELU}(x,\alpha)=\begin{cases} x & \text{if } x>0 \\ \alpha(e^x-1) & \text{otherwise}\end{cases}$$

- Closer to zero mean
- Work better than ReLU according to this
- CELU: $f(x)=\begin{cases} x & \text{if } x>0 \\ \alpha(e^{x/\alpha}-1) & \text{otherwise}\end{cases}$
    - $x \to x/\alpha$ to make function differentiable at 0
- SELU: Adjust α and add scale to make function self-normalize (zero-mean, unit variance input$\Rightarrow$ zero-mean, unit variance output)
    - $\mathrm{SELU}(x)=\lambda \mathrm{ELU}(x,\alpha))$
    - $\lambda \approx 1.0507$, $\alpha \approx 1.6733$

Gaussian ELU (often known as GeLU)

- GeLU is motivated by dropout. The authors like to drop some of the node randomly based on the input

Gaussian ELU (often known as GeLU)

- GeLU is motivated by dropout. The authors like to drop some of the node randomly based on the input
- Nodes are still randomly dropped if we consider input as stochastic
- But the actual operation is deterministic w.r.t. the input

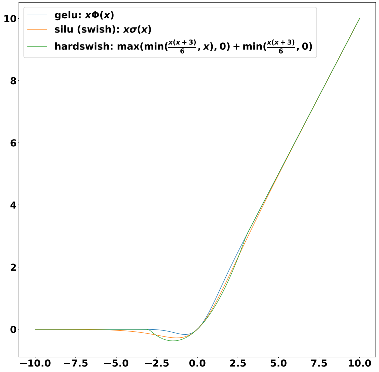
Gaussian ELU (often known as GeLU)

- GeLU is motivated by dropout. The authors like to drop some of the node randomly based on the input
- Nodes are still randomly dropped if we consider input as stochastic
- But the actual operation is deterministic w.r.t. the input
- They choose an activation function $GeLU(x) = x\Phi(x) \approx x\sigma(1.702x)$, where $\Phi(x) = Pr(Z < x)$ for $Z \sim N(0, 1)$ is the cdf of Z

Gaussian ELU (often known as GeLU)

- GeLU is motivated by dropout. The authors like to drop some of the node randomly based on the input
- Nodes are still randomly dropped if we consider input as stochastic
- But the actual operation is deterministic w.r.t. the input
- They choose an activation function $GeLU(x) = x\Phi(x) \approx x\sigma(1.702x)$, where $\Phi(x) = Pr(Z < x)$ for $Z \sim N(0,1)$ is the cdf of Z
- Quite widely adopted by OpenAI and used in Transformers

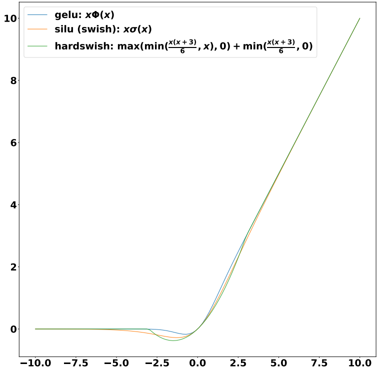
Swish and Hardswish



Baselines	ReLU	LReLU	PReLU	Softplus	ELU	SELU
Swish > Baseline	9	8	6	7	8	8
Swish = Baseline	0	1	3	1	0	1
Swish < Baseline	0	0	0	1	1	0

- Swish: $f(x) = x\sigma(\beta x)$
  - β is a learnable parameter
  - When β is fixed to 1, it is equal to SiLU
    - Often SiLU rather than Swish is implemented
  - Converge to ReLU when $\beta \to \infty$

Swish and Hardswish



Baselines	ReLU	LReLU	PReLU	Softplus	ELU	SELU
Swish > Baseline	9	8	6	7	8	8
Swish = Baseline	0	1	3	1	0	1
Swish < Baseline	0	0	0	1	1	0

- Swish: $f(x) = x\sigma(\beta x)$
  - β is a learnable parameter
  - When β is fixed to 1, it is equal to SiLU
    - Often SiLU rather than Swish is implemented
  - Converge to ReLU when $\beta \to \infty$
- Hardswish: $f(x) = \begin{cases} 0 & \text{if } x \leq -3, \\ x & \text{if } x \geq +3, \\ x \cdot (x+3)/6 & \text{otherwise} \end{cases}$
  - Piecewise approximation of Swish
  - Use in MobileNet V3

GLU and variants

- Gated Linear Unit: $\mathrm{GLU}(x, W, V, b, c) = \sigma(xW + b) \otimes (xV + c)$

GLU and variants

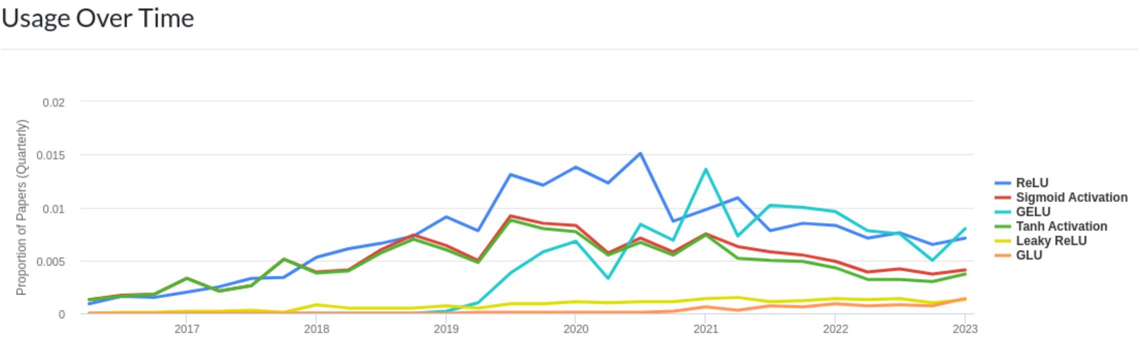
- Gated Linear Unit: $\mathrm{GLU}(x, W, V, b, c) = \sigma(xW + b) \otimes (xV + c)$
- Introduced by Dauphin et al. and inspired by a bilinear unit by Mnih and Hinton: $\mathrm{Bilinear}(x, W.V.b.c) = (xW + b) \otimes (xV + c)$

GLU and variants

- Gated Linear Unit: $\text{GLU}(x, W, V, b, c) = \sigma(xW + b) \otimes (xV + c)$
- Introduced by Dauphin et al. and inspired by a bilinear unit by Mnih and Hinton: $\text{Bilinear}(x, W.V.b.c) = (xW + b) \otimes (xV + c)$
- Other variants introduced in Shazeer

$$\text{ReGLU}(x, W, V, b, c) = \max(0, xW + b) \otimes (xV + c)$$
$$\text{GEGLU}(x, W, V, b, c) = \text{GELU}(xW + b) \otimes (xV + c)$$
$$\text{SwiGLU}(x, W, V, b, c, \beta) = \text{Swish}_\beta(xW + b) \otimes (xV + c)$$

Trend (from paperswithcode)



Summary (IMHO)

- Still a hot topic and nothing is final

Summary (IMHO)

- Still a hot topic and nothing is final
- Vanishing gradient seems to be a bigger problem than exploding gradient
  - ReLU > Sigmoid/tanh

Summary (IMHO)

- Still a hot topic and nothing is final
- Vanishing gradient seems to be a bigger problem than exploding gradient
    - ReLU > Sigmoid/tanh
- Zero-mean is not essential. But all positive does hurt performance
    - ReLU/Softplus < ELU, Leaky-ReLU, etc.

Summary (IMHO)

- Still a hot topic and nothing is final
- Vanishing gradient seems to be a bigger problem than exploding gradient
  - ReLU > Sigmoid/tanh
- Zero-mean is not essential. But all positive does hurt performance
  - ReLU/Softplus < ELU, Leaky-ReLU, etc.
- Sparsity may have a role after all (just my guess)
  - Softplus < ReLU
  - ELU, Leaky-ReLU < Swish, GELU

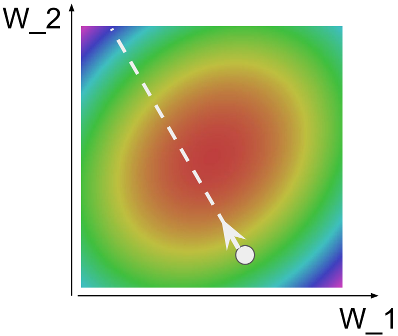
Summary (IMHO)

- Still a hot topic and nothing is final
- Vanishing gradient seems to be a bigger problem than exploding gradient
  - ReLU > Sigmoid/tanh
- Zero-mean is not essential. But all positive does hurt performance
  - ReLU/Softplus < ELU, Leaky-ReLU, etc.
- Sparsity may have a role after all (just my guess)
  - Softplus < ReLU
  - ELU, Leaky-ReLU < Swish, GELU
- When in doubt, just use ReLU and it is usually good enough
  - Can try out GeLU/Swish if complexity is not a huge concern

Optimizers

Optimization

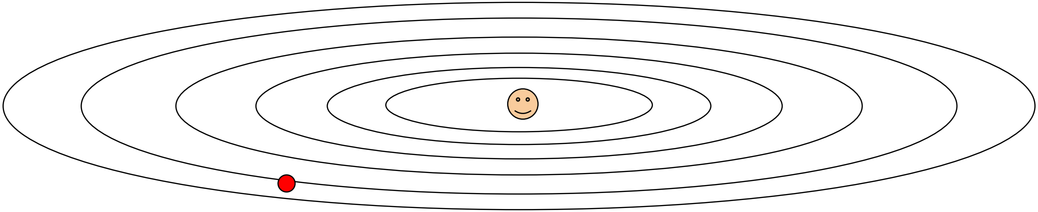
```
# Vanilla Gradient Descent

while True:
  weights_grad = evaluate_gradient(loss_fun, data, weights)
  weights += - step_size * weights_grad # perform parameter update
```



Optimizers

Optimization: Problems with SGD

What if loss changes quickly in one direction and slowly in another?
What does gradient descent do?

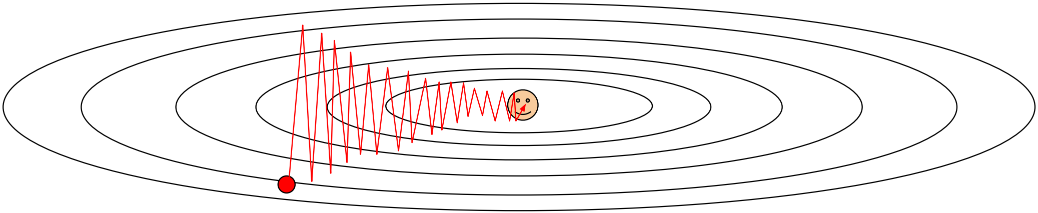


Loss function has high **condition number**: ratio of largest to smallest singular value of the Hessian matrix is large

Optimizers

Optimization: Problems with SGD

What if loss changes quickly in one direction and slowly in another?
What does gradient descent do?
Very slow progress along shallow dimension, jitter along steep direction

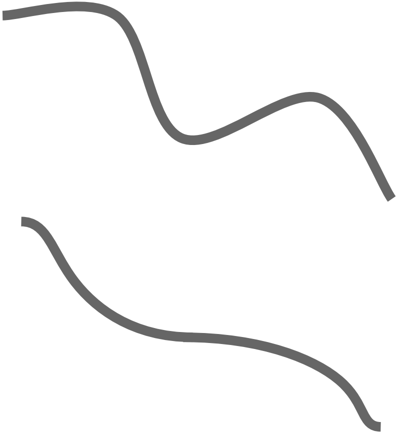


Loss function has high **condition number**: ratio of largest to smallest singular value of the Hessian matrix is large

Optimizers

Optimization: Problems with SGD

What if the loss function has a **local minima** or **saddle point**?

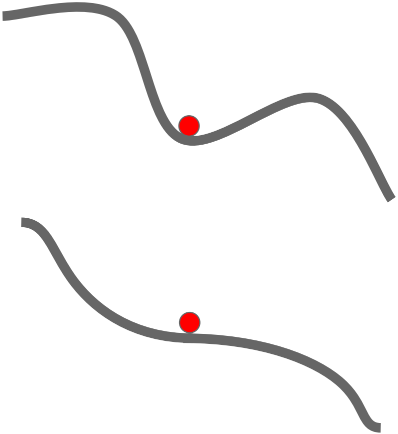


Optimizers

Optimization: Problems with SGD

What if the loss function has a **local minima** or **saddle point**?

Zero gradient, gradient descent gets stuck

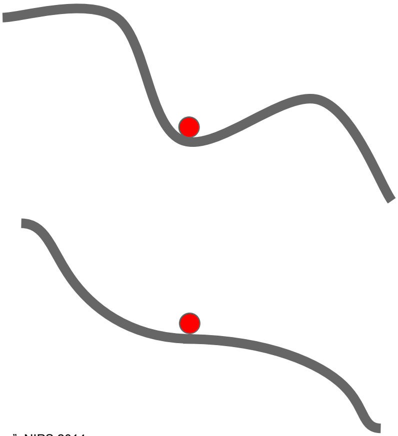


Optimizers

Optimization: Problems with SGD

What if the loss function has a **local minima** or **saddle point**?

Saddle points much more common in high dimension



Dauphin et al, “Identifying and attacking the saddle point problem in high-dimensional non-convex optimization”, NIPS 2014

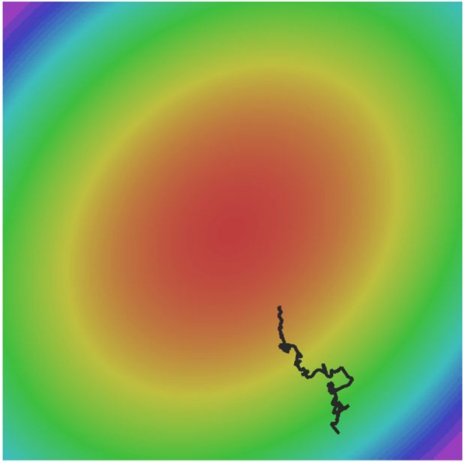
Optimizers

Optimization: Problems with SGD

Our gradients come from minibatches so they can be noisy!

$$L(W) = \frac{1}{N} \sum_{i=1}^{N} L_i(x_i, y_i, W)$$

$$\nabla_W L(W) = \frac{1}{N} \sum_{i=1}^{N} \nabla_W L_i(x_i, y_i, W)$$



Exponential moving average

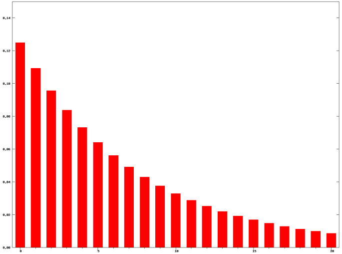
- $S_t = \begin{cases} Y_1, & t = 1 \\ \alpha \cdot Y_t + (1-\alpha) \cdot S_{t-1}, & t > 1 \end{cases}$

Exponential moving average

- $S_t = \begin{cases} Y_1, & t = 1 \\ \alpha \cdot Y_t + (1-\alpha) \cdot S_{t-1}, & t > 1 \end{cases}$
- $S_t = \alpha \left[Y_{t-1} + (1-\alpha) Y_{t-2} + (1-\alpha)^2 Y_{t-3} + \cdots \right]$

Exponential moving average

- $S_t = \begin{cases} Y_1, & t = 1 \\ \alpha \cdot Y_t + (1-\alpha)\cdot S_{t-1}, & t > 1 \end{cases}$
- $S_t = \alpha \left[Y_{t-1} + (1-\alpha) Y_{t-2} + (1-\alpha)^2 Y_{t-3} + \cdots \right] = \frac{Y_{t-1} + (1-\alpha) Y_{t-2} + (1-\alpha)^2 Y_{t-3} + \cdots}{1 + (1-\alpha) + (1-\alpha)^2 + \cdots}$



Momentum update

Sutskever et al.:

$$\Delta \mathbf{x} \leftarrow \mu \Delta \mathbf{x} - \mathrm{lr}(1-\mu)\nabla_{\mathbf{x}} L$$

$$\mathbf{x} \leftarrow \mathbf{x} + \Delta \mathbf{x}$$

$$\mu \in [0,1), \mu = 0 \Rightarrow \text{No momentum}$$

Alternative:

$$\Delta \mathbf{x} \leftarrow \mu \Delta \mathbf{x} + (1-\mu)\nabla_{\mathbf{x}} L$$

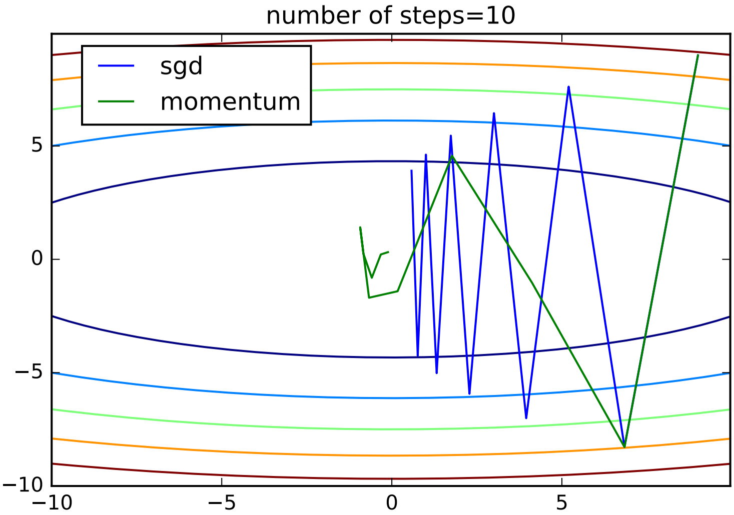
$$\mathbf{x} \leftarrow \mathbf{x} - \mathrm{lr} \cdot \Delta \mathbf{x}$$

$$\mu \in [0,1), \mu = 0 \Rightarrow \text{No momentum}$$

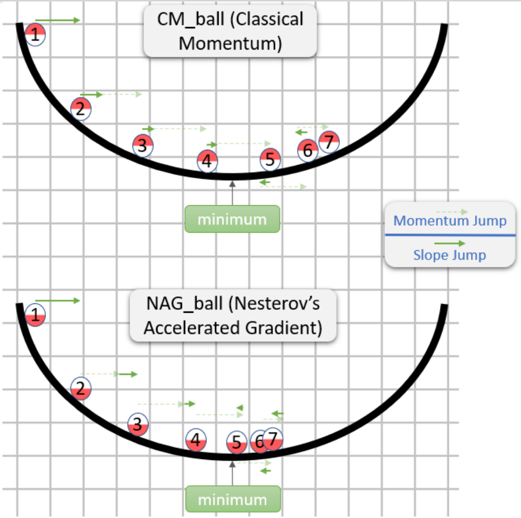
- μ often takes values such as 0.5, 0.9, and 0.99. And can annealed over time as well
- Allows "velocity" to build up along shallow directions
- Velocity becomes damped in steep valley with rapid change of gradient sign

Remark: In PyTorch, $\Delta \mathbf{x} \leftarrow \mu \Delta \mathbf{x} + \nabla_{\mathbf{x}} L$ is implemented instead of the one shown on the right. It saves one mulitiplication operation, but note that lr is effectively $\frac{1}{1-\mu}$ times larger

Momentum update vs SGD

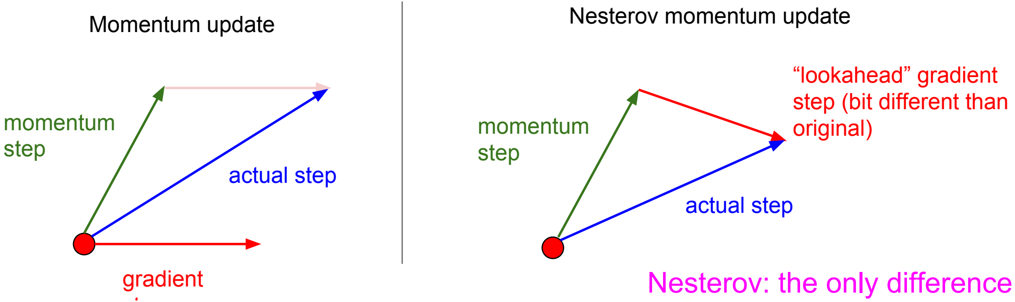


NAG



Reference:
https://stats.stackexchange.com/questions/179915/whats-the-difference-between-momentum-based-gradient-descent-and-nesterovs-acc

NAG



Nesterov: the only difference...

$$v_t = \mu v_{t-1} - \underbrace{(1-\mu) lr}_{\epsilon} \nabla f(x_{t-1} + \mu v_{t-1})$$

$$x_t = x_{t-1} + v_t$$

We want to deal with $\nabla f(x_{t-1})$ instead

NAG

In many cases such as backprop, we only have gradient for the current x. However, NAG can be "fixed" as follows

$$
\begin{aligned}
v_t &= \mu v_{t-1} - \epsilon \nabla f(x_{t-1} + \mu v_{t-1}) \\
x_t &= x_{t-1} + v_t
\end{aligned}
$$

NAG

In many cases such as backprop, we only have gradient for the current x. However, NAG can be "fixed" as follows

$$v_t = \mu v_{t-1} - \epsilon \nabla f(x_{t-1} + \mu v_{t-1})$$
$$x_t = x_{t-1} + v_t$$

Pick $\tilde{x}_t = x_t + \mu v_t$,

$$v_t = \mu v_{t-1} - \epsilon \nabla f(\tilde{x}_{t-1})$$

NAG

In many cases such as backprop, we only have gradient for the current x. However, NAG can be "fixed" as follows

$$v_t = \mu v_{t-1} - \epsilon \nabla f(x_{t-1} + \mu v_{t-1})$$
$$x_t = x_{t-1} + v_t$$

Pick $\tilde{x}_t = x_t + \mu v_t$,

$$v_t = \mu v_{t-1} - \epsilon \nabla f(\tilde{x}_{t-1})$$
$$\tilde{x}_t = x_t + \mu v_t = x_{t-1} + v_t + \mu v_t$$

NAG

In many cases such as backprop, we only have gradient for the current x. However, NAG can be "fixed" as follows

$$v_t = \mu v_{t-1} - \epsilon \nabla f(x_{t-1} + \mu v_{t-1})$$
$$x_t = x_{t-1} + v_t$$

Pick $\tilde{x}_t = x_t + \mu v_t$,

$$v_t = \mu v_{t-1} - \epsilon \nabla f(\tilde{x}_{t-1})$$
$$\tilde{x}_t = x_t + \mu v_t = x_{t-1} + v_t + \mu v_t$$
$$= \tilde{x}_{t-1} - \mu v_{t-1} + v_t + \mu v_t$$

NAG

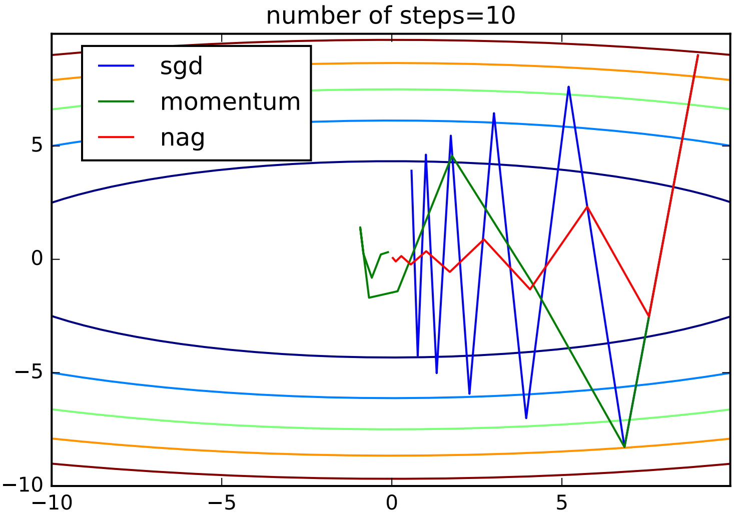
In many cases such as backprop, we only have gradient for the current x. However, NAG can be "fixed" as follows

$$
\begin{aligned}
v_t &= \mu v_{t-1} - \epsilon \nabla f(x_{t-1} + \mu v_{t-1}) \\
x_t &= x_{t-1} + v_t
\end{aligned}
$$

Pick $\tilde{x}_t = x_t + \mu v_t$,

$$
\begin{aligned}
v_t &= \mu v_{t-1} - \epsilon \nabla f(\tilde{x}_{t-1}) \\
\tilde{x}_t &= x_t + \mu v_t = x_{t-1} + v_t + \mu v_t \\
&= \tilde{x}_{t-1} - \mu v_{t-1} + v_t + \mu v_t \\
&= \tilde{x}_{t-1} + v_t + \mu(v_t - v_{t-1})
\end{aligned}
$$

Optimizers



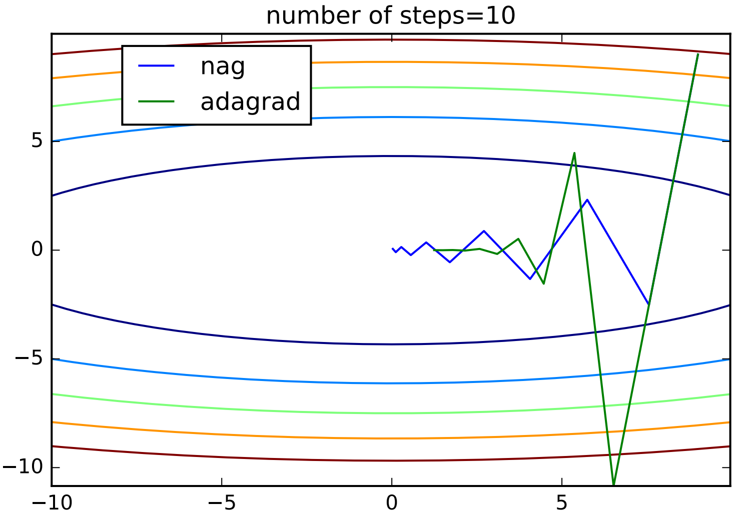
AdaGrad

```
# Adagrad update
cache += dx**2
x += - learning_rate * dx / (np.sqrt(cache) + 1e-7)
```

Added element-wise scaling of the gradient based on the historical sum of squares in each dimension

The idea is to penalize direction that has already have explored a lot (with large cumulative partial derivative)

Optimizers



RMSProp

RMSProp update

[Tieleman and Hinton, 2012]

```
# Adagrad update
cache += dx**2
x += - learning_rate * dx / (np.sqrt(cache) + 1e-7)
```

↓

```
# RMSProp
cache = decay_rate * cache + (1 - decay_rate) * dx**2
x += - learning_rate * dx / (np.sqrt(cache) + 1e-7)
```

RMSProp

rmsprop: A mini-batch version of rprop

- rprop is equivalent to using the gradient but also dividing by the size of the gradient.
  - The problem with mini-batch rprop is that we divide by a different number for each mini-batch. So why not force the number we divide by to be very similar for adjacent mini-batches?
- rmsprop: Keep a moving average of the squared gradient for each weight

$$MeanSquare(w,t) = 0.9\ MeanSquare(w,\ t-1) + 0.1\left(\partial E/\partial w\,(t)\right)^2$$

- Dividing the gradient by $\sqrt{MeanSquare(w,\ t)}$ makes the learning work much better (Tijmen Tieleman, unpublished).

Introduced in a slide in Geoff Hinton's Coursera class, lecture 6

RMSProp

rmsprop: A mini-batch version of rprop

- rprop is equivalent to using the gradient but also dividing by the size of the gradient.
  - The problem with mini-batch rprop is that we divide by a different number for each mini-batch. So why not force the number we divide by to be very similar for adjacent mini-batches?
- rmsprop: Keep a moving average of the squared gradient for each weight

$$MeanSquare(w,t) = 0.9\ MeanSquare(w,\ t-1) + 0.1\left(\partial E / \partial w\,(t)\right)^2$$

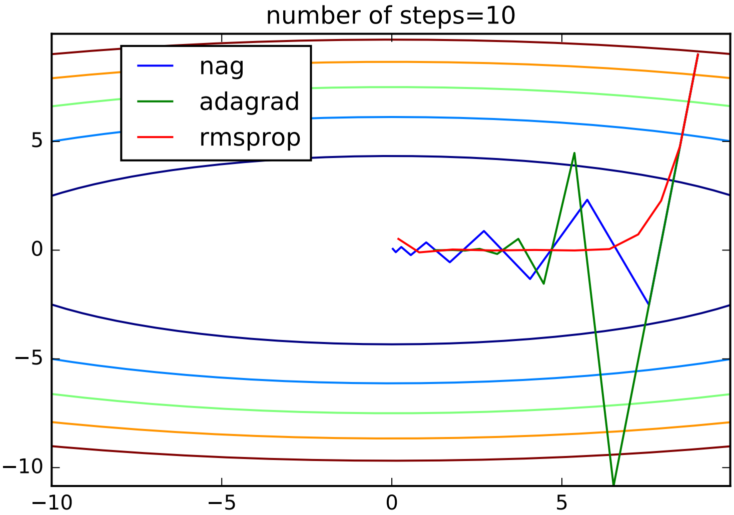
- Dividing the gradient by $\sqrt{MeanSquare(w,\ t)}$ makes the learning work much better (Tijmen Tieleman, unpublished).

Introduced in a slide in Geoff Hinton's Coursera class, lecture 6

Cited by several papers as:

[52] T. Tieleman and G. E. Hinton. Lecture 6.5-rmsprop: Divide the gradient by a running average of its recent magnitude., 2012.

Optimizers



ADAM

Adam update

[Kingma and Ba, 2014]

(incomplete, but close)

```
# Adam
m = beta1*m + (1-beta1)*dx # update first moment
v = beta2*v + (1-beta2)*(dx**2) # update second moment
x += - learning_rate * m / (np.sqrt(v) + 1e-7)
```

ADAM

Adam update

[Kingma and Ba, 2014]

(incomplete, but close)

```
# Adam
m = beta1*m + (1-beta1)*dx # update first moment
v = beta2*v + (1-beta2)*(dx**2) # update second moment
x += - learning_rate * m / (np.sqrt(v) + 1e-7)
```

momentum

RMSProp-like

Looks a bit like RMSProp with momentum

ADAM

[Kingma and Ba, 2014]

Adam update

(incomplete, but close)

```
# Adam
m = beta1*m + (1-beta1)*dx # update first moment
v = beta2*v + (1-beta2)*(dx**2) # update second moment
x += - learning_rate * m / (np.sqrt(v) + 1e-7)
```

momentum

RMSProp-like

Looks a bit like RMSProp with momentum

```
# RMSProp
cache = decay_rate * cache + (1 - decay_rate) * dx**2
x += - learning_rate * dx / (np.sqrt(cache) + 1e-7)
```

ADAM

Adam (full form)

```
first_moment = 0
second_moment = 0
for t in range(1, num_iterations):
  dx = compute_gradient(x)
  first_moment = beta1 * first_moment  + (1 - beta1) * dx
  second_moment = beta2 * second_moment + (1 - beta2) * dx * dx
  first_unbias = first_moment / (1 - beta1 ** t)
  second_unbias = second_moment / (1 - beta2 ** t)
  x -= learning_rate * first_unbias / (np.sqrt(second_unbias) + 1e-7))
```

Momentum

Bias correction

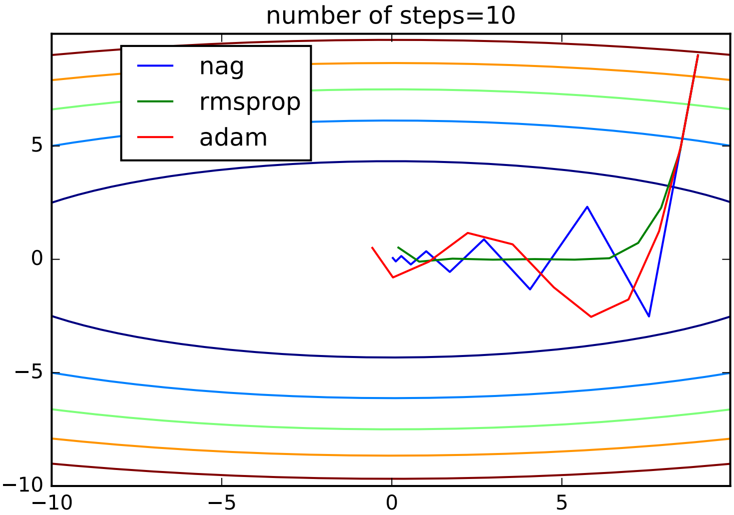
AdaGrad / RMSProp

Bias correction for the fact that first and second moment estimates start at zero

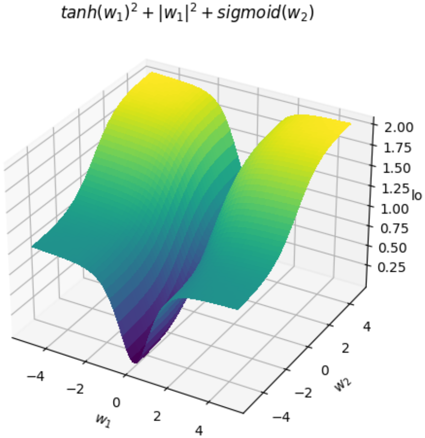
Adam with beta1 = 0.9, beta2 = 0.999, and learning_rate = 1e-3 or 5e-4 is a great starting point for many models!

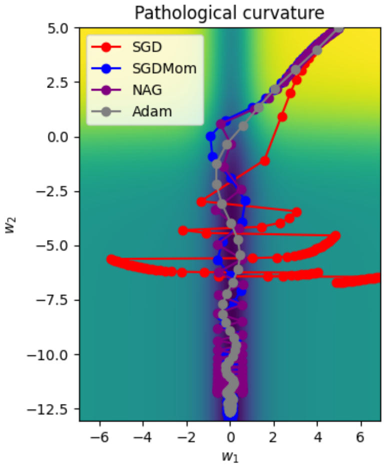
Kingma and Ba, “Adam: A method for stochastic optimization”, ICLR 2015

Optimizers



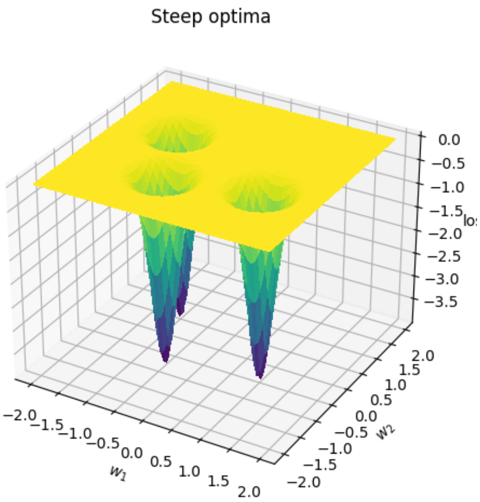
Pathological cases

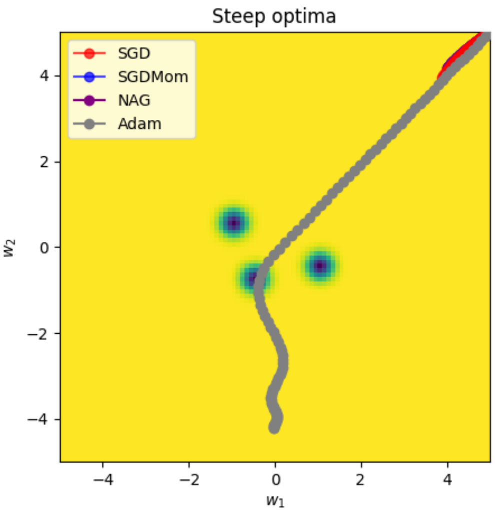




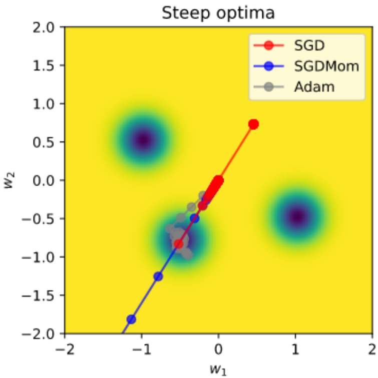
PyTorch Lightning Tutorial 3

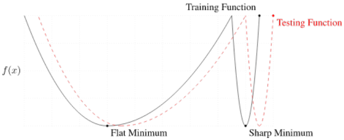
Pathological cases





Adam and Local Minima





- Several reported that Adam can be caught in deep local minimum and doesn't work well with ResNet (see this PyTorch tutorial post and here)
- Caught in deep minimum can be bad as the value of testing function can differ quite a bit for sharp minimum
- On the other hand, actual performance depends significantly with subtle details. I didn't see Adam got trapped by the local minima example. But I didn't try train on ResNet myself

AdamW

input : γ(lr), β_1, β_2(betas), θ_0(params), $f(\theta)$(objective), ϵ (epsilon)
λ(weight decay), *amsgrad*, *maximize*
initialize : $m_0 \leftarrow 0$ (first moment), $v_0 \leftarrow 0$ (second moment), $\widehat{v_0}^{max} \leftarrow 0$

for $t = 1$ **to** ... **do**
  if *maximize* :
    $g_t \leftarrow -\nabla_\theta f_t(\theta_{t-1})$
  else
    $g_t \leftarrow \nabla_\theta f_t(\theta_{t-1})$
  $\theta_t \leftarrow \theta_{t-1} - \gamma\lambda\theta_{t-1}$
  $m_t \leftarrow \beta_1 m_{t-1} + (1-\beta_1) g_t$
  $v_t \leftarrow \beta_2 v_{t-1} + (1-\beta_2) g_t^2$
  $\widehat{m_t} \leftarrow m_t / \big(1-\beta_1^t\big)$
  $\widehat{v_t} \leftarrow v_t / \big(1-\beta_2^t\big)$
  if *amsgrad*
    $\widehat{v_t}^{max} \leftarrow \max(\widehat{v_t}^{max}, \widehat{v_t})$
    $\theta_t \leftarrow \theta_t - \gamma\widehat{m_t} / \big(\sqrt{\widehat{v_t}^{max}} + \epsilon\big)$
  else
    $\theta_t \leftarrow \theta_t - \gamma\widehat{m_t} / \big(\sqrt{\widehat{v_t}} + \epsilon\big)$

return θ_t

- In "The Marginal Value of Adaptive Gradient Methods in Machine Learning," the authors question the effectiveness of adaptive gradient methods including AdaGrad, RMSProp and Adam. The debate is not final

AdamW

---

input : γ(lr), β_1, β_2(betas), θ_0(params), $f(\theta)$(objective), ϵ (epsilon)
λ(weight decay), *amsgrad*, *maximize*
initialize : $m_0 \leftarrow 0$ (first moment), $v_0 \leftarrow 0$ (second moment), $\widehat{v_0}^{max} \leftarrow 0$

---

for $t = 1$ **to** ... **do**
  if *maximize* :
    $g_t \leftarrow -\nabla_\theta f_t(\theta_{t-1})$
  else
    $g_t \leftarrow \nabla_\theta f_t(\theta_{t-1})$
  $\theta_t \leftarrow \theta_{t-1} - \gamma\lambda\theta_{t-1}$
  $m_t \leftarrow \beta_1 m_{t-1} + (1-\beta_1) g_t$
  $v_t \leftarrow \beta_2 v_{t-1} + (1-\beta_2) g_t^2$
  $\widehat{m_t} \leftarrow m_t / (1-\beta_1^t)$
  $\widehat{v_t} \leftarrow v_t / (1-\beta_2^t)$
  if *amsgrad*
    $\widehat{v_t}^{max} \leftarrow \max(\widehat{v_t}^{max}, \widehat{v_t})$
    $\theta_t \leftarrow \theta_t - \gamma\widehat{m_t} / (\sqrt{\widehat{v_t}^{max}} + \epsilon)$
  else
    $\theta_t \leftarrow \theta_t - \gamma\widehat{m_t} / (\sqrt{\widehat{v_t}} + \epsilon)$

---

return θ_t

- In "The Marginal Value of Adaptive Gradient Methods in Machine Learning," the authors question the effectiveness of adaptive gradient methods including AdaGrad, RMSProp and Adam. The debate is not final
- Some argued that Adam needs more regularization but the the conventional L2 regularization, which is the same as weight decay in plain SGD

AdamW

---

input : γ(lr), β_1, β_2(betas), θ_0(params), $f(\theta)$(objective), ϵ (epsilon)
λ(weight decay), *amsgrad*, *maximize*
initialize : $m_0 \leftarrow 0$ (first moment), $v_0 \leftarrow 0$ (second moment), $\widehat{v_0}^{max} \leftarrow 0$

---

for $t = 1$ **to** $\ldots$ **do**
 if *maximize* :
  $g_t \leftarrow -\nabla_\theta f_t(\theta_{t-1})$
 else
  $g_t \leftarrow \nabla_\theta f_t(\theta_{t-1})$
 $\theta_t \leftarrow \theta_{t-1} - \gamma\lambda\theta_{t-1}$
 $m_t \leftarrow \beta_1 m_{t-1} + (1-\beta_1) g_t$
 $v_t \leftarrow \beta_2 v_{t-1} + (1-\beta_2) g_t^2$
 $\widehat{m_t} \leftarrow m_t / \big(1-\beta_1^t\big)$
 $\widehat{v_t} \leftarrow v_t / \big(1-\beta_2^t\big)$
 if *amsgrad*
  $\widehat{v_t}^{max} \leftarrow \max(\widehat{v_t}^{max}, \widehat{v_t})$
  $\theta_t \leftarrow \theta_t - \gamma\widehat{m_t} / \big(\sqrt{\widehat{v_t}^{max}} + \epsilon\big)$
 else
  $\theta_t \leftarrow \theta_t - \gamma\widehat{m_t} / \big(\sqrt{\widehat{v_t}} + \epsilon\big)$

---

return θ_t

- In "The Marginal Value of Adaptive Gradient Methods in Machine Learning," the authors question the effectiveness of adaptive gradient methods including AdaGrad, RMSProp and Adam. The debate is not final
- Some argued that Adam needs more regularization but the the conventional L2 regularization, which is the same as weight decay in plain SGD
    - L2 regularization and weight decay are not the same in Adam

AdamW

---

input : γ(lr), β_1, β_2(betas), θ_0(params), $f(\theta)$(objective), ϵ (epsilon)
λ(weight decay), *amsgrad*, *maximize*
initialize : $m_0 \leftarrow 0$ (first moment), $v_0 \leftarrow 0$ (second moment), $\widehat{v_0}^{max} \leftarrow 0$

---

for $t = 1$ **to** $\ldots$ **do**
  if *maximize* :
    $g_t \leftarrow -\nabla_\theta f_t(\theta_{t-1})$
  else
    $g_t \leftarrow \nabla_\theta f_t(\theta_{t-1})$
  $\theta_t \leftarrow \theta_{t-1} - \gamma\lambda\theta_{t-1}$
  $m_t \leftarrow \beta_1 m_{t-1} + (1-\beta_1) g_t$
  $v_t \leftarrow \beta_2 v_{t-1} + (1-\beta_2) g_t^2$
  $\widehat{m_t} \leftarrow m_t / (1-\beta_1^t)$
  $\widehat{v_t} \leftarrow v_t / (1-\beta_2^t)$
  if *amsgrad*
    $\widehat{v_t}^{max} \leftarrow \max(\widehat{v_t}^{max}, \widehat{v_t})$
    $\theta_t \leftarrow \theta_t - \gamma\widehat{m_t} / (\sqrt{\widehat{v_t}^{max}} + \epsilon)$
  else
    $\theta_t \leftarrow \theta_t - \gamma\widehat{m_t} / (\sqrt{\widehat{v_t}} + \epsilon)$

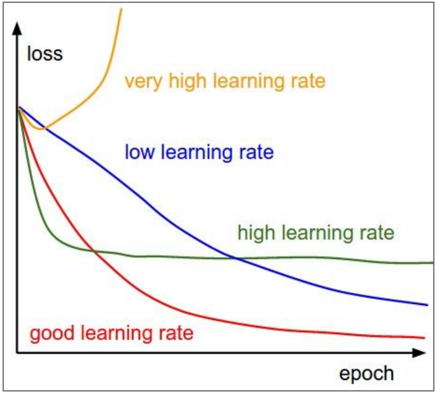
---

return θ_t

- In "The Marginal Value of Adaptive Gradient Methods in Machine Learning," the authors question the effectiveness of adaptive gradient methods including AdaGrad, RMSProp and Adam. The debate is not final
- Some argued that Adam needs more regularization but the the conventional L2 regularization, which is the same as weight decay in plain SGD
  - L2 regularization and weight decay are not the same in Adam
- AdamW implements weight decay for Adam, essential just an extra step

LR Scheduler

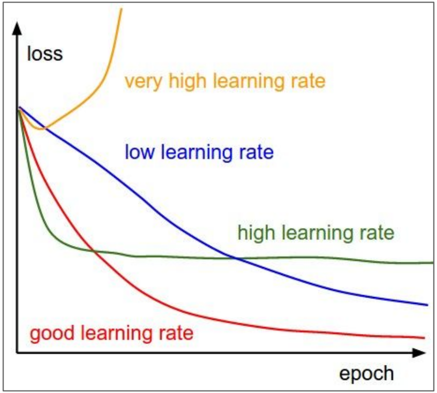
SGD, SGD+Momentum, Adagrad, RMSProp, Adam all have **learning rate** as a hyperparameter.



Q: Which one of these learning rates is best to use?

LR Scheduler

SGD, SGD+Momentum, Adagrad, RMSProp, Adam all have **learning rate** as a hyperparameter.



=> Learning rate decay over time!

step decay:
e.g. decay learning rate by half every few epochs.

exponential decay:

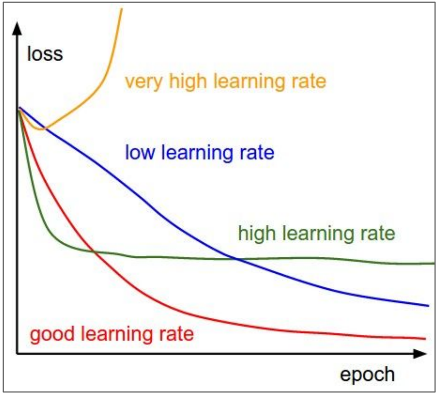
$$\alpha = \alpha_0 e^{-kt}$$

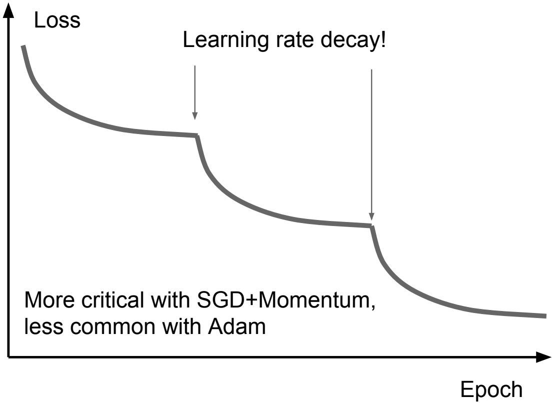
1/t decay:

$$\alpha = \alpha_0 / (1 + kt)$$

LR Scheduler

SGD, SGD+Momentum, Adagrad, RMSProp, Adam all have **learning rate** as a hyperparameter.

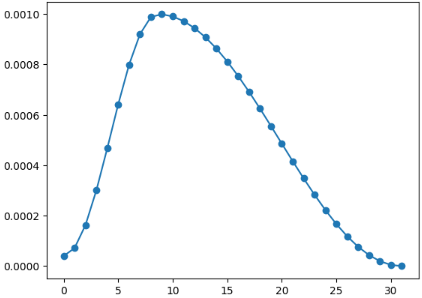




LR Scheduler

In [78]:
```
from torch.optim.lr_scheduler import OneCycleLR

lr=[]
scheduler = OneCycleLR(optimizer,
                       max_lr = 1e-3, # Upper LR boundaries
                       anneal_strategy = 'cos') # annealing strategy

for _ in range(32):
    lr.append(scheduler.get_last_lr())
    scheduler.step()

plt.plot(lr,'o-')
```

Out[78]: `[<matplotlib.lines.Line2D at 0x7fa96e6bb340>]`



- Many more schedulers are available
  - Check out torch.optim.lr_scheduler
  - optimizer = optim.SGD(parms,lr)
    scheduler = lr_scheduler.CyclicLR ...
    ...
    loss.backward()
    optimizer.step()
    scheduler.step()
- In particular, check out
  - OneCycleLR
    - Recommended by FastAI
  - CosineAnnealingWarmRestartsLR
    - Try to escape local minima

2nd order optimizers

Second order optimization methods

second-order Taylor expansion:

$$J(\boldsymbol{\theta}) \approx J(\boldsymbol{\theta}_0) + (\boldsymbol{\theta} - \boldsymbol{\theta}_0)^\top \nabla_{\boldsymbol{\theta}} J(\boldsymbol{\theta}_0) + \frac{1}{2}(\boldsymbol{\theta} - \boldsymbol{\theta}_0)^\top \boldsymbol{H}(\boldsymbol{\theta} - \boldsymbol{\theta}_0)$$

Solving for the critical point we obtain the Newton parameter update:

$$\boldsymbol{\theta}^* = \boldsymbol{\theta}_0 - \boldsymbol{H}^{-1} \nabla_{\boldsymbol{\theta}} J(\boldsymbol{\theta}_0)$$

Q: what is nice about this update?

2nd order optimizer

Second order optimization methods

$$\boldsymbol{\theta}^* = \boldsymbol{\theta}_0 - \boldsymbol{H}^{-1} \nabla_{\boldsymbol{\theta}} J(\boldsymbol{\theta}_0)$$

Inverting Hessian is very expensive ($O(N^3)$). Avoiding that resulting in so-called Quasi-Newton methods

2nd order optimizer

Second order optimization methods

$$\boldsymbol{\theta}^* = \boldsymbol{\theta}_0 - \boldsymbol{H}^{-1} \nabla_{\boldsymbol{\theta}} J(\boldsymbol{\theta}_0)$$

Inverting Hessian is very expensive ($O(N^3)$). Avoiding that resulting in so-called Quasi-Newton methods

- Rank-1 inverse Hessian update (simple but not too commonly used)

2nd order optimizer

Second order optimization methods

$$\boldsymbol{\theta}^* = \boldsymbol{\theta}_0 - \boldsymbol{H}^{-1}\nabla_{\boldsymbol{\theta}} J(\boldsymbol{\theta}_0)$$

Inverting Hessian is very expensive ($O(N^3)$). Avoiding that resulting in so-called Quasi-Newton methods

- Rank-1 inverse Hessian update (simple but not too commonly used)
- Rank-2 inverse Hessian update

2nd order optimizer

Second order optimization methods

$$\boldsymbol{\theta}^* = \boldsymbol{\theta}_0 - \boldsymbol{H}^{-1} \nabla_{\boldsymbol{\theta}} J(\boldsymbol{\theta}_0)$$

Inverting Hessian is very expensive ($O(N^3)$). Avoiding that resulting in so-called Quasi-Newton methods

- Rank-1 inverse Hessian update (simple but not too commonly used)
- Rank-2 inverse Hessian update
    - BFGS (most popular) and DFS

2nd order optimizer

Second order optimization methods

$$\boldsymbol{\theta}^* = \boldsymbol{\theta}_0 - \boldsymbol{H}^{-1} \nabla_{\boldsymbol{\theta}} J(\boldsymbol{\theta}_0)$$

Inverting Hessian is very expensive ($O(N^3)$). Avoiding that resulting in so-called Quasi-Newton methods

- Rank-1 inverse Hessian update (simple but not too commonly used)
- Rank-2 inverse Hessian update
  - BFGS (most popular) and DFS
  - LBFGS
    - Does not store the entire inverse Hessian
    - Tradeoff space with time and accuracy

Quasi-Newton methods (watch this)

- Ref:
    1. https://www.youtube.com/watch?v=uo2z0AT_83k
    2. Nocedal & Wright - Numerical Optimization ($B \leftrightarrow H$)
    3. http://users.ece.utexas.edu/ cmcaram/EE381V_2012F/Lecture_10_Scribe_Notes.final.pdf
- The inverse of Hessian H is expensive to compute. Want to approximate it iteratively instead

Quasi-Newton methods (watch this)

- Ref:
  1. https://www.youtube.com/watch?v=uo2z0AT_83k
  2. Nocedal & Wright - Numerical Optimization ($B \leftrightarrow H$)
  3. http://users.ece.utexas.edu/ cmcaram/EE381V_2012F/Lecture_10_Scribe_Notes.final.pdf
- The inverse of Hessian H is expensive to compute. Want to approximate it iteratively instead
- Quasi-Newton methods:
  1. Approximate Newton direction

$$d_k \leftarrow -B_k g_k,$$

where $B_k \approx H_k^{-1}$ and $g_k = \nabla J(\theta_k)$

Quasi-Newton methods (watch this)

- Ref:
  1. https://www.youtube.com/watch?v=uo2z0AT_83k
  2. Nocedal & Wright - Numerical Optimization ($B \leftrightarrow H$)
  3. http://users.ece.utexas.edu/ cmcaram/EE381V_2012F/Lecture_10_Scribe_Notes.final.pdf
- The inverse of Hessian H is expensive to compute. Want to approximate it iteratively instead
- Quasi-Newton methods:
  1. Approximate Newton direction
     $$d_k \leftarrow -B_k g_k,$$
     where $B_k \approx H_k^{-1}$ and $g_k = \nabla J(\theta_k)$
  2. Line search: $\theta_{k+1} = \theta_k + \alpha_k d_k$

Quasi-Newton methods (watch this)

- Ref:
  1. https://www.youtube.com/watch?v=uo2z0AT_83k
  2. Nocedal & Wright - Numerical Optimization ($B \leftrightarrow H$)
  3. http://users.ece.utexas.edu/ cmcaram/EE381V_2012F/Lecture_10_Scribe_Notes.final.pdf
- The inverse of Hessian H is expensive to compute. Want to approximate it iteratively instead
- Quasi-Newton methods:
  1. Approximate Newton direction
  $$d_k \leftarrow -B_k g_k,$$
  where $B_k \approx H_k^{-1}$ and $g_k = \nabla J(\theta_k)$
  2. Line search: $\theta_{k+1} = \theta_k + \alpha_k d_k$
  3. Update $g_{k+1} = \nabla J(\theta_{k+1})$

Quasi-Newton methods (watch this)

- Ref:
  1. https://www.youtube.com/watch?v=uo2z0AT_83k
  2. Nocedal & Wright - Numerical Optimization ($B \leftrightarrow H$)
  3. http://users.ece.utexas.edu/ cmcaram/EE381V_2012F/Lecture_10_Scribe_Notes.final.pdf
- The inverse of Hessian H is expensive to compute. Want to approximate it iteratively instead
- Quasi-Newton methods:
  1. Approximate Newton direction
  $$d_k \leftarrow -B_k g_k,$$
  where $B_k \approx H_k^{-1}$ and $g_k = \nabla J(\theta_k)$
  2. Line search: $\theta_{k+1} = \theta_k + \alpha_k d_k$
  3. Update $g_{k+1} = \nabla J(\theta_{k+1})$
  4. Approximate inverse Hessian
  $$B_{k+1} = \text{update\_formula}(B_k, \theta_{k+1} - \theta_k, g_{k+1} - g_k)$$

Approximation with rank-1 update

- As Hessian is essentially the "derivative" of ∇J, we have $\nabla J(\theta_{k+1}) \approx \nabla J(\theta_k) + H(\theta_{k+1} - \theta_k)$

Approximation with rank-1 update

- As Hessian is essentially the "derivative" of ∇J, we have $\nabla J(\theta_{k+1}) \approx \nabla J(\theta_k) + H(\theta_{k+1} - \theta_k)$
- We may assume the above is satisfied and use this to iteratively approximate H.

Approximation with rank-1 update

- As Hessian is essentially the "derivative" of ∇J, we have $\nabla J(\theta_{k+1}) \approx \nabla J(\theta_k) + H(\theta_{k+1} - \theta_k)$
- We may assume the above is satisfied and use this to iteratively approximate H. That is (known as secant equation) $Hp_k = q_k$, where $p_k = \theta_{k+1} - \theta_k$ and $q_k = \nabla J(\theta_{k+1}) - \nabla J(\theta_k)$

Approximation with rank-1 update

- As Hessian is essentially the "derivative" of ∇J, we have $\nabla J(\theta_{k+1}) \approx \nabla J(\theta_k) + H(\theta_{k+1} - \theta_k)$
- We may assume the above is satisfied and use this to iteratively approximate H. That is (known as secant equation) $Hp_k = q_k$, where $p_k = \theta_{k+1} - \theta_k$ and $q_k = \nabla J(\theta_{k+1}) - \nabla J(\theta_k)$
- Let $H_{k+1} = H_k + uv^T$

Approximation with rank-1 update

- As Hessian is essentially the "derivative" of ∇J, we have $\nabla J(\theta_{k+1}) \approx \nabla J(\theta_k) + H(\theta_{k+1} - \theta_k)$
- We may assume the above is satisfied and use this to iteratively approximate H. That is (known as secant equation) $Hp_k = q_k$, where $p_k = \theta_{k+1} - \theta_k$ and $q_k = \nabla J(\theta_{k+1}) - \nabla J(\theta_k)$
- Let $H_{k+1} = H_k + uv^T \Rightarrow (H_k + uv^T)p_k = q_k$

Approximation with rank-1 update

- As Hessian is essentially the "derivative" of ∇J, we have $\nabla J(\theta_{k+1}) \approx \nabla J(\theta_k) + H(\theta_{k+1} - \theta_k)$
- We may assume the above is satisfied and use this to iteratively approximate H. That is (known as secant equation) $Hp_k = q_k$, where $p_k = \theta_{k+1} - \theta_k$ and $q_k = \nabla J(\theta_{k+1}) - \nabla J(\theta_k)$
- Let $H_{k+1} = H_k + uv^T \Rightarrow (H_k + uv^T)p_k = q_k \Rightarrow u(v^T p_k) = q_k - H_k p_k$

Approximation with rank-1 update

- As Hessian is essentially the "derivative" of ∇J, we have $\nabla J(\theta_{k+1}) \approx \nabla J(\theta_k) + H(\theta_{k+1} - \theta_k)$
- We may assume the above is satisfied and use this to iteratively approximate H. That is (known as secant equation) $Hp_k = q_k$, where $p_k = \theta_{k+1} - \theta_k$ and $q_k = \nabla J(\theta_{k+1}) - \nabla J(\theta_k)$
- Let $H_{k+1} = H_k + uv^T \Rightarrow (H_k + uv^T)p_k = q_k \Rightarrow u(v^T p_k) = q_k - H_k p_k$ $\Rightarrow u = \frac{1}{v^T p_k}(q_k - H_k p_k)$

Approximation with rank-1 update

- As Hessian is essentially the "derivative" of ∇J, we have $\nabla J(\theta_{k+1}) \approx \nabla J(\theta_k) + H(\theta_{k+1} - \theta_k)$
- We may assume the above is satisfied and use this to iteratively approximate H. That is (known as secant equation) $Hp_k = q_k$, where $p_k = \theta_{k+1} - \theta_k$ and $q_k = \nabla J(\theta_{k+1}) - \nabla J(\theta_k)$
- Let $H_{k+1} = H_k + uv^T \Rightarrow (H_k + uv^T)p_k = q_k \Rightarrow u(v^T p_k) = q_k - H_k p_k$ $\Rightarrow u = \frac{1}{v^T p_k}(q_k - H_k p_k) \Rightarrow H_{k+1} = H_k + \frac{1}{v^T p_k}(q_k - H_k p_k)v^T$

Approximation with rank-1 update

- As Hessian is essentially the "derivative" of ∇J, we have $\nabla J(\theta_{k+1}) \approx \nabla J(\theta_k) + H(\theta_{k+1} - \theta_k)$
- We may assume the above is satisfied and use this to iteratively approximate H. That is (known as secant equation) $Hp_k = q_k$, where $p_k = \theta_{k+1} - \theta_k$ and $q_k = \nabla J(\theta_{k+1}) - \nabla J(\theta_k)$
- Let $H_{k+1} = H_k + uv^T \Rightarrow (H_k + uv^T)p_k = q_k \Rightarrow u(v^T p_k) = q_k - H_k p_k$ $\Rightarrow u = \frac{1}{v^T p_k}(q_k - H_k p_k) \Rightarrow H_{k+1} = H_k + \frac{1}{v^T p_k}(q_k - H_k p_k)v^T$
- We are free to pick v. But since we know H has to be symmetric, let's pick $v = q_k - H_k p_k$.

Approximation with rank-1 update

- As Hessian is essentially the "derivative" of ∇J, we have $\nabla J(\theta_{k+1}) \approx \nabla J(\theta_k) + H(\theta_{k+1} - \theta_k)$
- We may assume the above is satisfied and use this to iteratively approximate H. That is (known as secant equation) $Hp_k = q_k$, where $p_k = \theta_{k+1} - \theta_k$ and $q_k = \nabla J(\theta_{k+1}) - \nabla J(\theta_k)$
- Let $H_{k+1} = H_k + uv^T \Rightarrow (H_k + uv^T)p_k = q_k \Rightarrow u(v^T p_k) = q_k - H_k p_k$ $\Rightarrow u = \frac{1}{v^T p_k}(q_k - H_k p_k) \Rightarrow H_{k+1} = H_k + \frac{1}{v^T p_k}(q_k - H_k p_k)v^T$
- We are free to pick v. But since we know H has to be symmetric, let's pick $v = q_k - H_k p_k$. Thus

$$H_{k+1} = H_k + \frac{1}{v^T p_k} v v^T$$

with $v = q_k - H_k p_k$

Updating B

- Recall that we need $B_k = H_k^{-1}$ to approximate the Newton direction ($d_k \leftarrow -B_k g_k$)

Updating B

- Recall that we need $B_k = H_k^{-1}$ to approximate the Newton direction ($d_k \leftarrow -B_k g_k$)
- We don't need to invert the matrix H_k directly. Note that $Hp_k = q_k$ give us $H_{k+1} = H_k + \frac{1}{v^T p_k} v v^T$ and $v = q_k - H_k p_k$

Updating B

- Recall that we need $B_k = H_k^{-1}$ to approximate the Newton direction ($d_k \leftarrow -B_k g_k$)
- We don't need to invert the matrix H_k directly. Note that $Hp_k = q_k$ give us $H_{k+1} = H_k + \frac{1}{v^T p_k} v v^T$ and $v = q_k - H_k p_k$
- Similarly, since $Hp_k = q_k \Rightarrow Bq_k = p_k$, we have

$$B_{k+1} = B_k + \frac{1}{w^T q_k} w w^T$$

with $w = p_k - B_k q_k$

Rank-2 approximation

- BFGS utilizes rank-2 approximation update for H. There are other variations (such as DFP). But BFGS is considered the state of the art

Rank-2 approximation

- BFGS utilizes rank-2 approximation update for H. There are other variations (such as DFP). But BFGS is considered the state of the art
- Recall our rank-1 approximation that
  $H_{k+1} = H_k + \frac{1}{v^T p_k} v v^T$ and $v = q_k - H_k p_k$

Rank-2 approximation

- BFGS utilizes rank-2 approximation update for H. There are other variations (such as DFP). But BFGS is considered the state of the art
- Recall our rank-1 approximation that $H_{k+1} = H_k + \frac{1}{v^T p_k} v v^T$ and $v = q_k - H_k p_k$
- Consider update $H_{k+1} = H_k + \frac{1}{\alpha} u u^T + \frac{1}{\beta} w w^T$ instead.

Rank-2 approximation

- BFGS utilizes rank-2 approximation update for H. There are other variations (such as DFP). But BFGS is considered the state of the art
- Recall our rank-1 approximation that $H_{k+1} = H_k + \frac{1}{v^T p_k} v v^T$ and $v = q_k - H_k p_k$
- Consider update $H_{k+1} = H_k + \frac{1}{\alpha} u u^T + \frac{1}{\beta} w w^T$ instead.
  - Need to pick u and w, q_k and $H_k p_k$ are reasonable choice

Rank-2 approximation

- BFGS utilizes rank-2 approximation update for H. There are other variations (such as DFP). But BFGS is considered the state of the art
- Recall our rank-1 approximation that $H_{k+1} = H_k + \frac{1}{v^T p_k} v v^T$ and $v = q_k - H_k p_k$
- Consider update $H_{k+1} = H_k + \frac{1}{\alpha} u u^T + \frac{1}{\beta} w w^T$ instead.
  - Need to pick u and w, q_k and $H_k p_k$ are reasonable choice
- Again, we want $H_{k+1} p_k = q_k \Rightarrow H_k p_k + \frac{1}{\alpha} q_k (q_k^T p_k) + \frac{1}{\beta} H_k p_k (p_k^T H_k^T p_k) = q_k$.

Rank-2 approximation

- BFGS utilizes rank-2 approximation update for H. There are other variations (such as DFP). But BFGS is considered the state of the art
- Recall our rank-1 approximation that $H_{k+1} = H_k + \frac{1}{v^T p_k} v v^T$ and $v = q_k - H_k p_k$
- Consider update $H_{k+1} = H_k + \frac{1}{\alpha} u u^T + \frac{1}{\beta} w w^T$ instead.
  - Need to pick u and w, q_k and $H_k p_k$ are reasonable choice
- Again, we want $H_{k+1} p_k = q_k \Rightarrow H_k p_k + \frac{1}{\alpha} q_k (q_k^T p_k) + \frac{1}{\beta} H_k p_k (p_k^T H_k^T p_k) = q_k$. By inspection, this can be satisfied if we pick $\alpha = q_k^T p_k$ and $\beta = -p_k^T H_k^T p_k$.

Rank-2 approximation

- BFGS utilizes rank-2 approximation update for H. There are other variations (such as DFP). But BFGS is considered the state of the art
- Recall our rank-1 approximation that $H_{k+1} = H_k + \frac{1}{v^T p_k} v v^T$ and $v = q_k - H_k p_k$
- Consider update $H_{k+1} = H_k + \frac{1}{\alpha} u u^T + \frac{1}{\beta} w w^T$ instead.
  - Need to pick u and w, q_k and $H_k p_k$ are reasonable choice
- Again, we want $H_{k+1} p_k = q_k \Rightarrow H_k p_k + \frac{1}{\alpha} q_k (q_k^T p_k) + \frac{1}{\beta} H_k p_k (p_k^T H_k^T p_k) = q_k$. By inspection, this can be satisfied if we pick $\alpha = q_k^T p_k$ and $\beta = -p_k^T H_k^T p_k$. Thus we have

$$H_{k+1} = H_k + \frac{q_k q_k^T}{q_k^T p_k} - \frac{H_k p_k p_k^T H_k}{p_k^T H_k^T p_k}$$

Sherman-Morrison-formula

- But we are interested in $B_k = H_k^{-1}$
- Sherman-Morrison-formula:

$$(A + uv^T)^{-1} = A^{-1} + \frac{A^{-1}uv^T A^{-1}}{1 - v^T A^{-1} u}$$

Proof.

$(A + uv^T)\left(A^{-1} - \frac{A^{-1}uv^T A^{-1}}{1+v^T A^{-1}u}\right)$

Sherman-Morrison-formula

- But we are interested in $B_k = H_k^{-1}$
- Sherman-Morrison-formula:

$$(A + uv^T)^{-1} = A^{-1} + \frac{A^{-1}uv^TA^{-1}}{1 - v^TA^{-1}u}$$

Proof.

$$(A + uv^T)\left(A^{-1} - \frac{A^{-1}uv^TA^{-1}}{1+v^TA^{-1}u}\right) = AA^{-1} + uv^TA^{-1}$$

Sherman-Morrison-formula

- But we are interested in $B_k = H_k^{-1}$
- Sherman-Morrison-formula:

$$(A + uv^T)^{-1} = A^{-1} + \frac{A^{-1}uv^T A^{-1}}{1 - v^T A^{-1} u}$$

Proof.

$$(A + uv^T)\left(A^{-1} - \frac{A^{-1}uv^T A^{-1}}{1+v^T A^{-1}u}\right) = AA^{-1} + uv^T A^{-1} - \frac{AA^{-1}uv^T A^{-1} + uv^T A^{-1}uv^T A^{-1}}{1+v^T A^{-1}u}$$

Sherman-Morrison-formula

- But we are interested in $B_k = H_k^{-1}$
- Sherman-Morrison-formula:

$$(A + uv^T)^{-1} = A^{-1} + \frac{A^{-1}uv^TA^{-1}}{1 - v^TA^{-1}u}$$

Proof.

$$(A + uv^T)\left(A^{-1} - \frac{A^{-1}uv^TA^{-1}}{1+v^TA^{-1}u}\right) = AA^{-1} + uv^TA^{-1} - \frac{AA^{-1}uv^TA^{-1} + uv^TA^{-1}uv^TA^{-1}}{1+v^TA^{-1}u}$$

$$= I + uv^TA^{-1} - \frac{uv^TA^{-1} + uv^TA^{-1}uv^TA^{-1}}{1+v^TA^{-1}u}$$

Sherman-Morrison-formula

- But we are interested in $B_k = H_k^{-1}$
- Sherman-Morrison-formula:

$$(A + uv^T)^{-1} = A^{-1} + \frac{A^{-1}uv^T A^{-1}}{1 - v^T A^{-1} u}$$

Proof.

$$(A + uv^T)\left(A^{-1} - \frac{A^{-1}uv^T A^{-1}}{1+v^T A^{-1}u}\right) = AA^{-1} + uv^T A^{-1} - \frac{AA^{-1}uv^T A^{-1} + uv^T A^{-1}uv^T A^{-1}}{1+v^T A^{-1}u}$$

$$= I + uv^T A^{-1} - \frac{uv^T A^{-1} + uv^T A^{-1}uv^T A^{-1}}{1+v^T A^{-1}u} = I + uv^T A^{-1} - \frac{u(1+v^T A^{-1}u)v^T A^{-1}}{1+v^T A^{-1}u}$$

Sherman-Morrison-formula

- But we are interested in $B_k = H_k^{-1}$
- Sherman-Morrison-formula:

$$(A + uv^T)^{-1} = A^{-1} + \frac{A^{-1}uv^TA^{-1}}{1 - v^TA^{-1}u}$$

Proof.

$$(A + uv^T)\left(A^{-1} - \frac{A^{-1}uv^TA^{-1}}{1+v^TA^{-1}u}\right) = AA^{-1} + uv^TA^{-1} - \frac{AA^{-1}uv^TA^{-1}+uv^TA^{-1}uv^TA^{-1}}{1+v^TA^{-1}u}$$
$$= I + uv^TA^{-1} - \frac{uv^TA^{-1}+uv^TA^{-1}uv^TA^{-1}}{1+v^TA^{-1}u} = I + uv^TA^{-1} - \frac{u(1+v^TA^{-1}u)v^TA^{-1}}{1+v^TA^{-1}u}$$
$$= I + uv^TA^{-1} - uv^TA^{-1} = I$$
□

BFGS

- Recall $H_{k+1} = \underbrace{H_k + \frac{q_k q_k^T}{q_k^T p_k}}_{D} - \frac{H_k p_k p_k^T H_k}{p_k^T H_k^T p_k}$ and $(A + uv^T)^{-1} = A^{-1} + \frac{A^{-1}uv^T A^{-1}}{1 - v^T A^{-1} u}$

BFGS

- Recall $H_{k+1} = \underbrace{H_k + \frac{q_k q_k^T}{q_k^T p_k}}_{D} - \frac{H_k p_k p_k^T H_k}{p_k^T H_k^T p_k}$ and $(A + uv^T)^{-1} = A^{-1} + \frac{A^{-1}uv^T A^{-1}}{1 - v^T A^{-1} u}$
- $D^{-1} = (H + \frac{qq^T}{q^T p})^{-1} = H^{-1} + \frac{H^{-1}qq^T H^{-1}}{(q^T p)(1 - q^T H^{-1} q/(q^T p))} = B + \frac{Bqq^T B}{q^T p - q^T B q}$

BFGS

- Recall $H_{k+1} = \underbrace{H_k + \frac{q_k q_k^T}{q_k^T p_k}}_{D} - \frac{H_k p_k p_k^T H_k}{p_k^T H_k^T p_k}$ and $(A + uv^T)^{-1} = A^{-1} + \frac{A^{-1} u v^T A^{-1}}{1 - v^T A^{-1} u}$
- $D^{-1} = (H + \frac{qq^T}{q^T p})^{-1} = H^{-1} + \frac{H^{-1} q q^T H^{-1}}{(q^T p)(1 - q^T H^{-1} q/(q^T p))} = B + \frac{B q q^T B}{q^T p - q^T B q}$
- $(D - \frac{H p p^T H}{p^T H^T p})^{-1} = D^{-1} - \frac{D^{-1} H p p^T H D^{-1}}{p^T H^T p (1 - p^T H D^{-1} H p/(p^T H^T p))}$

BFGS

- Recall $H_{k+1} = \underbrace{H_k + \frac{q_k q_k^T}{q_k^T p_k}}_{D} - \frac{H_k p_k p_k^T H_k}{p_k^T H_k^T p_k}$ and $(A + uv^T)^{-1} = A^{-1} + \frac{A^{-1}uv^T A^{-1}}{1 - v^T A^{-1} u}$

- $D^{-1} = (H + \frac{qq^T}{q^T p})^{-1} = H^{-1} + \frac{H^{-1}qq^T H^{-1}}{(q^T p)(1 - q^T H^{-1} q/(q^T p))} = B + \frac{Bqq^T B}{q^T p - q^T B q}$
- $(D - \frac{Hpp^T H}{p^T H^T p})^{-1} = D^{-1} - \frac{D^{-1}Hpp^T H D^{-1}}{p^T H^T p(1 - p^T H D^{-1} Hp/(p^T H^T p))} = D^{-1} - \frac{D^{-1}Hpp^T H D^{-1}}{p^T Hp - p^T H D^{-1} Hp}$

BFGS

- Recall $H_{k+1} = \underbrace{H_k + \frac{q_k q_k^T}{q_k^T p_k}}_{D} - \frac{H_k p_k p_k^T H_k}{p_k^T H_k^T p_k}$ and $(A + uv^T)^{-1} = A^{-1} + \frac{A^{-1} u v^T A^{-1}}{1 - v^T A^{-1} u}$
- $D^{-1} = (H + \frac{qq^T}{q^T p})^{-1} = H^{-1} + \frac{H^{-1} q q^T H^{-1}}{(q^T p)(1 - q^T H^{-1} q / (q^T p))} = B + \frac{Bqq^T B}{q^T p - q^T B q}$
- $(D - \frac{Hpp^T H}{p^T H^T p})^{-1} = D^{-1} - \frac{D^{-1} H p p^T H D^{-1}}{p^T H^T p (1 - p^T H D^{-1} H p / (p^T H^T p))} = D^{-1} - \frac{D^{-1} H p p^T H D^{-1}}{p^T H p - p^T H D^{-1} H p}$
- $D^{-1} H p = (BHp + \frac{Bqq^T BHp}{q^T p - q^T B q}) = (p + \frac{Bqq^T p}{q^T p - q^T B q})$

BFGS

- Recall $H_{k+1} = \underbrace{H_k + \frac{q_k q_k^T}{q_k^T p_k}}_{D} - \frac{H_k p_k p_k^T H_k}{p_k^T H_k^T p_k}$ and $(A + uv^T)^{-1} = A^{-1} + \frac{A^{-1} uv^T A^{-1}}{1 - v^T A^{-1} u}$
- $D^{-1} = (H + \frac{qq^T}{q^T p})^{-1} = H^{-1} + \frac{H^{-1} qq^T H^{-1}}{(q^T p)(1 - q^T H^{-1} q/(q^T p))} = B + \frac{Bqq^T B}{q^T p - q^T Bq}$
- $(D - \frac{Hpp^T H}{p^T H^T p})^{-1} = D^{-1} - \frac{D^{-1} Hpp^T HD^{-1}}{p^T H^T p(1 - p^T HD^{-1} Hp/(p^T H^T p))} = D^{-1} - \frac{D^{-1} Hpp^T HD^{-1}}{p^T Hp - p^T HD^{-1} Hp}$
- $D^{-1} Hp = (BHp + \frac{Bqq^T BHp}{q^T p - q^T Bq}) = (p + \frac{Bqq^T p}{q^T p - q^T Bq})$
- $(D - \frac{Hpp^T H}{p^T H^T p})^{-1} = D^{-1} - \frac{D^{-1} Hpp^T HD^{-1}}{p^T qq^T p(q^T p - q^T Bq)}$

BFGS

- Recall $H_{k+1} = \underbrace{H_k + \frac{q_k q_k^T}{q_k^T p_k}}_{D} - \frac{H_k p_k p_k^T H_k}{p_k^T H_k^T p_k}$ and $(A + uv^T)^{-1} = A^{-1} + \frac{A^{-1}uv^T A^{-1}}{1 - v^T A^{-1} u}$
- $D^{-1} = (H + \frac{qq^T}{q^T p})^{-1} = H^{-1} + \frac{H^{-1}qq^T H^{-1}}{(q^T p)(1 - q^T H^{-1} q/(q^T p))} = B + \frac{Bqq^T B}{q^T p - q^T Bq}$
- $(D - \frac{Hpp^T H}{p^T H^T p})^{-1} = D^{-1} - \frac{D^{-1}Hpp^T HD^{-1}}{p^T H^T p(1 - p^T HD^{-1}Hp/(p^T H^T p))} = D^{-1} - \frac{D^{-1}Hpp^T HD^{-1}}{p^T Hp - p^T HD^{-1}Hp}$
- $D^{-1}Hp = (BHp + \frac{Bqq^T BHp}{q^T p - q^T Bq}) = (p + \frac{Bqq^T p}{q^T p - q^T Bq})$
- $(D - \frac{Hpp^T H}{p^T H^T p})^{-1} = D^{-1} - \frac{D^{-1}Hpp^T HD^{-1}}{p^T qq^T p(q^T p - q^T Bq)} \cdots$

BFGS

- Recall $H_{k+1} = \underbrace{H_k + \frac{q_k q_k^T}{q_k^T p_k}}_{D} - \frac{H_k p_k p_k^T H_k}{p_k^T H_k^T p_k}$ and $(A + uv^T)^{-1} = A^{-1} + \frac{A^{-1}uv^T A^{-1}}{1 - v^T A^{-1} u}$
- $D^{-1} = (H + \frac{qq^T}{q^T p})^{-1} = H^{-1} + \frac{H^{-1}qq^T H^{-1}}{(q^T p)(1 - q^T H^{-1} q/(q^T p))} = B + \frac{Bqq^T B}{q^T p - q^T Bq}$
- $(D - \frac{Hpp^T H}{p^T H^T p})^{-1} = D^{-1} - \frac{D^{-1}Hpp^T HD^{-1}}{p^T H^T p(1 - p^T HD^{-1}Hp/(p^T H^T p))} = D^{-1} - \frac{D^{-1}Hpp^T HD^{-1}}{p^T Hp - p^T HD^{-1}Hp}$
- $D^{-1}Hp = (BHp + \frac{Bqq^T BHp}{q^T p - q^T Bq}) = (p + \frac{Bqq^T p}{q^T p - q^T Bq})$
- $(D - \frac{Hpp^T H}{p^T H^T p})^{-1} = D^{-1} - \frac{D^{-1}Hpp^T HD^{-1}}{p^T qq^T p(q^T p - q^T Bq)} \cdots$
- $(D - \frac{Hpp^T H}{p^T H^T p})^{-1} = \left(I - \frac{pq^T}{q^T p}\right) B \left(I - \frac{qp^T}{q^T p}\right) + \frac{pp^T}{q^T p} \Rightarrow B_{k+1} = \left(I - \frac{p_k q_k^T}{q_k^T p_k}\right) B_k \left(I - \frac{q_k p_k^T}{q_k^T p_k}\right) + \frac{p_k p_k^T}{q_k^T p_k}$

BFGS

- Recall $H_{k+1} = \underbrace{H_k + \frac{q_k q_k^T}{q_k^T p_k}}_{D} - \frac{H_k p_k p_k^T H_k}{p_k^T H_k^T p_k}$ and $(A + uv^T)^{-1} = A^{-1} + \frac{A^{-1}uv^T A^{-1}}{1 - v^T A^{-1} u}$
- $D^{-1} = (H + \frac{qq^T}{q^T p})^{-1} = H^{-1} + \frac{H^{-1}qq^T H^{-1}}{(q^T p)(1 - q^T H^{-1} q/(q^T p))} = B + \frac{Bqq^T B}{q^T p - q^T Bq}$
- $(D - \frac{Hpp^T H}{p^T H^T p})^{-1} = D^{-1} - \frac{D^{-1}Hpp^T HD^{-1}}{p^T H^T p(1 - p^T HD^{-1}Hp/(p^T H^T p))} = D^{-1} - \frac{D^{-1}Hpp^T HD^{-1}}{p^T Hp - p^T HD^{-1}Hp}$
- $D^{-1}Hp = (BHp + \frac{Bqq^T BHp}{q^T p - q^T Bq}) = (p + \frac{Bqq^T p}{q^T p - q^T Bq})$
- $(D - \frac{Hpp^T H}{p^T H^T p})^{-1} = D^{-1} - \frac{D^{-1}Hpp^T HD^{-1}}{p^T qq^T p(q^T p - q^T Bq)} \cdots$
- $(D - \frac{Hpp^T H}{p^T H^T p})^{-1} = \left(I - \frac{pq^T}{q^T p}\right) B \left(I - \frac{qp^T}{q^T p}\right) + \frac{pp^T}{q^T p} \Rightarrow B_{k+1} = \left(I - \frac{p_k q_k^T}{q_k^T p_k}\right) B_k \left(I - \frac{q_k p_k^T}{q_k^T p_k}\right) + \frac{p_k p_k^T}{q_k^T p_k}$
- Bounty: 3% bonus to complete the algebra

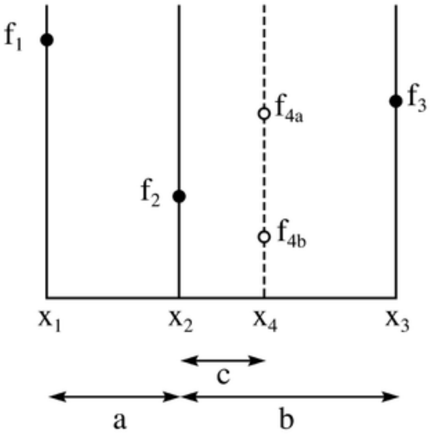
Summary of BFGS

Initialize Initialize inverse Hessian approximation $B \leftarrow B_0$. Can set $B \leftarrow I$ if no initial estimate; $k \leftarrow 0$; Pick a random starting point θ_0

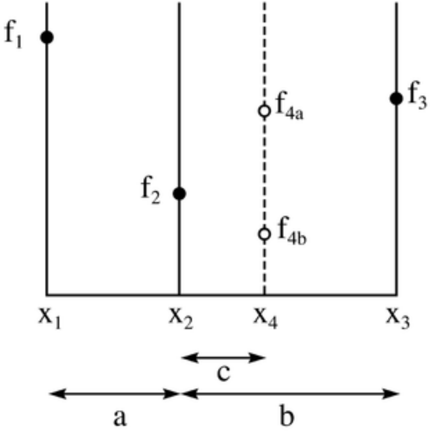
Loop

1. Get search direction $d_k = -B_k \nabla J(\theta_k)$
2. Conduct line search to find optimum $\theta_{k+1} = \theta_k + \alpha_k d_k$
3. $p_k \leftarrow \theta_{k+1} - \theta_k$; $q_k \leftarrow \nabla J(\theta_{k+1}) - \nabla J(\theta_k)$;
   $B_{k+1} = \left(I - \frac{p_k q_k^T}{q_k^T p_k}\right) B_k \left(I - \frac{q_k p_k^T}{q_k^T p_k}\right) + \frac{p_k p_k^T}{q_k^T p_k}$
4. $k \leftarrow k+1$; Exit if $\|\nabla J(\theta_k)\| < \epsilon$

Golden-section search

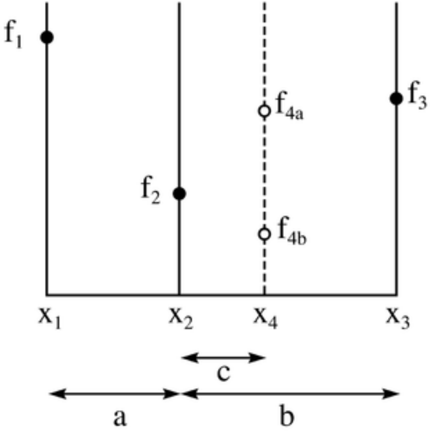


Golden-section search



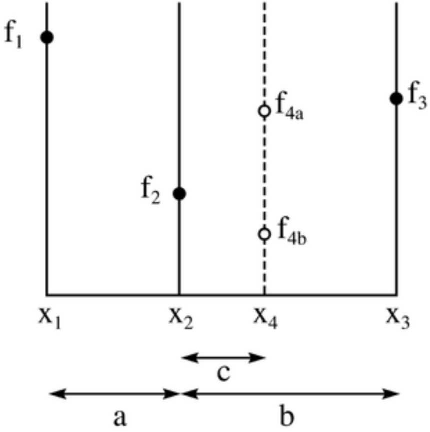
- If we have f_{4a}, minimum is in $[x_1, x_4]$

Golden-section search



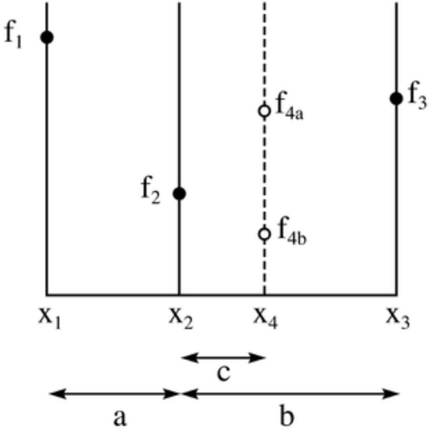
- If we have f_{4a}, minimum is in $[x_1, x_4]$
- If we have f_{4b}, minimum is in $[x_2, x_3]$

Golden-section search



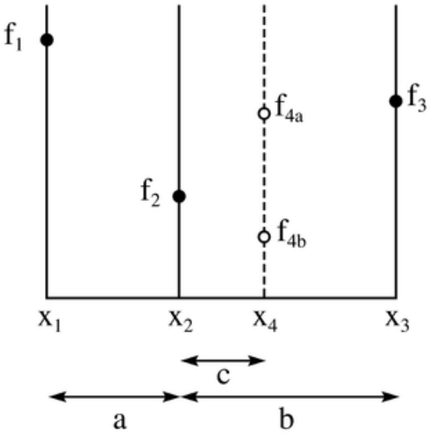
- If we have f_{4a}, minimum is in $[x_1, x_4]$
- If we have f_{4b}, minimum is in $[x_2, x_3]$
- To maximize expected search speed, set $x_4 - x_1 = x_3 - x_2 \Rightarrow a + c = b$

Golden-section search



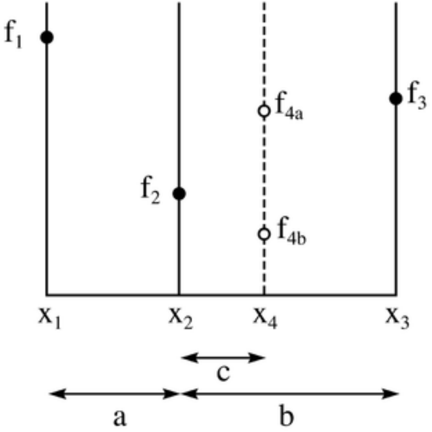
- If we have f_{4a}, minimum is in $[x_1, x_4]$
- If we have f_{4b}, minimum is in $[x_2, x_3]$
- To maximize expected search speed, set $x_4 - x_1 = x_3 - x_2 \Rightarrow a + c = b$
  - Given x_1, x_2, x_3, we know how to pick x_4

Golden-section search



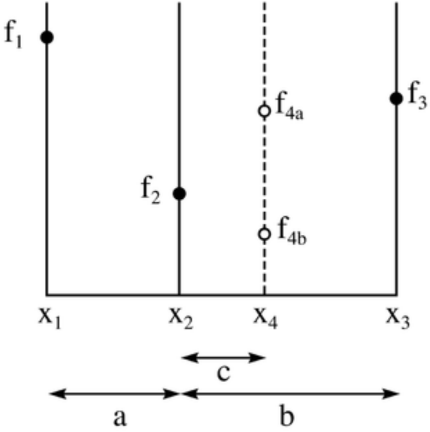
- If we have f_{4a}, minimum is in $[x_1, x_4]$
- If we have f_{4b}, minimum is in $[x_2, x_3]$
- To maximize expected search speed, set $x_4 - x_1 = x_3 - x_2 \Rightarrow a + c = b$
  - Given x_1, x_2, x_3, we know how to pick x_4
  - How to pick x_2 given x_1 and x_3?

Golden-section search



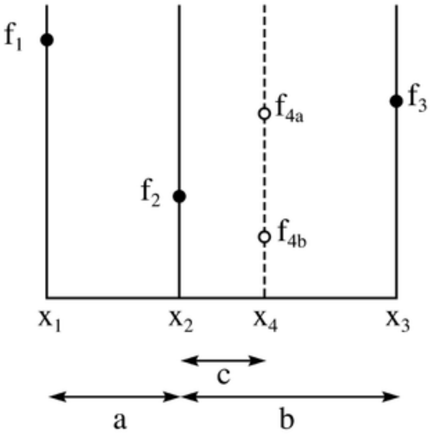
- If we have f_{4a}, minimum is in $[x_1, x_4]$
- If we have f_{4b}, minimum is in $[x_2, x_3]$
- To maximize expected search speed, set $x_4 - x_1 = x_3 - x_2 \Rightarrow a + c = b$
  - Given x_1, x_2, x_3, we know how to pick x_4
  - How to pick x_2 given x_1 and x_3?
- Golden-section search simply assume the "spacing" of each iteration is proportional

Golden-section search



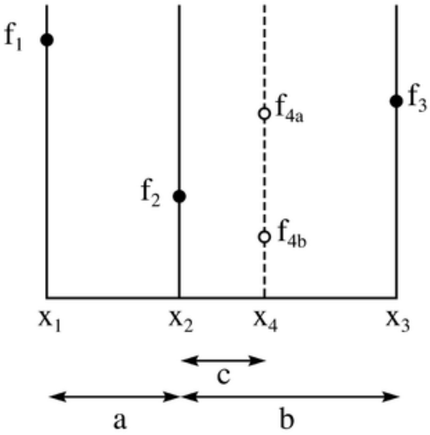
- If we have f_{4a}, minimum is in $[x_1, x_4]$
- If we have f_{4b}, minimum is in $[x_2, x_3]$
- To maximize expected search speed, set $x_4 - x_1 = x_3 - x_2 \Rightarrow a + c = b$
  - Given x_1, x_2, x_3, we know how to pick x_4
  - How to pick x_2 given x_1 and x_3?
- Golden-section search simply assume the "spacing" of each iteration is proportional
  - That is, $\frac{c}{a} = \frac{a}{b}$

Golden-section search



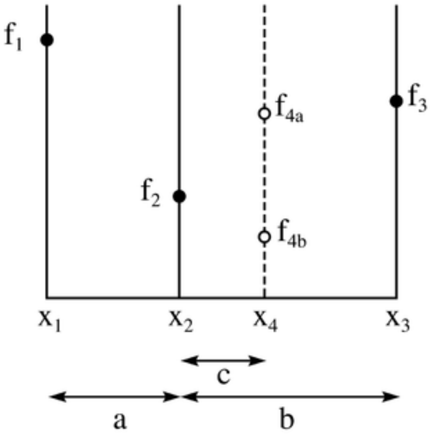
- $a + c = b$ and $\frac{c}{a} = \frac{a}{b}$

Golden-section search



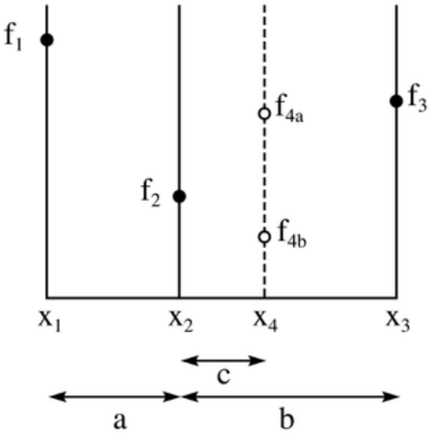
- $a + c = b$ and $\frac{c}{a} = \frac{a}{b}$
  $\Rightarrow \frac{b-a}{a} = \frac{a}{b}$

Golden-section search



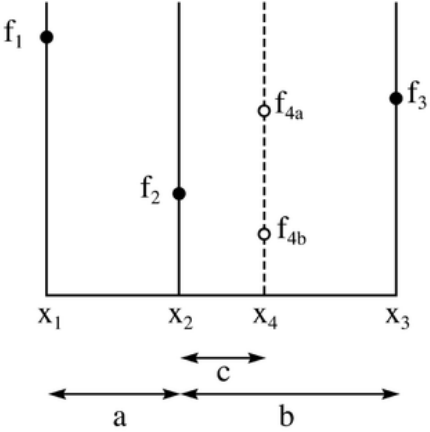
- $a + c = b$ and $\frac{c}{a} = \frac{a}{b}$
  $\Rightarrow \frac{b-a}{a} = \frac{a}{b}$
  $\Rightarrow \frac{b}{a} - 1 = \frac{1}{b/a}$

Golden-section search



- $a + c = b$ and $\frac{c}{a} = \frac{a}{b}$
  $\Rightarrow \frac{b-a}{a} = \frac{a}{b}$
  $\Rightarrow \frac{b}{a} - 1 = \frac{1}{b/a}$
  $\Rightarrow \left(\frac{b}{a}\right)^2 - \frac{b}{a} - 1 = 0$

Golden-section search



- $a + c = b$ and $\frac{c}{a} = \frac{a}{b}$
  $\Rightarrow \frac{b-a}{a} = \frac{a}{b}$
  $\Rightarrow \frac{b}{a} - 1 = \frac{1}{b/a}$
  $\Rightarrow \left(\frac{b}{a}\right)^2 - \frac{b}{a} - 1 = 0$

$$\frac{b}{a} = \frac{1+\sqrt{5}}{2} = 1.618034\ldots \triangleq \underset{\substack{\uparrow \\ \textit{golden} \\ \textit{ratio}}}{\varphi}$$

Inverse Hessian update for BFGS

- Like rank-1 update, we can also rearrange the variables to obtain an update rule for $B = H^{-1}$
- Instead of $H_{k+1}p_k = q_k$, we want $B_{k+1}q_k = p_k$.

Inverse Hessian update for BFGS

- Like rank-1 update, we can also rearrange the variables to obtain an update rule for $B = H^{-1}$
- Instead of $H_{k+1}p_k = q_k$, we want $B_{k+1}q_k = p_k$. Thus we have

$$B_{k+1} = B_k + \frac{p_k p_k^T}{p_k^T q_k} - \frac{B_k q_k q_k^T B_k}{q_k^T B_k^T q_k}$$

- Note that this update rule of B is different from before. Actually this is the update rule of DFP. An older approach that is considered worse compared with BFGS

Some theoretical notes

- A prettier but more technical explanation of BFGS/DFP involves weighted matrix norm

Some theoretical notes

- A prettier but more technical explanation of BFGS/DFP involves weighted matrix norm
- Comparing with rank-1 update, we have more degree of freedom and thus can impose more requirement. Besides
  1. $B_{k+1}q_k = p_k$ (secant equation)
  2. $B_{k+1} \succ 0$ (symmetric and positive definite),

  we also require each update to be small.

Some theoretical notes

- A prettier but more technical explanation of BFGS/DFP involves weighted matrix norm
- Comparing with rank-1 update, we have more degree of freedom and thus can impose more requirement. Besides
  1. $B_{k+1}q_k = p_k$ (secant equation)
  2. $B_{k+1} \succ 0$ (symmetric and positive definite),

  we also require each update to be small. Namely,

$$\|B_{k+1} - B_k\|_W \to \min,$$

  where $\|A\|_W = \|W^{1/2}AW^{1/2}\|_F$ is the weighted Frobenius norm

Some theoretical notes

- A prettier but more technical explanation of BFGS/DFP involves weighted matrix norm
- Comparing with rank-1 update, we have more degree of freedom and thus can impose more requirement. Besides
  1. $B_{k+1}q_k = p_k$ (secant equation)
  2. $B_{k+1} \succ 0$ (symmetric and positive definite),

  we also require each update to be small. Namely,

$$\|B_{k+1} - B_k\|_W \to \min,$$

  where $\|A\|_W = \|W^{1/2} A W^{1/2}\|_F$ is the weighted Frobenius norm
- $\Rightarrow \begin{cases} \text{BFGS} & W = H \\ \text{DFP} & W = H^{-1} \end{cases}$

LBFGS

- BFGS requires us to store the complete estimate of the Hessian or inverse Hessian

LBFGS

- BFGS requires us to store the complete estimate of the Hessian or inverse Hessian
- The matrix is too big to be stored in deep learning setting (millions of variables)

LBFGS

- BFGS requires us to store the complete estimate of the Hessian or inverse Hessian
- The matrix is too big to be stored in deep learning setting (millions of variables)
- Recall that $B_{k+1} = \left(I - \frac{p_k q_k^T}{q_k^T p_k}\right) B_k \left(I - \frac{q_k p_k^T}{q_k^T p_k}\right) + \frac{p_k p_k^T}{q_k^T p_k}$, size of p_k and q_k are much smaller

LBFGS

- BFGS requires us to store the complete estimate of the Hessian or inverse Hessian
- The matrix is too big to be stored in deep learning setting (millions of variables)
- Recall that $B_{k+1} = \left(I - \frac{p_k q_k^T}{q_k^T p_k}\right) B_k \left(I - \frac{q_k p_k^T}{q_k^T p_k}\right) + \frac{p_k p_k^T}{q_k^T p_k}$, size of p_k and q_k are much smaller
- Instead of storing B_k, we can store the previous last several p and q to estimate B_{k+1}

LBFGS

- BFGS requires us to store the complete estimate of the Hessian or inverse Hessian
- The matrix is too big to be stored in deep learning setting (millions of variables)
- Recall that $B_{k+1} = \left(I - \frac{p_k q_k^T}{q_k^T p_k}\right) B_k \left(I - \frac{q_k p_k^T}{q_k^T p_k}\right) + \frac{p_k p_k^T}{q_k^T p_k}$, size of p_k and q_k are much smaller
- Instead of storing B_k, we can store the previous last several p and q to estimate B_{k+1}
    - Let say we store the last r pairs, we need to iterate r times (instead of just once) and the estimate is less accurate

LBFGS

- BFGS requires us to store the complete estimate of the Hessian or inverse Hessian
- The matrix is too big to be stored in deep learning setting (millions of variables)
- Recall that $B_{k+1} = \left(I - \frac{p_k q_k^T}{q_k^T p_k}\right) B_k \left(I - \frac{q_k p_k^T}{q_k^T p_k}\right) + \frac{p_k p_k^T}{q_k^T p_k}$, size of p_k and q_k are much smaller
- Instead of storing B_k, we can store the previous last several p and q to estimate B_{k+1}
  - Let say we store the last r pairs, we need to iterate r times (instead of just once) and the estimate is less accurate
  - Storage requirement decreases drastically

LBFGS

- BFGS requires us to store the complete estimate of the Hessian or inverse Hessian
- The matrix is too big to be stored in deep learning setting (millions of variables)
- Recall that $B_{k+1} = \left(I - \frac{p_k q_k^T}{q_k^T p_k}\right) B_k \left(I - \frac{q_k p_k^T}{q_k^T p_k}\right) + \frac{p_k p_k^T}{q_k^T p_k}$, size of p_k and q_k are much smaller
- Instead of storing B_k, we can store the previous last several p and q to estimate B_{k+1}
  - Let say we store the last r pairs, we need to iterate r times (instead of just once) and the estimate is less accurate
  - Storage requirement decreases drastically
- LBFGS works very well in full batch, function is more or less deterministic

LBFGS

- BFGS requires us to store the complete estimate of the Hessian or inverse Hessian
- The matrix is too big to be stored in deep learning setting (millions of variables)
- Recall that $B_{k+1} = \left(I - \frac{p_k q_k^T}{q_k^T p_k}\right) B_k \left(I - \frac{q_k p_k^T}{q_k^T p_k}\right) + \frac{p_k p_k^T}{q_k^T p_k}$, size of p_k and q_k are much smaller
- Instead of storing B_k, we can store the previous last several p and q to estimate B_{k+1}
  - Let say we store the last r pairs, we need to iterate r times (instead of just once) and the estimate is less accurate
  - Storage requirement decreases drastically
- LBFGS works very well in full batch, function is more or less deterministic
  - But does not seem to work very well to mini-batch setting

Summary

- ADAM is a good default choice in most cases

Summary

- ADAM is a good default choice in most cases
  - Some reported that Momentum SGD works better for ResNet, where some contested that they can have sharp minimum (see this)

Summary

- ADAM is a good default choice in most cases
  - Some reported that Momentum SGD works better for ResNet, where some contested that they can have sharp minimum (see this)
  - If you worry about stucking in local minimum, you may set amsgrad to True, that try to prevent ADAM from getting stuck (see this)

Summary

- ADAM is a good default choice in most cases
  - Some reported that Momentum SGD works better for ResNet, where some contested that they can have sharp minimum (see this)
  - If you worry about stucking in local minimum, you may set amsgrad to True, that try to prevent ADAM from getting stuck (see this)
- Learning rate depends on implementations. One has to be careful to transfer that from one package to another

Summary

- ADAM is a good default choice in most cases
  - Some reported that Momentum SGD works better for ResNet, where some contested that they can have sharp minimum (see this)
  - If you worry about stucking in local minimum, you may set amsgrad to True, that try to prevent ADAM from getting stuck (see this)
- Learning rate depends on implementations. One has to be careful to transfer that from one package to another
  - LR for SGD with momentum for PyTorch is effectively $\frac{1}{1-\mu}$ more than original Sutskever's or SGD implementation

Summary

- ADAM is a good default choice in most cases
  - Some reported that Momentum SGD works better for ResNet, where some contested that they can have sharp minimum (see this)
  - If you worry about stucking in local minimum, you may set amsgrad to True, that try to prevent ADAM from getting stuck (see this)
- Learning rate depends on implementations. One has to be careful to transfer that from one package to another
  - LR for SGD with momentum for PyTorch is effectively $\frac{1}{1-\mu}$ more than original Sutskever's or SGD implementation
  - E.g., if SGD works well with LR 1, you may want to change LR to 0.1 if a momentum $\mu = 0.9$ is applied

Summary

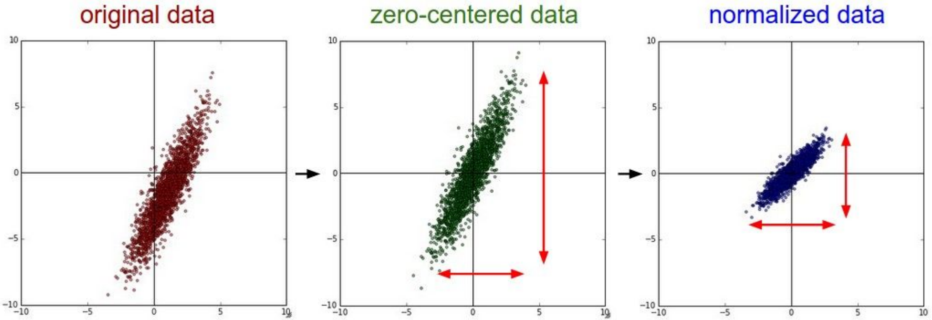
- ADAM is a good default choice in most cases
  - Some reported that Momentum SGD works better for ResNet, where some contested that they can have sharp minimum (see this)
  - If you worry about stucking in local minimum, you may set amsgrad to True, that try to prevent ADAM from getting stuck (see this)
- Learning rate depends on implementations. One has to be careful to transfer that from one package to another
  - LR for SGD with momentum for PyTorch is effectively $\frac{1}{1-\mu}$ more than original Sutskever's or SGD implementation
  - E.g., if SGD works well with LR 1, you may want to change LR to 0.1 if a momentum $\mu = 0.9$ is applied
- Many more parameters besides LR, e.g., weight decay (L2 penalty). Check doc

Summary

- ADAM is a good default choice in most cases
  - Some reported that Momentum SGD works better for ResNet, where some contested that they can have sharp minimum (see this)
  - If you worry about stucking in local minimum, you may set amsgrad to True, that try to prevent ADAM from getting stuck (see this)
- Learning rate depends on implementations. One has to be careful to transfer that from one package to another
  - LR for SGD with momentum for PyTorch is effectively $\frac{1}{1-\mu}$ more than original Sutskever's or SGD implementation
  - E.g., if SGD works well with LR 1, you may want to change LR to 0.1 if a momentum $\mu = 0.9$ is applied
- Many more parameters besides LR, e.g., weight decay (L2 penalty). Check doc
- For nearly deterministic objective function (full-batch), one may try to use LBFGS as well. But it probably needs too much computational resources in most applications (a few exception can be style transfer)

Babysitting learning process

Step 1: Preprocess the data




```
X -= np.mean(X, axis = 0)
```

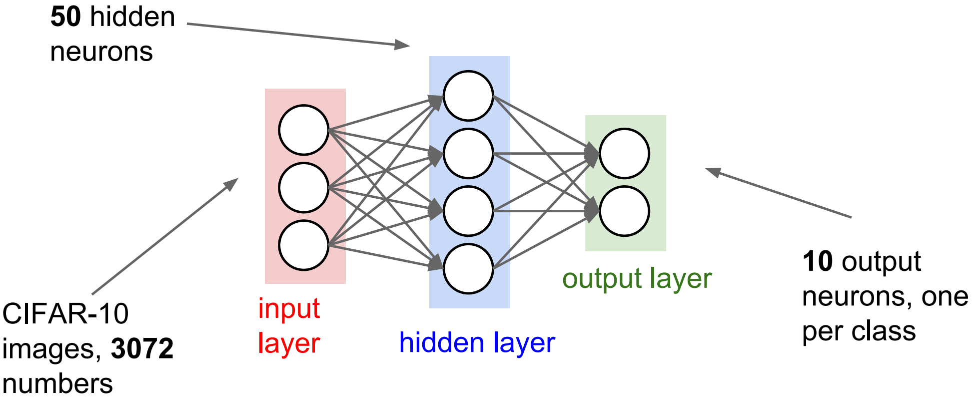
```
X /= np.std(X, axis = 0)
```

(Assume X [NxD] is data matrix, each example in a row)

Babysitting learning process

Step 2: Choose the architecture:

say we start with one hidden layer of 50 neurons:



Babysitting learning process

Double check that the loss is reasonable:

```
def init_two_layer_model(input_size, hidden_size, output_size):
  # initialize a model
  model = {}
  model['W1'] = 0.0001 * np.random.randn(input_size, hidden_size)
  model['b1'] = np.zeros(hidden_size)
  model['W2'] = 0.0001 * np.random.randn(hidden_size, output_size)
  model['b2'] = np.zeros(output_size)
  return model
```

```
model = init_two_layer_model(32*32*3, 50, 10) # input size, hidden size, number of classes
loss, grad = two_layer_net(X_train, model, y_train, 0.0)
print loss
```

```
2.30261216167
```

disable regularization

loss ~2.3.
"correct " for
10 classes

returns the loss and the
gradient for all parameters

Debugging optimizer

Double check that the loss is reasonable:

```
def init_two_layer_model(input_size, hidden_size, output_size):
  # initialize a model
  model = {}
  model['W1'] = 0.0001 * np.random.randn(input_size, hidden_size)
  model['b1'] = np.zeros(hidden_size)
  model['W2'] = 0.0001 * np.random.randn(hidden_size, output_size)
  model['b2'] = np.zeros(output_size)
  return model
```

```
model = init_two_layer_model(32*32*3, 50, 10) # input_size, hidden size, number of classes
loss, grad = two_layer_net(X_train, model, y_train, 1e3)
print loss
```

```
3.06859716482
```

crank up regularization

loss went up, good. (sanity check)

Debugging optimizer

Lets try to train now…

Tip: Make sure that you can overfit very small portion of the training data

```
model = init_two_layer_model(32*32*3, 50, 10) # input size, hidden size, number of classes
trainer = ClassifierTrainer()
X_tiny = X_train[:20] # take 20 examples
y_tiny = y_train[:20]
best_model, stats = trainer.train(X_tiny, y_tiny, X_tiny, y_tiny,
                                  model, two_layer_net,
                                  num_epochs=200, reg=0.0,
                                  update='sgd', learning_rate_decay=1,
                                  sample_batches = False,
                                  learning_rate=1e-3, verbose=True)
```

The above code:

- take the first 20 examples from CIFAR-10
- turn off regularization (reg = 0.0)
- use simple vanilla ‘sgd’

Debugging optimizer

Lets try to train now…

Tip: Make sure that you can overfit very small portion of the training data

Very small loss, train accuracy 1.00, nice!

```
model = init_two_layer_model(32*32*3, 50, 10) # input size, hidden size, number of classes
trainer = ClassifierTrainer()
X_tiny = X_train[:20] # take 20 examples
y_tiny = y_train[:20]
best_model, stats = trainer.train(X_tiny, y_tiny, X_tiny, y_tiny,
                                  model, two_layer_net,
                                  num_epochs=200, reg=0.0,
                                  update='sgd', learning_rate_decay=1,
                                  sample_batches = False,
                                  learning_rate=1e-3, verbose=True)
```

```
Finished epoch 1 / 200: cost 2.302603, train: 0.400000, val 0.400000, lr 1.000000e-03
Finished epoch 2 / 200: cost 2.302258, train: 0.450000, val 0.450000, lr 1.000000e-03
Finished epoch 3 / 200: cost 2.301849, train: 0.600000, val 0.600000, lr 1.000000e-03
Finished epoch 4 / 200: cost 2.301196, train: 0.650000, val 0.650000, lr 1.000000e-03
Finished epoch 5 / 200: cost 2.300044, train: 0.650000, val 0.650000, lr 1.000000e-03
Finished epoch 6 / 200: cost 2.297864, train: 0.550000, val 0.550000, lr 1.000000e-03
Finished epoch 7 / 200: cost 2.293595, train: 0.600000, val 0.600000, lr 1.000000e-03
Finished epoch 8 / 200: cost 2.285096, train: 0.550000, val 0.550000, lr 1.000000e-03
Finished epoch 9 / 200: cost 2.268094, train: 0.550000, val 0.550000, lr 1.000000e-03
Finished epoch 10 / 200: cost 2.234787, train: 0.500000, val 0.500000, lr 1.000000e-03
Finished epoch 11 / 200: cost 2.173187, train: 0.500000, val 0.500000, lr 1.000000e-03
Finished epoch 12 / 200: cost 2.076862, train: 0.500000, val 0.500000, lr 1.000000e-03
Finished epoch 13 / 200: cost 1.974090, train: 0.400000, val 0.400000, lr 1.000000e-03
Finished epoch 14 / 200: cost 1.895885, train: 0.400000, val 0.400000, lr 1.000000e-03
Finished epoch 15 / 200: cost 1.820876, train: 0.450000, val 0.450000, lr 1.000000e-03
Finished epoch 16 / 200: cost 1.737430, train: 0.450000, val 0.450000, lr 1.000000e-03
Finished epoch 17 / 200: cost 1.642356, train: 0.500000, val 0.500000, lr 1.000000e-03
Finished epoch 18 / 200: cost 1.535239, train: 0.600000, val 0.600000, lr 1.000000e-03
Finished epoch 19 / 200: cost 1.421527, train: 0.600000, val 0.600000, lr 1.000000e-03
```

```
Finished epoch 195 / 200: cost 0.002694, train: 1.000000, val 1.000000, lr 1.000000e-03
Finished epoch 196 / 200: cost 0.002674, train: 1.000000, val 1.000000, lr 1.000000e-03
Finished epoch 197 / 200: cost 0.002655, train: 1.000000, val 1.000000, lr 1.000000e-03
Finished epoch 198 / 200: cost 0.002635, train: 1.000000, val 1.000000, lr 1.000000e-03
Finished epoch 199 / 200: cost 0.002617, train: 1.000000, val 1.000000, lr 1.000000e-03
Finished epoch 200 / 200: cost 0.002597, train: 1.000000, val 1.000000, lr 1.000000e-03
finished optimization. best validation accuracy: 1.000000
```

Debugging optimizer

Lets try to train now…

I like to start with small regularization and find learning rate that makes the loss go down.

```
model = init_two_layer_model(32*32*3, 50, 10) # input size, hidden size, number of classes
trainer = ClassifierTrainer()
best_model, stats = trainer.train(X_train, y_train, X_val, y_val,
                                  model, two_layer_net,
                                  num_epochs=10, reg=0.000001,
                                  update='sgd', learning_rate_decay=1,
                                  sample_batches = True,
                                  learning_rate=1e-6, verbose=True)
```

Debugging optimizer

Lets try to train now…

I like to start with small regularization and find learning rate that makes the loss go down.

```
model = init_two_layer_model(32*32*3, 50, 10) # input size, hidden size, number of classes
trainer = ClassifierTrainer()
best_model, stats = trainer.train(X_train, y_train, X_val, y_val,
                                  model, two_layer_net,
                                  num_epochs=10, reg=0.000001,
                                  update='sgd', learning_rate_decay=1,
                                  sample_batches = True,
                                  learning_rate=1e-6, verbose=True)
```

```
Finished epoch 1 / 10: cost 2.302576, train: 0.080000, val 0.103000, lr 1.000000e-06
Finished epoch 2 / 10: cost 2.302582, train: 0.121000, val 0.124000, lr 1.000000e-06
Finished epoch 3 / 10: cost 2.302558, train: 0.119000, val 0.138000, lr 1.000000e-06
Finished epoch 4 / 10: cost 2.302519, train: 0.127000, val 0.151000, lr 1.000000e-06
Finished epoch 5 / 10: cost 2.302517, train: 0.158000, val 0.171000, lr 1.000000e-06
Finished epoch 6 / 10: cost 2.302518, train: 0.179000, val 0.172000, lr 1.000000e-06
Finished epoch 7 / 10: cost 2.302466, train: 0.180000, val 0.176000, lr 1.000000e-06
Finished epoch 8 / 10: cost 2.302452, train: 0.175000, val 0.185000, lr 1.000000e-06
Finished epoch 9 / 10: cost 2.302459, train: 0.206000, val 0.192000, lr 1.000000e-06
Finished epoch 10 / 10: cost 2.302420, train: 0.190000, val 0.192000, lr 1.000000e-06
finished optimization. best validation accuracy: 0.192000
```

Loss barely changing

Debugging optimizer

Lets try to train now…

I like to start with small regularization and find learning rate that makes the loss go down.

loss not going down: learning rate too low

```
model = init_two_layer_model(32*32*3, 50, 10) # input size, hidden size, number of classes
trainer = ClassifierTrainer()
best_model, stats = trainer.train(X_train, y_train, X_val, y_val,
                                  model, two_layer_net,
                                  num_epochs=10, reg=0.000001,
                                  update='sgd', learning_rate_decay=1,
                                  sample_batches = True,
                                  learning_rate=1e-6, verbose=True)
```

```
Finished epoch 1 / 10: cost 2.302576, train: 0.080000, val 0.103000, lr 1.000000e-06
Finished epoch 2 / 10: cost 2.302582, train: 0.121000, val 0.124000, lr 1.000000e-06
Finished epoch 3 / 10: cost 2.302558, train: 0.119000, val 0.138000, lr 1.000000e-06
Finished epoch 4 / 10: cost 2.302519, train: 0.127000, val 0.151000, lr 1.000000e-06
Finished epoch 5 / 10: cost 2.302517, train: 0.158000, val 0.171000, lr 1.000000e-06
Finished epoch 6 / 10: cost 2.302518, train: 0.179000, val 0.172000, lr 1.000000e-06
Finished epoch 7 / 10: cost 2.302466, train: 0.180000, val 0.176000, lr 1.000000e-06
Finished epoch 8 / 10: cost 2.302452, train: 0.175000, val 0.185000, lr 1.000000e-06
Finished epoch 9 / 10: cost 2.302459, train: 0.206000, val 0.192000, lr 1.000000e-06
Finished epoch 10 / 10: cost 2.302420, train: 0.190000, val 0.192000, lr 1.000000e-06
finished optimization. best validation accuracy: 0.192000
```

Loss barely changing: Learning rate is probably too low

Debugging optimizer

Lets try to train now…

I like to start with small regularization and find learning rate that makes the loss go down.

loss not going down:
learning rate too low

```
model = init_two_layer_model(32*32*3, 50, 10) # input size, hidden size, number of classes
trainer = ClassifierTrainer()
best_model, stats = trainer.train(X_train, y_train, X_val, y_val,
                                  model, two_layer_net,
                                  num_epochs=10, reg=0.000001,
                                  update='sgd', learning_rate_decay=1,
                                  sample_batches = True,
                                  learning_rate=1e-6, verbose=True)
```

```
Finished epoch 1 / 10: cost 2.302576, train: 0.080000, val 0.103000, lr 1.000000e-06
Finished epoch 2 / 10: cost 2.302582, train: 0.121000, val 0.124000, lr 1.000000e-06
Finished epoch 3 / 10: cost 2.302558, train: 0.119000, val 0.138000, lr 1.000000e-06
Finished epoch 4 / 10: cost 2.302519, train: 0.127000, val 0.151000, lr 1.000000e-06
Finished epoch 5 / 10: cost 2.302517, train: 0.158000, val 0.171000, lr 1.000000e-06
Finished epoch 6 / 10: cost 2.302518, train: 0.179000, val 0.172000, lr 1.000000e-06
Finished epoch 7 / 10: cost 2.302466, train: 0.180000, val 0.176000, lr 1.000000e-06
Finished epoch 8 / 10: cost 2.302452, train: 0.175000, val 0.185000, lr 1.000000e-06
Finished epoch 9 / 10: cost 2.302459, train: 0.206000, val 0.192000, lr 1.000000e-06
Finished epoch 10 / 10: cost 2.302420, train: 0.190000, val 0.192000, lr 1.000000e-06
finished optimization. best validation accuracy: 0.192000
```

Loss barely changing: Learning rate is probably too low

Notice train/val accuracy goes to 20% though, what's up with that? (remember this is softmax)

Debugging optimizer

Lets try to train now…

I like to start with small regularization and find learning rate that makes the loss go down.

loss not going down:
learning rate too low

```
model = init_two_layer_model(32*32*3, 50, 10) # input size, hidden size, number of classes
trainer = ClassifierTrainer()
best_model, stats = trainer.train(X_train, y_train, X_val, y_val,
                                  model, two_layer_net,
                                  num_epochs=10, reg=0.000001,
                                  update='sgd', learning_rate_decay=1,
                                  sample_batches = True,
                                  learning_rate=1e6, verbose=True)
```

Okay now lets try learning rate 1e6. What could possibly go wrong?

Debugging optimizer

Lets try to train now…

I like to start with small regularization and find learning rate that makes the loss go down.

loss not going down: learning rate too low
loss exploding: learning rate too high

```
model = init_two_layer_model(32*32*3, 50, 10) # input size, hidden size, number of classes
trainer = ClassifierTrainer()
best_model, stats = trainer.train(X_train, y_train, X_val, y_val,
                                  model, two_layer_net,
                                  num_epochs=10, reg=0.000001,
                                  update='sgd', learning_rate_decay=1,
                                  sample_batches = True,
                                  learning_rate=1e6, verbose=True)
```

```
/home/karpathy/cs231n/code/cs231n/classifiers/neural_net.py:50: RuntimeWarning: divide by zero en
countered in log
  data_loss = -np.sum(np.log(probs[range(N), y])) / N
/home/karpathy/cs231n/code/cs231n/classifiers/neural_net.py:48: RuntimeWarning: invalid value enc
ountered in subtract
  probs = np.exp(scores - np.max(scores, axis=1, keepdims=True))
Finished epoch 1 / 10: cost nan, train: 0.091000, val 0.087000, lr 1.000000e+06
Finished epoch 2 / 10: cost nan, train: 0.095000, val 0.087000, lr 1.000000e+06
Finished epoch 3 / 10: cost nan, train: 0.100000, val 0.087000, lr 1.000000e+06
```

cost: NaN almost always means high learning rate...

Debugging optimizer

Lets try to train now…

I like to start with small regularization and find learning rate that makes the loss go down.

loss not going down:
learning rate too low
loss exploding:
learning rate too high

```
model = init_two_layer_model(32*32*3, 50, 10) # input size, hidden size, number of classes
trainer = ClassifierTrainer()
best_model, stats = trainer.train(X_train, y_train, X_val, y_val,
                                  model, two_layer_net,
                                  num_epochs=10, reg=0.000001,
                                  update='sgd', learning_rate_decay=1,
                                  sample_batches = True,
                                  learning_rate=3e-3, verbose=True)
```

```
Finished epoch 1 / 10: cost 2.186654, train: 0.308000, val 0.306000, lr 3.000000e-03
Finished epoch 2 / 10: cost 2.176230, train: 0.330000, val 0.350000, lr 3.000000e-03
Finished epoch 3 / 10: cost 1.942257, train: 0.376000, val 0.352000, lr 3.000000e-03
Finished epoch 4 / 10: cost 1.827868, train: 0.329000, val 0.310000, lr 3.000000e-03
Finished epoch 5 / 10: cost inf, train: 0.128000, val 0.128000, lr 3.000000e-03
Finished epoch 6 / 10: cost inf, train: 0.144000, val 0.147000, lr 3.000000e-03
```

3e-3 is still too high. Cost explodes….

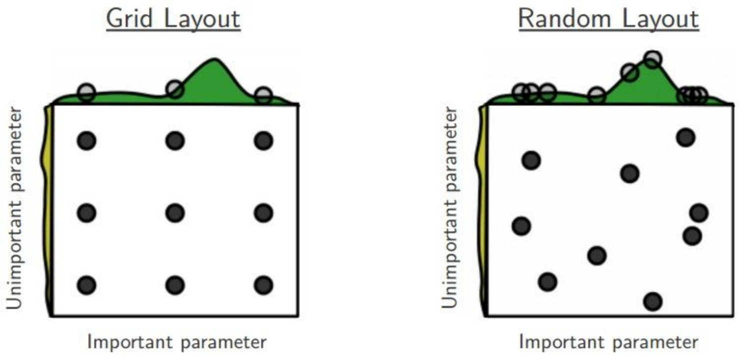
=> Rough range for learning rate we should be cross-validating is somewhere [1e-3 … 1e-5]

Hyperparameter optimization

Hyperparameter Optimization

Hyperparameter optimization

Random Search vs. Grid Search



Random Search for Hyper-Parameter Optimization
Bergstra and Bengio, 2012

Hyperparameter optimization

Cross-validation strategy

I like to do **coarse -> fine** cross-validation in stages

First stage: only a few epochs to get rough idea of what params work
Second stage: longer running time, finer search
… (repeat as necessary)

Tip for detecting explosions in the solver:
If the cost is ever > 3 * original cost, break out early

Hyperparameter optimization

For example: run coarse search for 5 epochs

```
max_count = 100
for count in xrange(max_count):
      reg = 10**uniform(-5, 5)
      lr = 10**uniform(-3, -6)

      trainer = ClassifierTrainer()
      model = init_two_layer_model(32*32*3, 50, 10) # input size, hidden size, number of classes
      trainer = ClassifierTrainer()
      best_model_local, stats = trainer.train(X_train, y_train, X_val, y_val,
                                   model, two_layer_net,
                                   num_epochs=5, reg=reg,
                                   update='momentum', learning_rate_decay=0.9,
                                   sample_batches = True, batch_size = 100,
                                   learning_rate=lr, verbose=False)
```

note it's best to optimize in log space!

```
val_acc: 0.412000, lr: 1.405206e-04, reg: 4.793564e-01, (1 / 100)
val_acc: 0.214000, lr: 7.231888e-06, reg: 2.321281e-04, (2 / 100)
val_acc: 0.208000, lr: 2.119571e-06, reg: 8.011857e+01, (3 / 100)
val_acc: 0.196000, lr: 1.551131e-05, reg: 4.374936e-05, (4 / 100)
val_acc: 0.079000, lr: 1.753300e-05, reg: 1.200424e+03, (5 / 100)
val_acc: 0.223000, lr: 4.215128e-05, reg: 4.196174e+01, (6 / 100)
val_acc: 0.441000, lr: 1.750259e-04, reg: 2.110807e-04, (7 / 100)
val_acc: 0.241000, lr: 6.749231e-05, reg: 4.226413e+01, (8 / 100)
val_acc: 0.482000, lr: 4.296863e-04, reg: 6.642555e-01, (9 / 100)
val_acc: 0.079000, lr: 5.401602e-06, reg: 1.599828e+04, (10 / 100)
val_acc: 0.154000, lr: 1.618508e-06, reg: 4.925252e-01, (11 / 100)
```

nice

Hyperparameter optimization

Now run finer search...

```
max_count = 100
for count in xrange(max_count):
      reg = 10**uniform(-5, 5)
      lr = 10**uniform(-3, -6)
```

adjust range →

```
max_count = 100
for count in xrange(max_count):
      reg = 10**uniform(-4, 0)
      lr = 10**uniform(-3, -4)
```

```
val_acc: 0.527000, lr: 5.340517e-04, reg: 4.097824e-01, (0 / 100)
val_acc: 0.492000, lr: 2.279484e-04, reg: 9.991345e-04, (1 / 100)
val_acc: 0.512000, lr: 8.680827e-04, reg: 1.349727e-02, (2 / 100)
val_acc: 0.461000, lr: 1.028377e-04, reg: 1.220193e-02, (3 / 100)
val_acc: 0.460000, lr: 1.113730e-04, reg: 5.244309e-02, (4 / 100)
val_acc: 0.498000, lr: 9.477776e-04, reg: 2.001293e-03, (5 / 100)
val_acc: 0.469000, lr: 1.484369e-04, reg: 4.328313e-01, (6 / 100)
val_acc: 0.522000, lr: 5.586261e-04, reg: 2.312685e-04, (7 / 100)
val_acc: 0.530000, lr: 5.808183e-04, reg: 8.259964e-02, (8 / 100)
val_acc: 0.489000, lr: 1.979168e-04, reg: 1.010889e-04, (9 / 100)
val_acc: 0.490000, lr: 2.036031e-04, reg: 2.406271e-03, (10 / 100)
val_acc: 0.475000, lr: 2.021162e-04, reg: 2.287807e-01, (11 / 100)
val_acc: 0.460000, lr: 1.135527e-04, reg: 3.905040e-02, (12 / 100)
val_acc: 0.515000, lr: 6.947668e-04, reg: 1.562808e-02, (13 / 100)
val_acc: 0.531000, lr: 9.471549e-04, reg: 1.433895e-03, (14 / 100)
val_acc: 0.509000, lr: 3.140888e-04, reg: 2.857518e-01, (15 / 100)
val_acc: 0.514000, lr: 6.438349e-04, reg: 3.033781e-01, (16 / 100)
val_acc: 0.502000, lr: 3.921784e-04, reg: 2.707126e-04, (17 / 100)
val_acc: 0.509000, lr: 9.752279e-04, reg: 2.850865e-03, (18 / 100)
val_acc: 0.500000, lr: 2.412048e-04, reg: 4.997821e-04, (19 / 100)
val_acc: 0.466000, lr: 1.319314e-04, reg: 1.189915e-02, (20 / 100)
val_acc: 0.516000, lr: 8.039527e-04, reg: 1.528291e-02, (21 / 100)
```

53% - relatively good for a 2-layer neural net with 50 hidden neurons.

Hyperparameter optimization

Now run finer search...

```
max_count = 100
for count in xrange(max_count):
      reg = 10**uniform(-5, 5)
      lr = 10**uniform(-3, -6)
```

adjust range →

```
max_count = 100
for count in xrange(max_count):
      reg = 10**uniform(-4, 0)
      lr = 10**uniform(-3, -4)
```

```
val_acc: 0.527000, lr: 5.340517e-04, reg: 4.097824e-01, (0 / 100)
val_acc: 0.492000, lr: 2.279484e-04, reg: 9.991345e-04, (1 / 100)
val_acc: 0.512000, lr: 8.680827e-04, reg: 1.349727e-02, (2 / 100)
val_acc: 0.461000, lr: 1.028377e-04, reg: 1.220193e-02, (3 / 100)
val_acc: 0.460000, lr: 1.113730e-04, reg: 5.244309e-02, (4 / 100)
val_acc: 0.498000, lr: 9.477776e-04, reg: 2.001293e-03, (5 / 100)
val_acc: 0.469000, lr: 1.484369e-04, reg: 4.328313e-01, (6 / 100)
val_acc: 0.522000, lr: 5.586261e-04, reg: 2.312685e-04, (7 / 100)
val_acc: 0.530000, lr: 5.808183e-04, reg: 8.259964e-02, (8 / 100)
val_acc: 0.489000, lr: 1.979168e-04, reg: 1.010889e-04, (9 / 100)
val_acc: 0.490000, lr: 2.036031e-04, reg: 2.406271e-03, (10 / 100)
val_acc: 0.475000, lr: 2.021162e-04, reg: 2.287807e-01, (11 / 100)
val_acc: 0.460000, lr: 1.135527e-04, reg: 3.905040e-02, (12 / 100)
val_acc: 0.515000, lr: 6.947668e-04, reg: 1.562808e-02, (13 / 100)
val_acc: 0.531000, lr: 9.471549e-04, reg: 1.433895e-03, (14 / 100)
val_acc: 0.509000, lr: 3.140888e-04, reg: 2.857518e-01, (15 / 100)
val_acc: 0.514000, lr: 6.438349e-04, reg: 3.033781e-01, (16 / 100)
val_acc: 0.502000, lr: 3.921784e-04, reg: 2.707126e-04, (17 / 100)
val_acc: 0.509000, lr: 9.752279e-04, reg: 2.850865e-03, (18 / 100)
val_acc: 0.500000, lr: 2.412048e-04, reg: 4.997821e-04, (19 / 100)
val_acc: 0.466000, lr: 1.319314e-04, reg: 1.189915e-02, (20 / 100)
val_acc: 0.516000, lr: 8.039527e-04, reg: 1.528291e-02, (21 / 100)
```

53% - relatively good for a 2-layer neural net with 50 hidden neurons.

But this best cross-validation result is worrying. Why?

Hyperparameter optimization

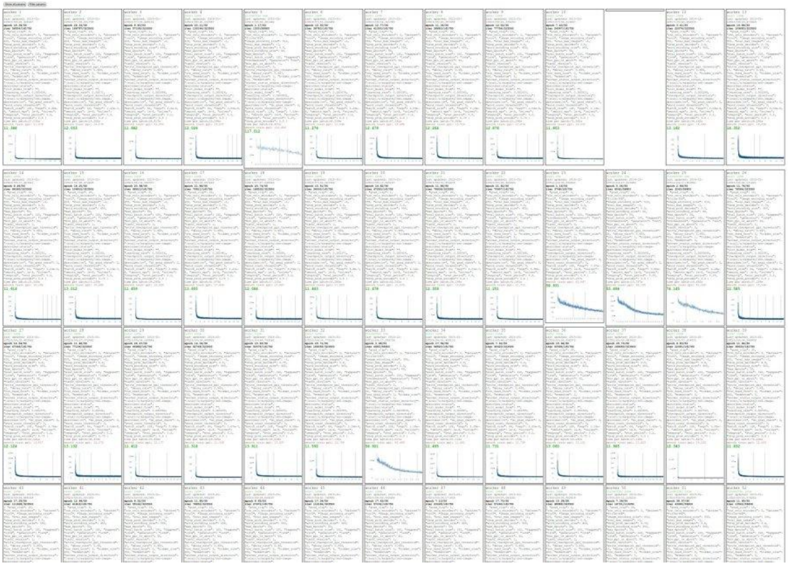
Hyperparameters to play with:

- network architecture
- learning rate, its decay schedule, update type
- regularization (L2/Dropout strength)

neural networks practitioner
music = loss function

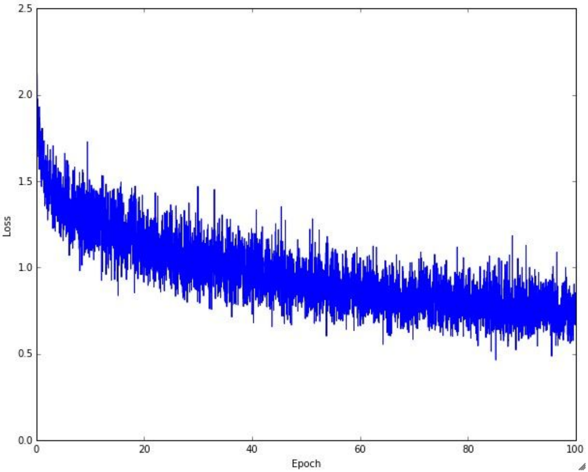
Hyperparameter optimization

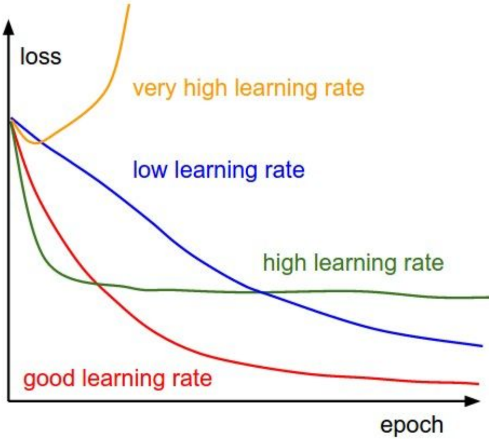
My cross-validation "command center"



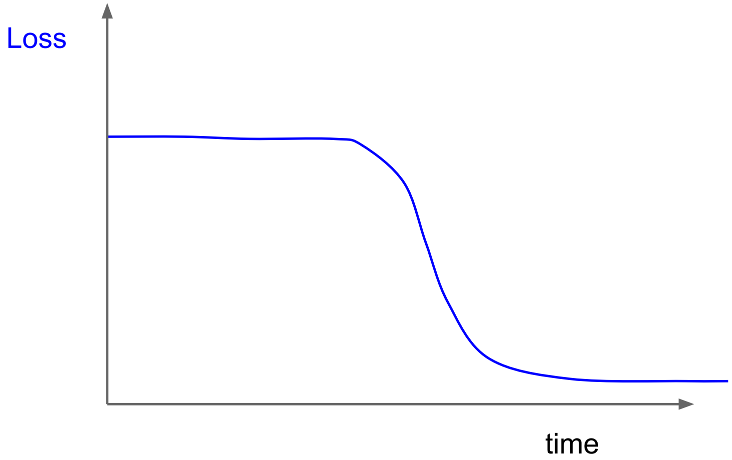
Hyperparameter optimization

Monitor and visualize the loss curve

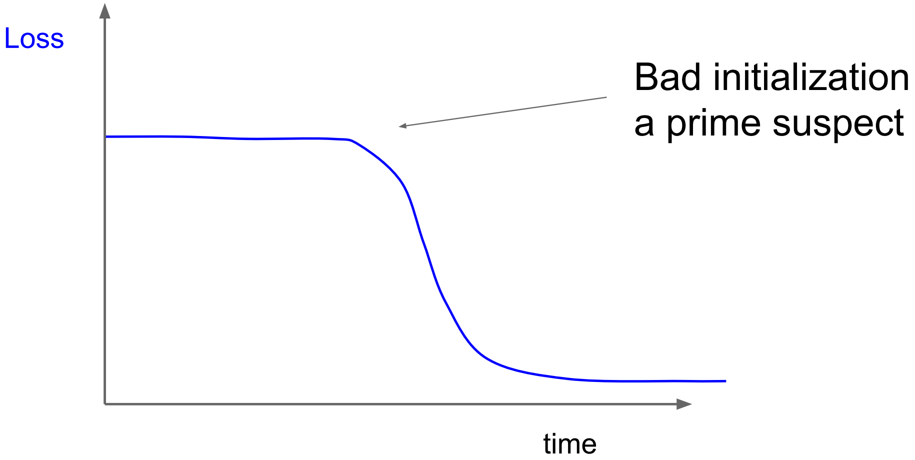




Hyperparameter optimization

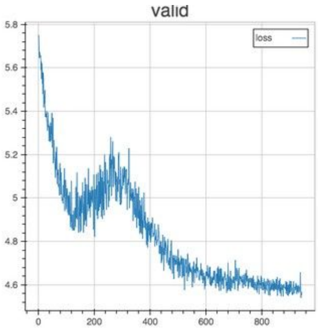


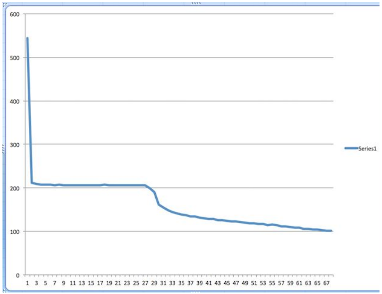
Hyperparameter optimization

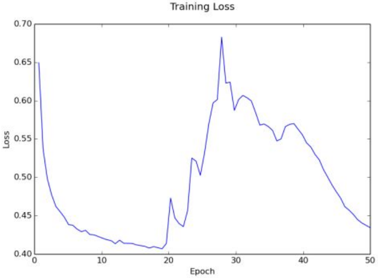


Hyperparameter optimization

lossfunctions.tumblr.com Loss function specimen

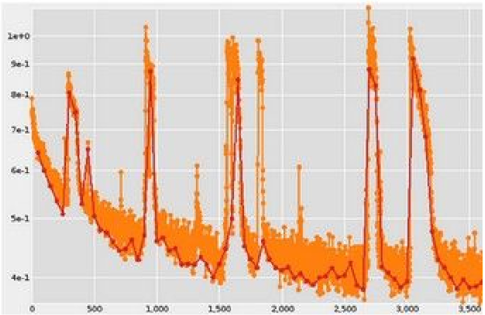


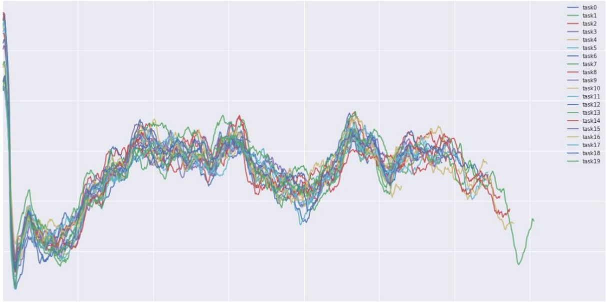




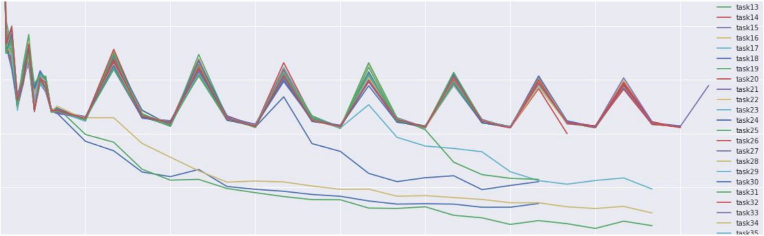
Hyperparameter optimization

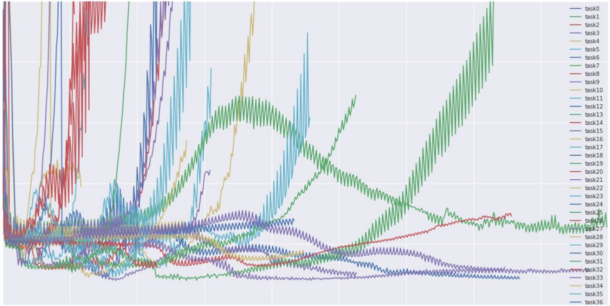
lossfunctions.tumblr.com





Hyperparameter optimization

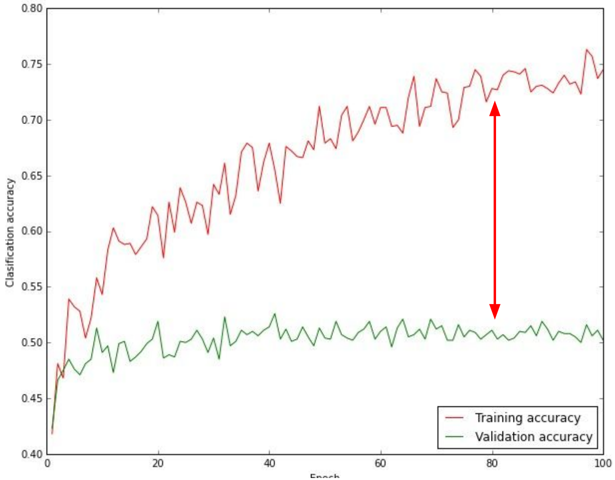




lossfunctions.tumblr.com

Hyperparameter optimization

Monitor and visualize the accuracy:



big gap = overfitting

=> increase regularization strength?

no gap

=> increase model capacity?

Hyperparameter optimization

Track the ratio of weight updates / weight magnitudes:

```
# assume parameter vector W and its gradient vector dW
param_scale = np.linalg.norm(W.ravel())
update = -learning_rate*dW # simple SGD update
update_scale = np.linalg.norm(update.ravel())
W += update # the actual update
print update_scale / param_scale # want ~1e-3
```

ratio between the values and updates: ~ 0.0002 / 0.02 = 0.01 (about okay)
want this to be somewhere around 0.001 or so

Conclusions

- BP is just chain rule in calculus
- Use ReLU. Never use Sigmoid (use Tanh instead)
- Input preprocessing is no longer very important
    - Do subtract mean
    - Whitening and normalizing are not much needed
- Weight initialization on the other hand is extremely important for deep networks
- Use batch normalization if you can
- Use dropout
- Use Adam (or maybe RMSprop) for optimizer. If you don't have much data, can consider LBFGS
- Need to babysit your learning for real-world problems
- Never use grid search for tuning your hyperparameters