Neural Networks

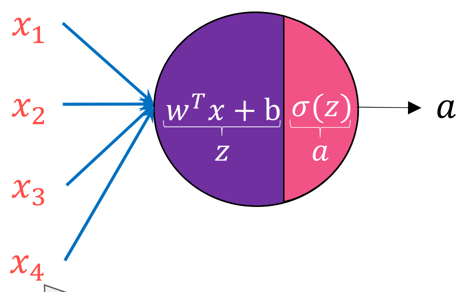
Kai-Lung Hua (花凱龍)

Slide Credit

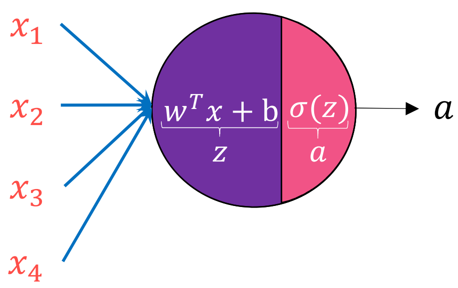
Most of the materials here come from Stanford's cs231n class.

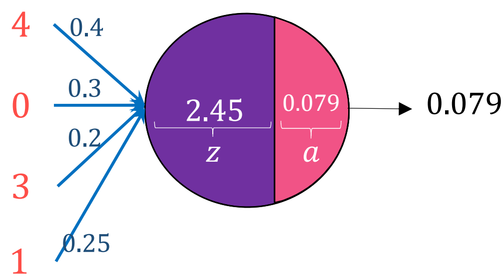
Remember logistic regression?

$$\underbrace{\begin{bmatrix} 4 & 0 & 3 & 1 \\ 3 & 2 & 1 & 1 \\ 0 & 4 & 3 & 1 \\ 2 & 0 & 2 & 3 \\ 2 & 2 & 0 & 1 \end{bmatrix}}_{N \times D}^{X} \quad \underbrace{\begin{bmatrix} 0.4 \\ 0.3 \\ 0.2 \\ 0.25 \end{bmatrix}}_{D \times 1}^{\theta} \quad \underbrace{\begin{bmatrix} 2.45 \\ 2.25 \\ 2.05 \\ 1.95 \\ 1.65 \end{bmatrix}}_{N \times 1}^{X\theta} \quad \underbrace{\begin{bmatrix} 0.079 \\ 0.095 \\ 0.114 \\ 0.125 \\ 0.161 \end{bmatrix}}_{N \times 1}^{\sigma(X\theta)}$$

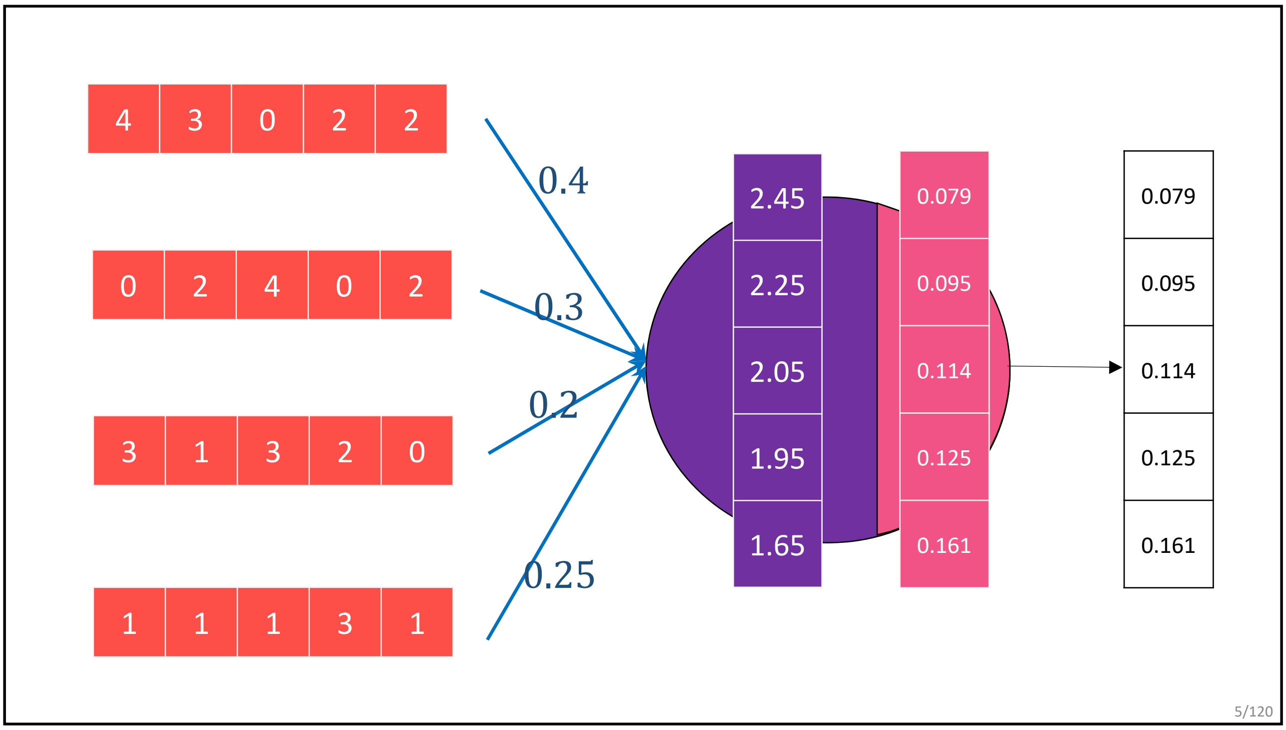


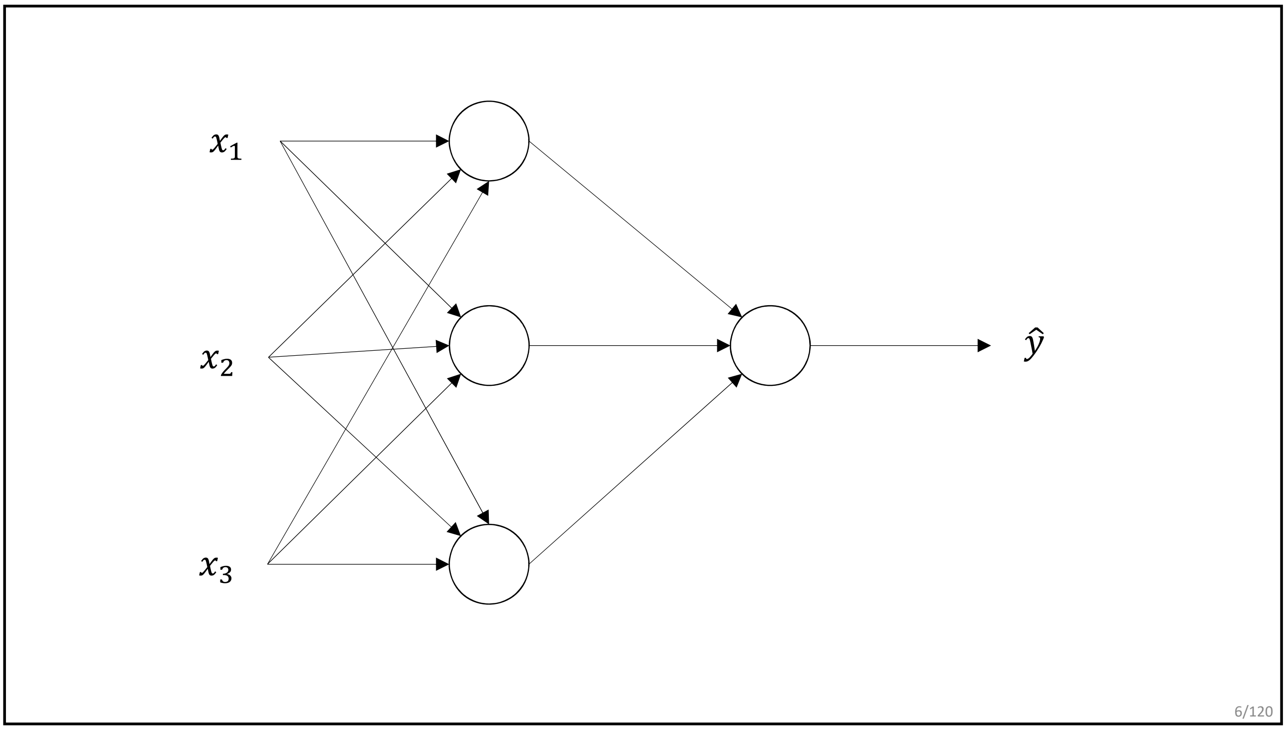
Subscript refers to dimension

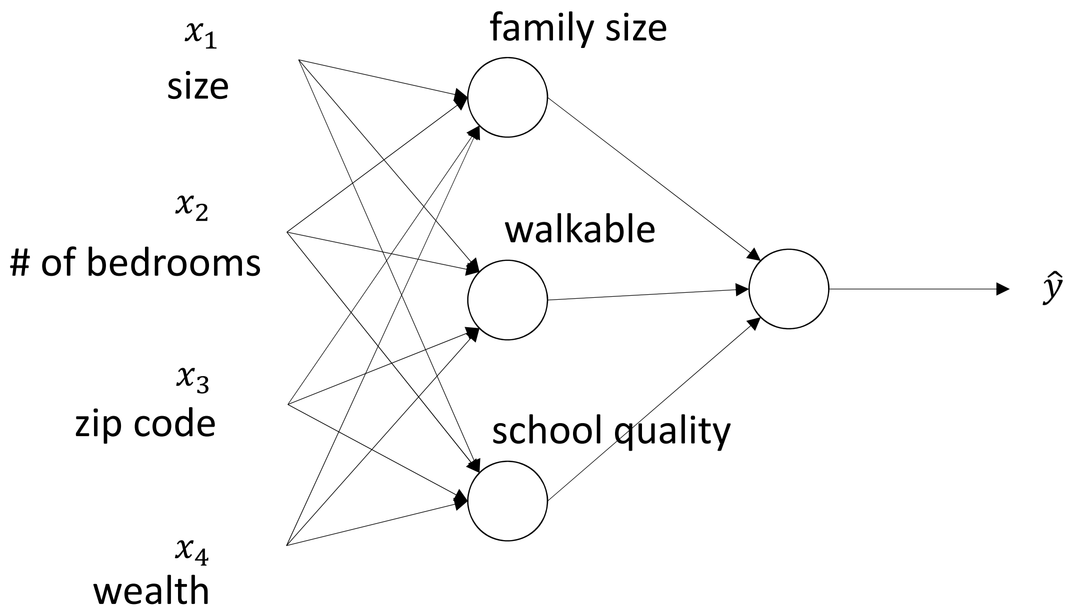


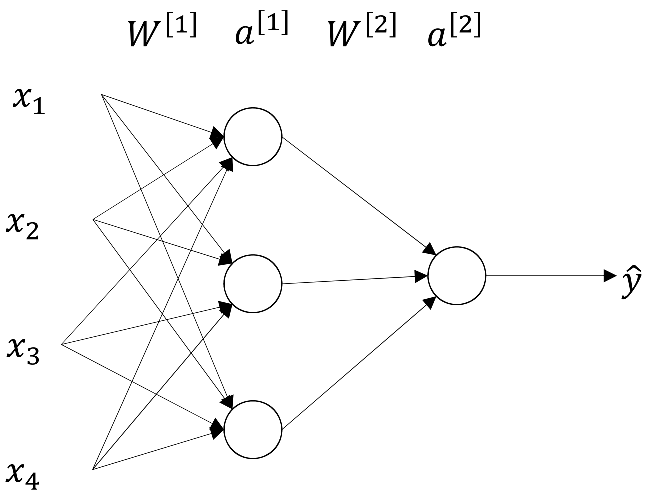


Subscript refers to dimension


4 3 0 2 2
0 2 4 0 2
3 1 3 2 0
1 1 1 3 1
0.4
0.3
0.2
0.25
2.45
2.25
2.05
1.95
1.65
0.079
0.095
0.114
0.125
0.161
0.079
0.095
0.114
0.125
0.161
5/120


x_1
x_2
x_3
$\hat{y}$
6/120

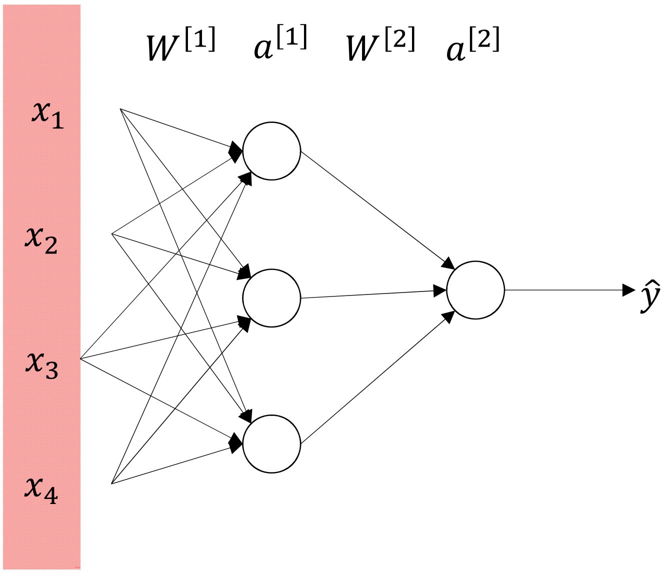




Neural network

They come in all shapes and sizes

This is a 2-layer network (a layer has a set of weights/edges)

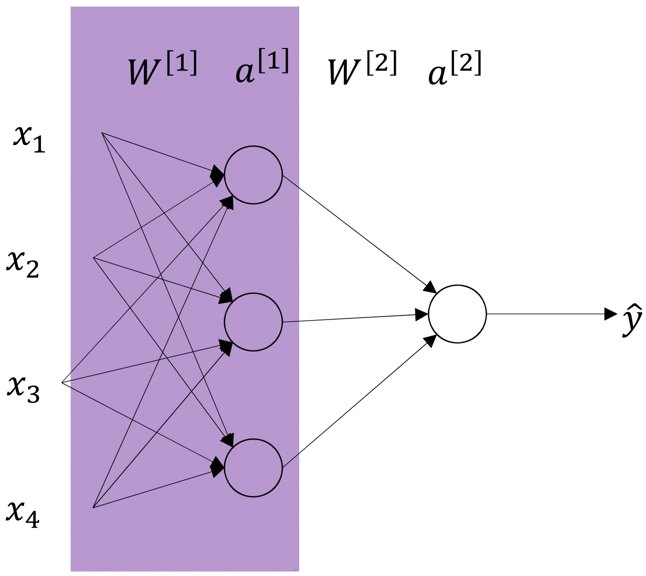


Input layer

Fixed format
Our data X

X has a shape $(N \times D)$

In this example, $D = 4$

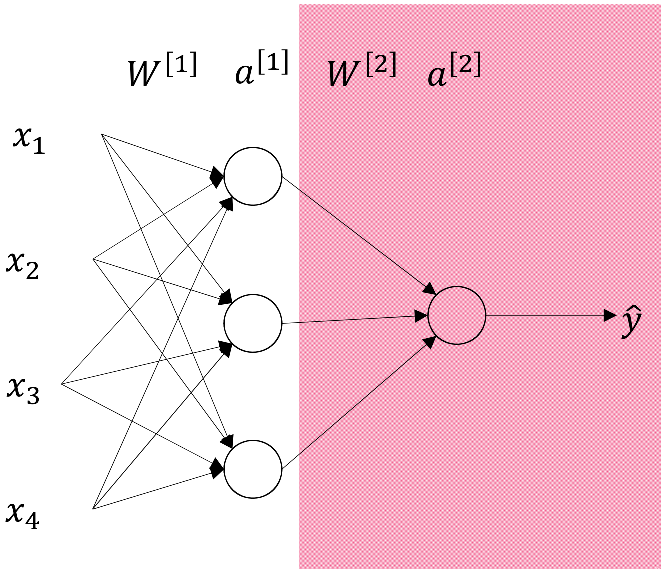


Hidden layer

Gives room for the network to learn other "representations" of your data to lower loss

Intermediate representation

We have 3 hidden neurons **fully connected** to the previous layer

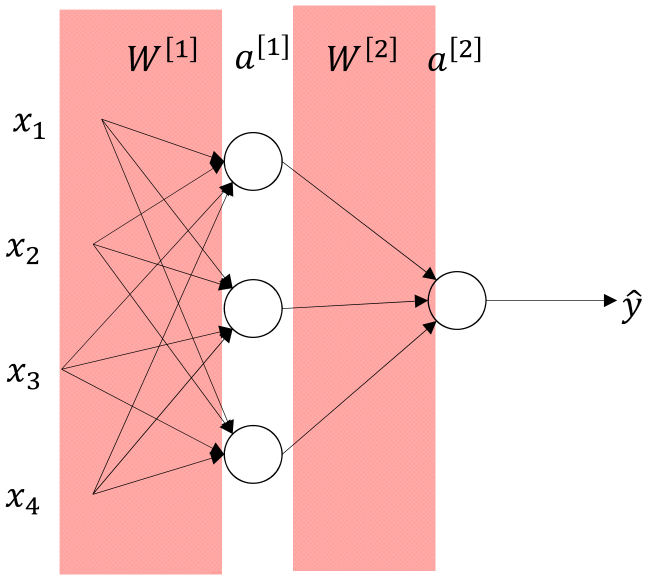


Output layer

Outputs the “answer” to your task

Maybe classification or regression

11/120



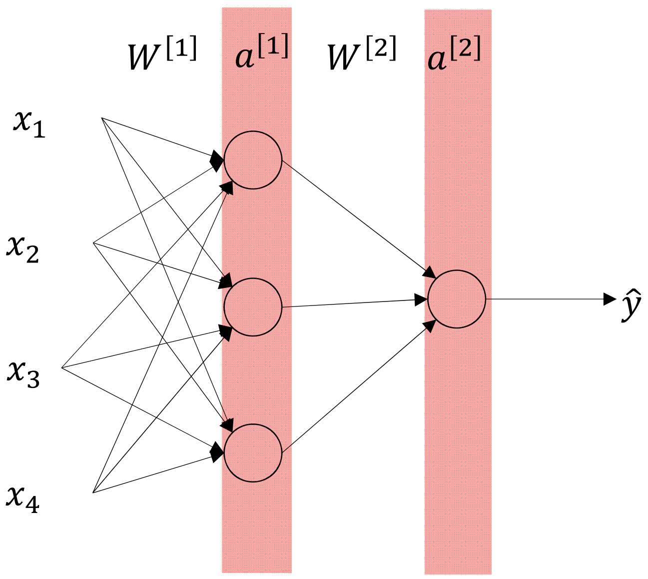
Weights

Our weights (θ) from before are now in separate layers

Layer 1 ($W^{[1]}$) has **12** weight parameters

Layer 2 ($W^{[2]}$) has **3** weight parameters

12/120



Activations

Non-linearities

Ex:
Sigmoid, softmax

13/120

Visit **playground.tensorflow.org**

Configuration 1
- concentric circles data
- No hidden layer
- Only X^1 and X^2 as input features

Configuration 2
- concentric circles data
- 1 hidden layer with 3 nodes
- Only X^1 and X^2 as input features

Check the train and test loss

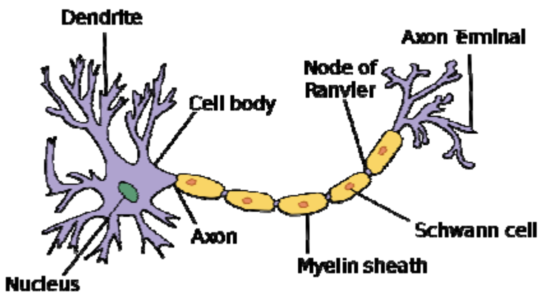
Hover over the hidden neurons to see the “hidden” representations

14/120

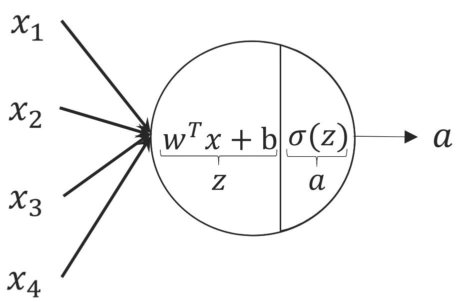
Preview of your Neural Networks Notebook exercise

So why is it called "neural networks"?

A neuron in our brain



A neuron in our neural network



But neurons in our brain are far more complex, and neural networks aren't accurate representations of the brain

Another way of visualizing neural networks

w_o

$$f(w, x) = \frac{1}{1 + \exp -(w_0 + w_1 x_1 + w_2 x_2)}$$

w_1

x_1

w_2

x_2

17/120

Another way of visualizing neural networks

w_o

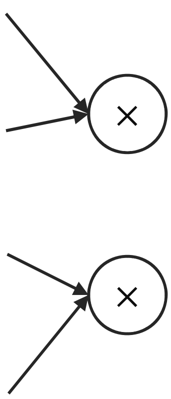
$$f(w, x) = \frac{1}{1 + \exp -(w_0 + w_1 x_1 + w_2 x_2)}$$

w_1

x_1

w_2

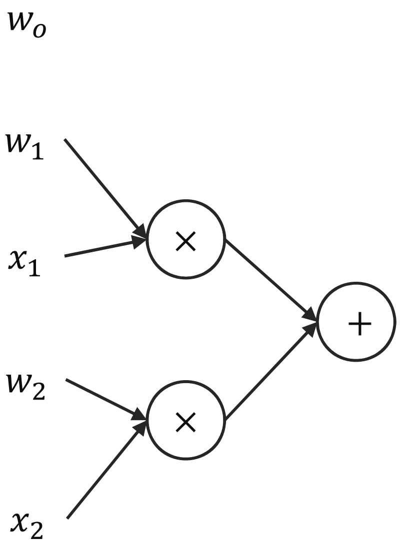
x_2



18/120

Another way of visualizing neural networks

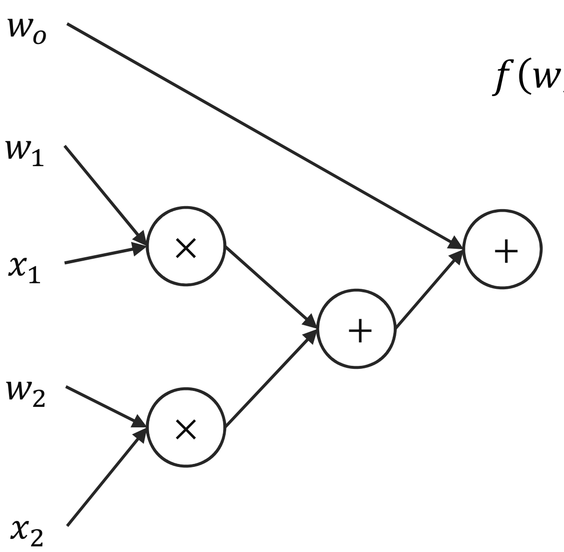
$$f(w,x) = \frac{1}{1 + \exp -(w_0 + w_1 x_1 + w_2 x_2)}$$



Another way of visualizing neural networks

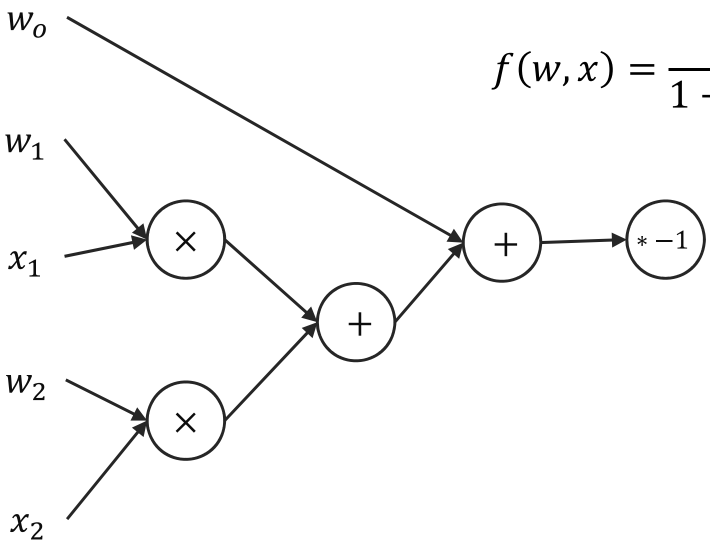
$$f(w,x) = \frac{1}{1 + \exp -(w_0 + w_1 x_1 + w_2 x_2)}$$

$score = \theta^T x_i$



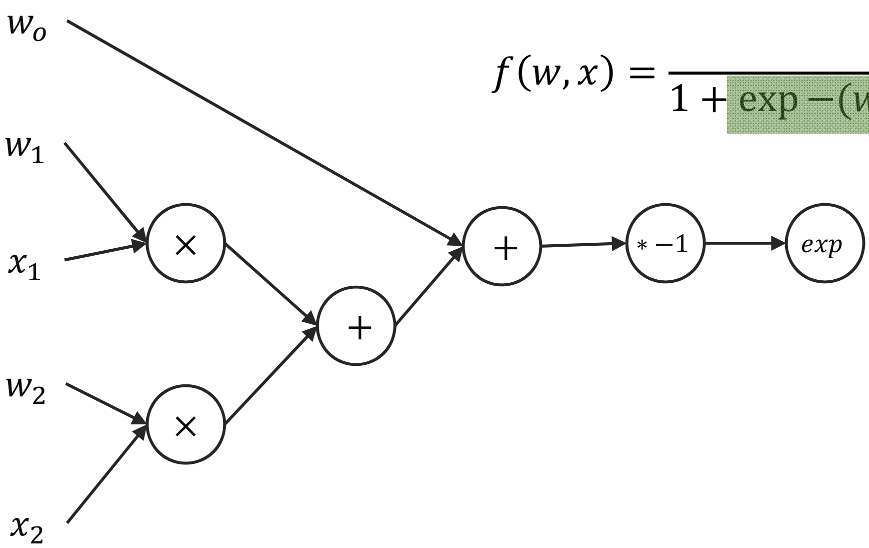
Another way of visualizing neural networks

$$f(w,x) = \frac{1}{1 + \exp -(w_0 + w_1 x_1 + w_2 x_2)}$$



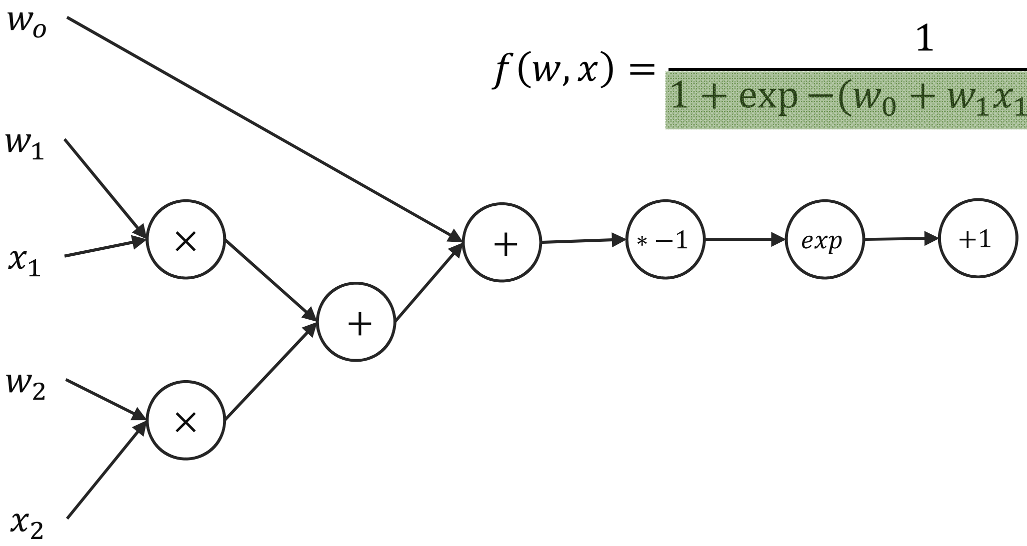
Another way of visualizing neural networks

$$f(w,x) = \frac{1}{1 + \exp -(w_0 + w_1 x_1 + w_2 x_2)}$$



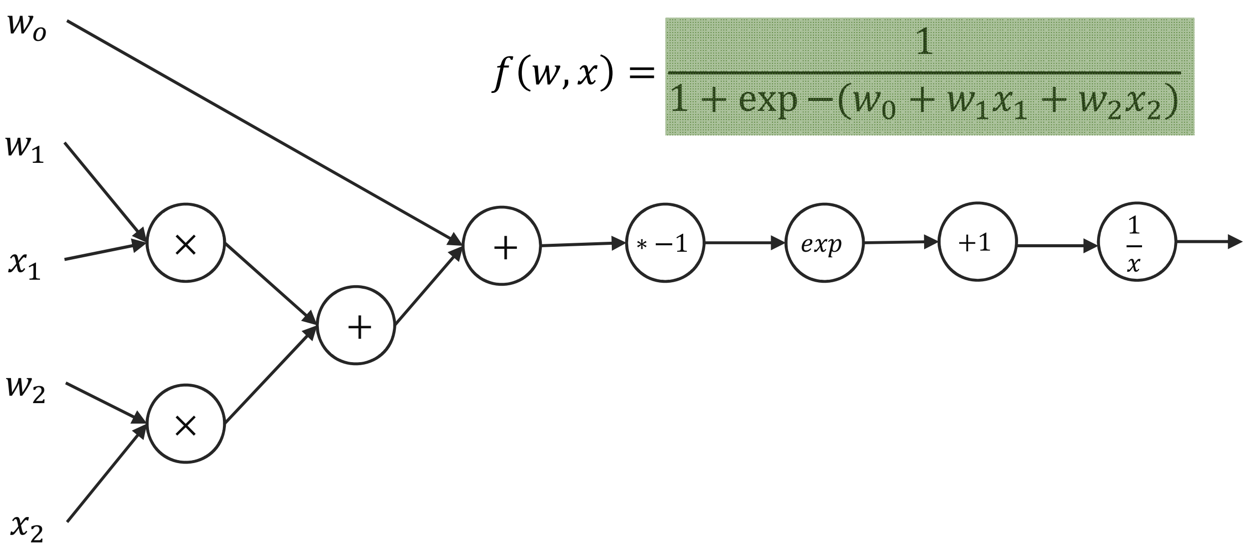
Another way of visualizing neural networks

$$f(w,x) = \frac{1}{1 + \exp -(w_0 + w_1 x_1 + w_2 x_2)}$$



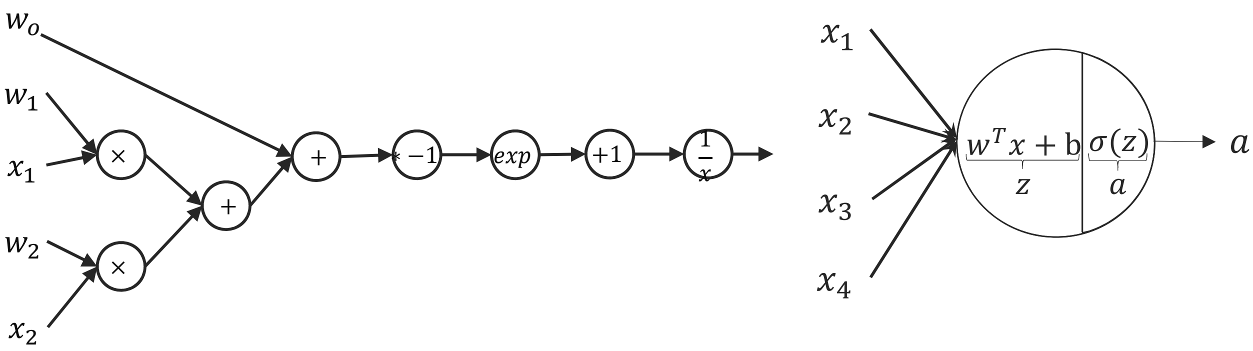
Another way of visualizing neural networks

$$f(w,x) = \frac{1}{1 + \exp -(w_0 + w_1 x_1 + w_2 x_2)}$$



Another way of visualizing neural networks

$$f(w,x) = \frac{1}{1 + \exp -(w_0 + w_1 x_1 + w_2 x_2)}$$



Training

while i < iter:

sample data

[forwardprop] get sample data’s predictions

get the loss by comparing predictions with ground truth

[backprop] get the gradients

update the weights with gradients

i++

Testing

while i < iter:

sample data

[forwardprop] get sample data's predictions

get the loss by comparing predictions with ground truth

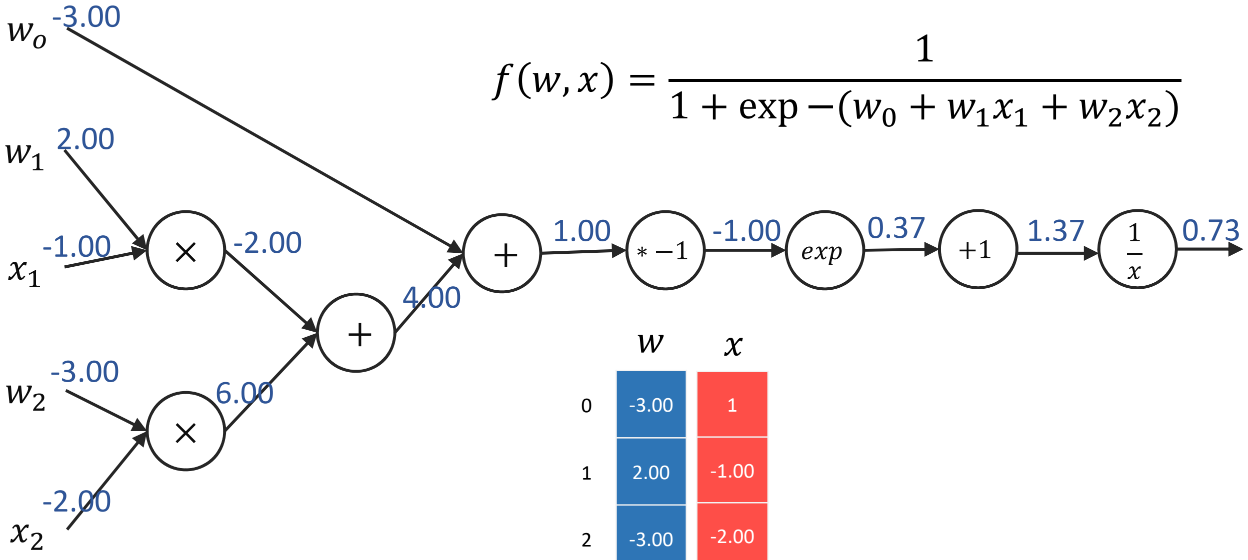
[backprop] get the gradients

update the weights with gradients

i++

Forward propagation, **let the numbers flow**

$$f(w,x) = \frac{1}{1+\exp -(w_0 + w_1 x_1 + w_2 x_2)}$$



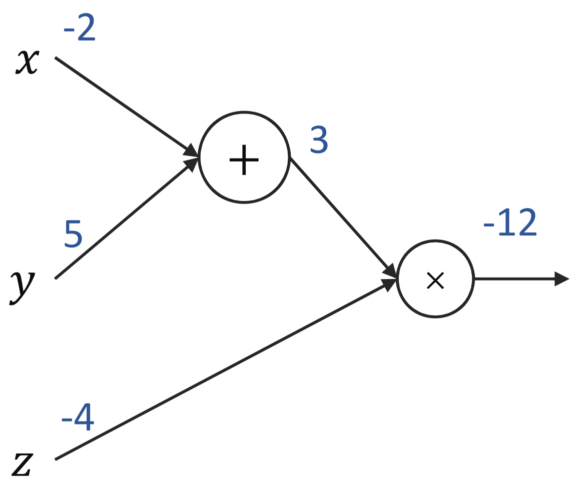
	w	x
0	-3.00	1
1	2.00	-1.00
2	-3.00	-2.00

How do we train Neural Networks?

What parts of the network can we "change"? **Weights.**

How much each **weight** is **contributing to the loss**?

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

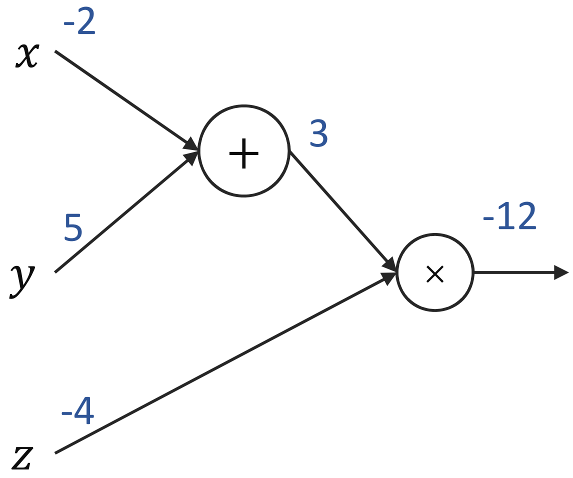


If I add 1 to x, how much will that change $f(x, y, z)$?

Will it make f lower, or higher?

What if I change y, or z?

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



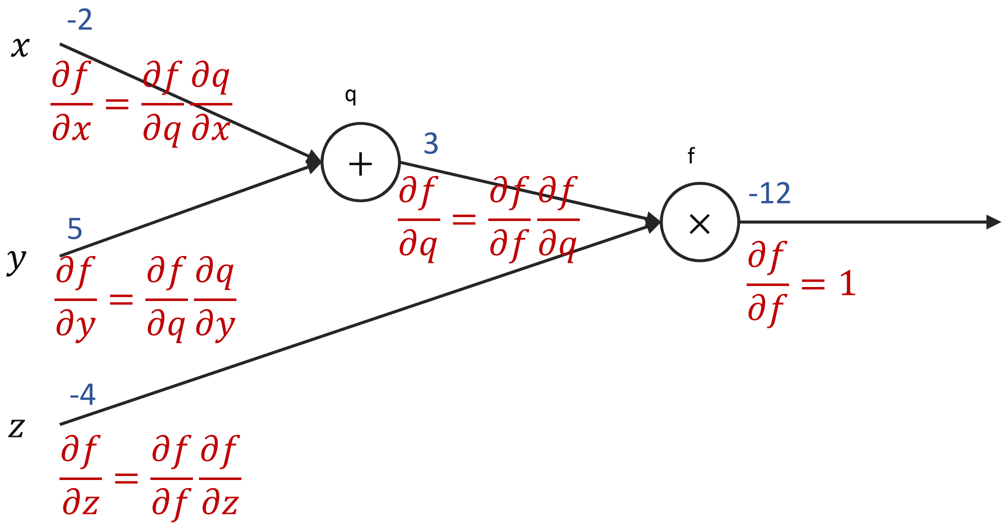
We can also increment and decrement the variables (x, y, z), and check if we are lowering the loss.

This is a tedious process.

Is there a more efficient way?

Calculate for the gradients!

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



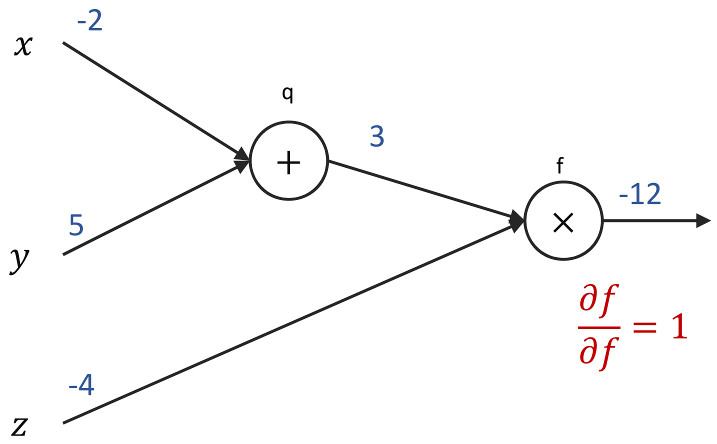
Calculate for the gradients! $f(x, y, z) = (x + y)z$

$$q = (x + y)$$

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

$$f = qz$$

$$\frac{\partial f}{\partial z} = q \,, \quad \frac{\partial f}{\partial q} = z$$



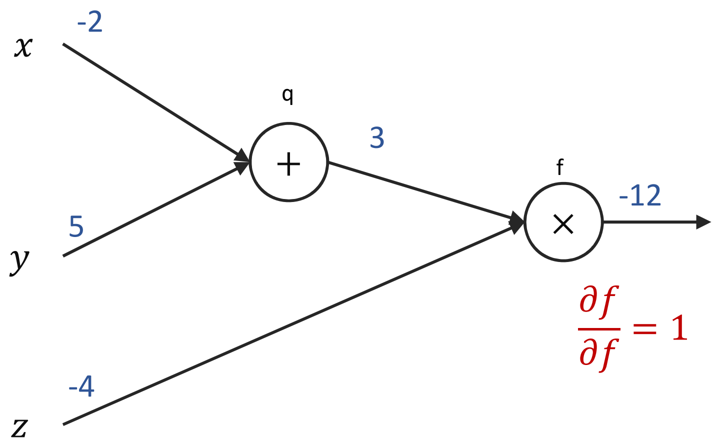
Calculate for the gradients! $f(x, y, z) = (x + y)z$

$$q = (x + y)$$

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

$$f = qz$$

$$\frac{\partial f}{\partial z} = q \,, \quad \frac{\partial f}{\partial q} = z$$



$$\frac{\partial f}{\partial z} = \frac{\partial f}{\partial f}\frac{\partial f}{\partial z} = 1 * q = 1 * 3 = 3$$

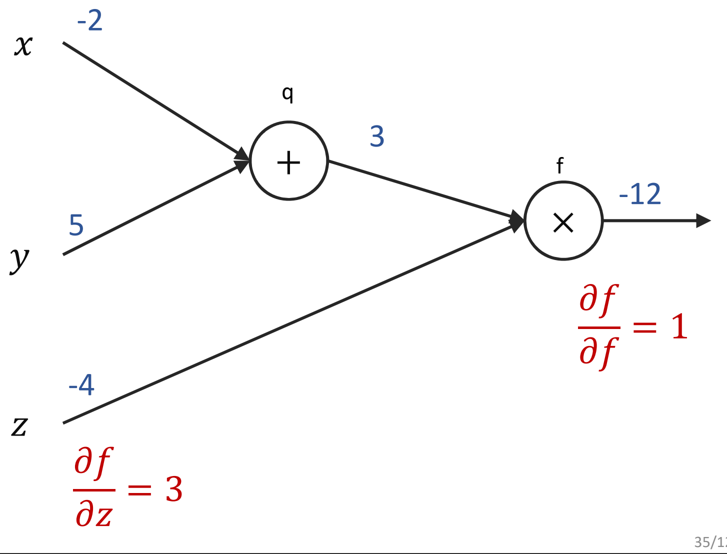
Calculate for the gradients! $f(x,y,z) = (x+y)z$

$q = (x+y)$

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

$f = qz$

$$\frac{\partial f}{\partial z} = q\,, \quad \frac{\partial f}{\partial q} = z$$



35/120

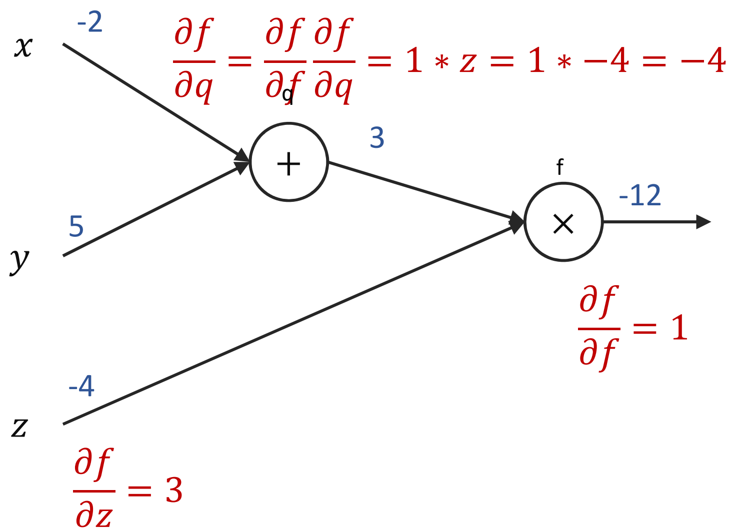
Calculate for the gradients! $f(x,y,z) = (x+y)z$

$q = (x+y)$

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

$f = qz$

$$\frac{\partial f}{\partial z} = q\,, \quad \frac{\partial f}{\partial q} = z$$



36/120

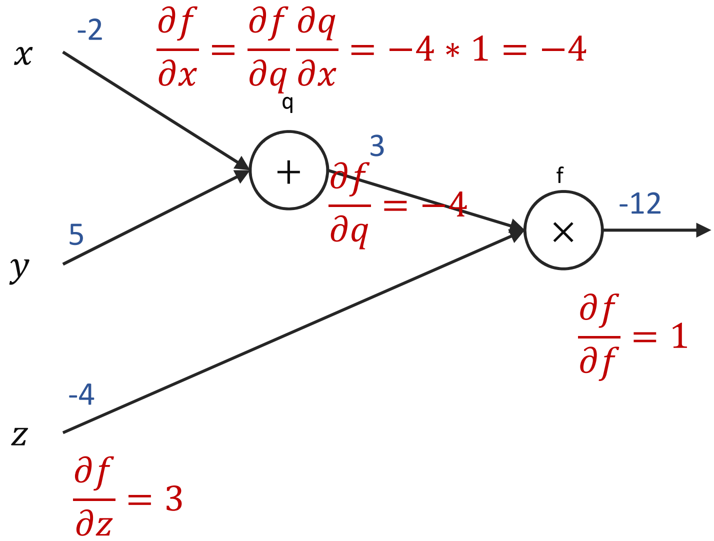
Calculate for the gradients! $f(x,y,z) = (x+y)z$

$$q = (x+y)$$

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

$$f = qz$$

$$\frac{\partial f}{\partial z} = q\,, \quad \frac{\partial f}{\partial q} = z$$



37/120

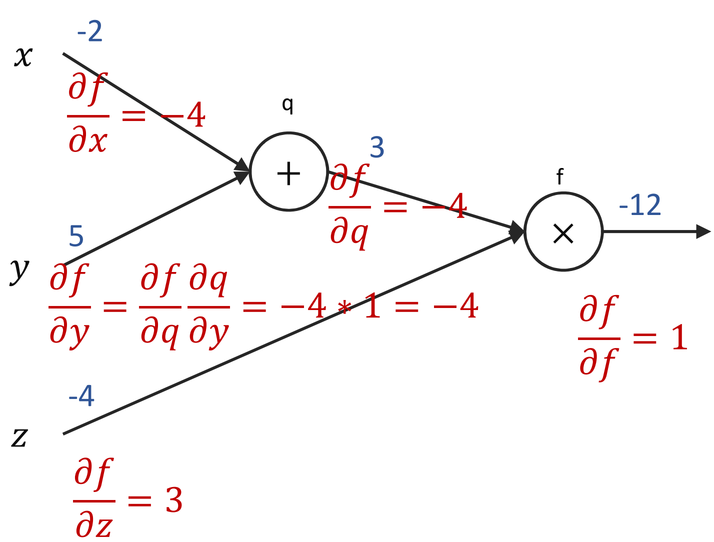
Calculate for the gradients! $f(x,y,z) = (x+y)z$

$$q = (x+y)$$

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

$$f = qz$$

$$\frac{\partial f}{\partial z} = q\,, \quad \frac{\partial f}{\partial q} = z$$



38/120

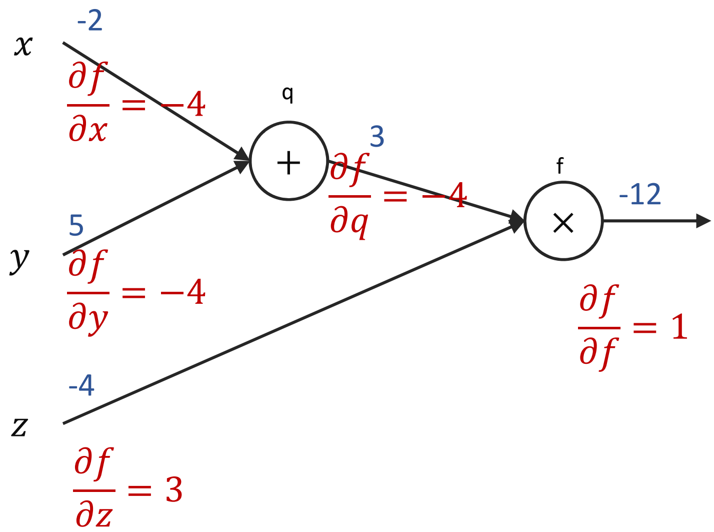
Calculate for the gradients! $f(x,y,z) = (x+y)z$

$q = (x+y)$

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

$f = qz$

$$\frac{\partial f}{\partial z} = q\,, \quad \frac{\partial f}{\partial q} = z$$



39/120

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

$f(x,y,z) = (x+y)z = -12$

if $x := x + h$,

$$\hat{f} = f + \frac{\partial f}{\partial x} * h$$

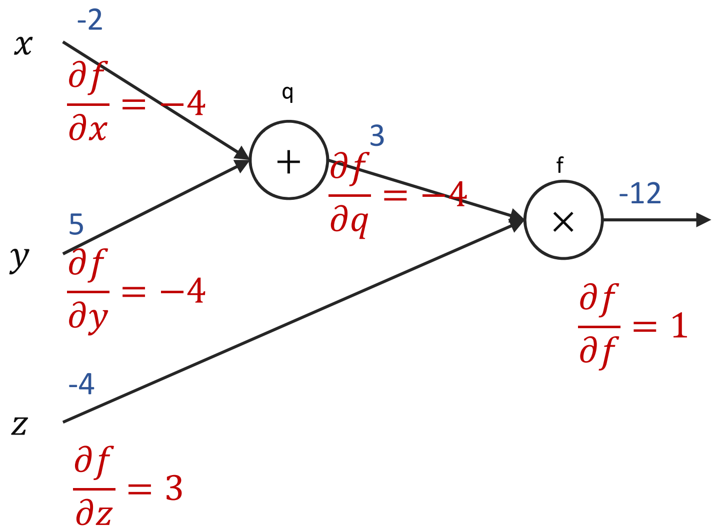
Example:

$h = 1$

$\hat{f} = ((-2+1)+5) * -4 = -16$

$$\hat{f} = f + \frac{\partial f}{\partial x} * h$$
$$\hat{f} = f + -4 * 1$$
$$\hat{f} = -12 + -4 = -16$$

An increase in x changes f proportional to $\frac{\partial f}{\partial x}$!



40/120

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

$$f(x, y, z) = (x + y)z = -12$$

If $y := y + h$,

$$\hat{f} = f + \frac{\partial f}{\partial y} * h$$

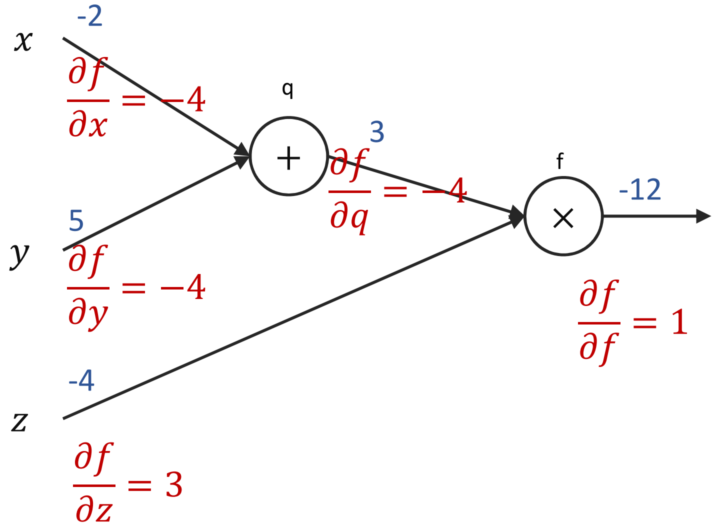
Example:

$h = 1$

$\hat{f} = (-2 + (5 + 1)) * -4 = -16$

$$\boxed{\begin{aligned} \hat{f} &= f + \frac{\partial f}{\partial y} * h \\ \hat{f} &= f + -4 * 1 \\ \hat{f} &= -12 + -4 = -16 \end{aligned}}$$

An increase in y changes f proportional to $\frac{\partial f}{\partial y}$!



41/120

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

$$f(x, y, z) = (x + y)z = -12$$

If $z := z + h$,

$$\hat{f} = f + \frac{\partial f}{\partial z} * h$$

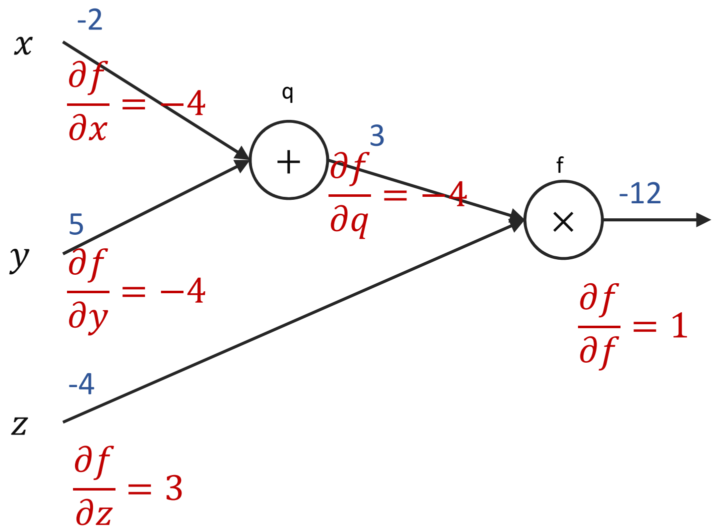
Example:

$h = 1$

$\hat{f} = (-2 + 5) * (-4 + 1) = -9$

$$\boxed{\begin{aligned} \hat{f} &= f + \frac{\partial f}{\partial z} * h \\ \hat{f} &= f + 3 * 1 \\ \hat{f} &= -12 + 3 = -9 \end{aligned}}$$

An increase in z changes f proportional to $\frac{\partial f}{\partial z}$!



42/120

How do we train Neural Networks?

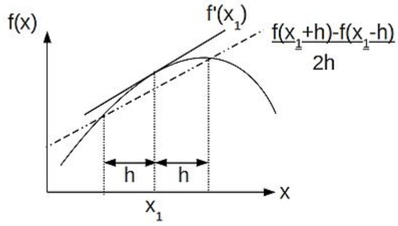
What parts of the network can we "change"? Weights.

How much each weight is contributing to the loss?

Backpropagation

- Backpropagation is just a procedure to compute gradients (Chain Rule) of computational graphs. More specifically, compute the gradients of the objective/loss/error function with respect to each of your parameters in every layer. **Intuitively, the gradients tells you the error contribution of each of your parameters.**
- Two ways of computing the gradients:
  - Numerical gradient (algorithm to approximate the gradient)
    - Slow, approximate, but easy to write / implement
  - Analytical gradient (can be thought of as the gradient solved / derived by hand)
    - Fast, exact, but error-prone
- In practice we use the analytical gradient and check our analytical implementation using the numerical gradient

Numerical Gradient



- Central Difference Method

$$\frac{\partial f(x)}{\partial x} = \frac{f(x+h) - f(x-h)}{2h}$$

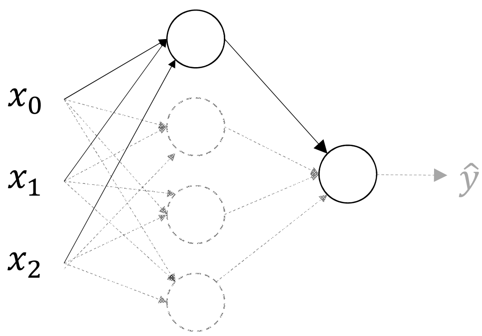
- For vector valued functions we repeat this for every dimension.
- Example:
- $x = \begin{bmatrix} x_1 + h \\ x_2 \\ x_3 \end{bmatrix} \rightarrow a1 = f\left(\begin{bmatrix} x_1 + h \\ x_2 \\ x_3 \end{bmatrix}\right), x = \begin{bmatrix} x_1 - h \\ x_2 \\ x_3 \end{bmatrix} \rightarrow a2 = f\left(\begin{bmatrix} x_1 - h \\ x_2 \\ x_3 \end{bmatrix}\right) \rightarrow \frac{\partial f(x)}{\partial x_1} = \frac{a1 - a2}{2h}$
- $x = \begin{bmatrix} x_1 \\ x_2 + h \\ x_3 \end{bmatrix} \rightarrow b1 = f\left(\begin{bmatrix} x_1 \\ x_2 + h \\ x_3 \end{bmatrix}\right), x = \begin{bmatrix} x_1 \\ x_2 - h \\ x_3 \end{bmatrix} \rightarrow b2 = f\left(\begin{bmatrix} x_1 \\ x_2 - h \\ x_3 \end{bmatrix}\right) \rightarrow \frac{\partial f(x)}{\partial x_2} = \frac{b1 - b2}{2h}$
- $x = \begin{bmatrix} x_1 \\ x_2 \\ x_3 + h \end{bmatrix} \rightarrow c1 = f\left(\begin{bmatrix} x_1 \\ x_2 \\ x_3 + h \end{bmatrix}\right), x = \begin{bmatrix} x_1 \\ x_2 \\ x_3 - h \end{bmatrix} \rightarrow c2 = f\left(\begin{bmatrix} x_1 \\ x_2 \\ x_3 - h \end{bmatrix}\right) \rightarrow \frac{\partial f(x)}{\partial x_3} = \frac{c1 - c2}{2h}$

$$\frac{\partial f(x)}{dx} = \begin{bmatrix} \frac{\partial f(x)}{\partial x_1} \\ \frac{\partial f(x)}{\partial x_2} \\ \frac{\partial f(x)}{\partial x_3} \end{bmatrix} = \begin{bmatrix} \frac{a1-a2}{2h} \\ \frac{b1-b2}{2h} \\ \frac{c1-c2}{2h} \end{bmatrix}$$

45/120

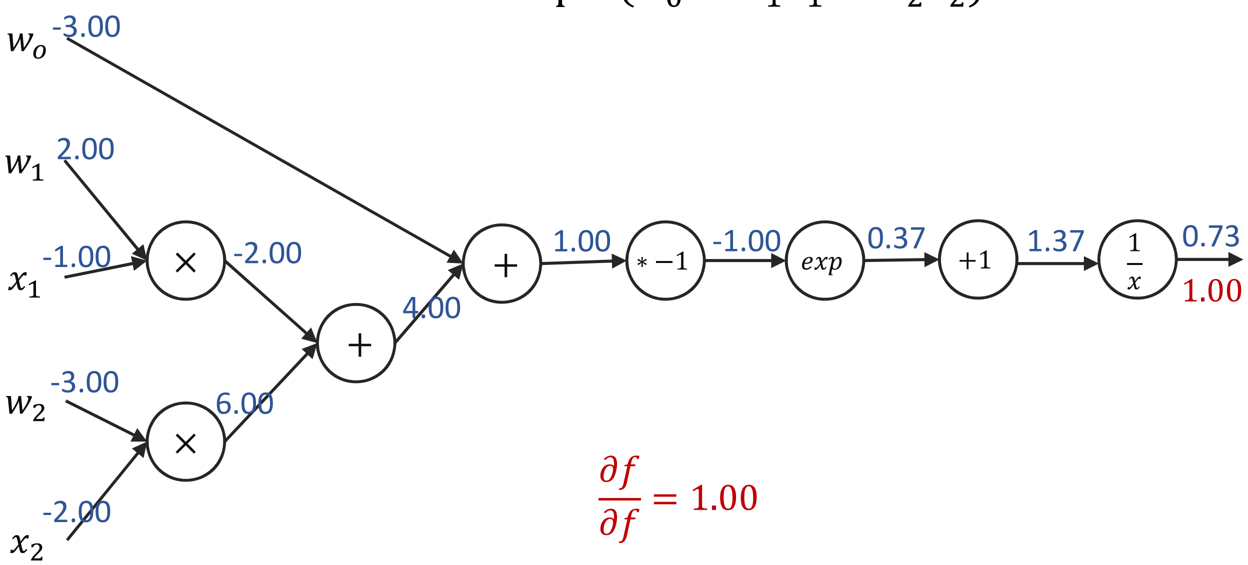
Analytical Gradient Example Next Slide

$$f(w, x) = \frac{1}{1 + \exp -(w_0 + w_1 x_1 + w_2 x_2)}$$



46/120

$$f(w,x)=\frac{1}{1+\exp-(w_0+w_1x_1+w_2x_2)}$$



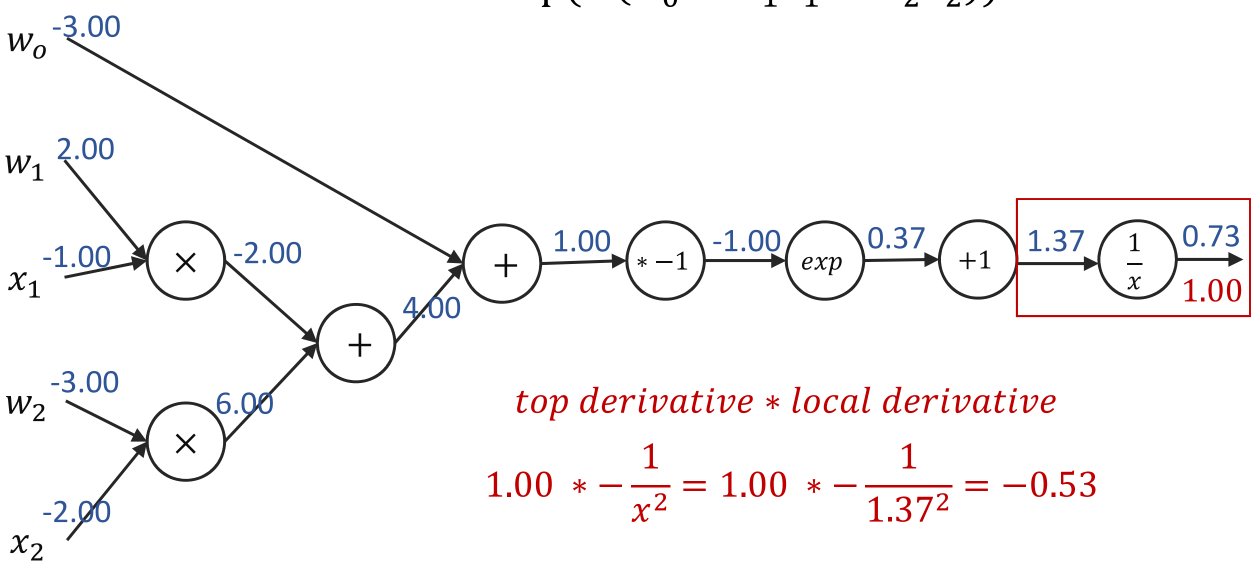
$$f(x)=e^x \qquad \frac{df}{dx}=e^x$$

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

$$f(x)=ax \qquad \frac{df}{dx}=a$$

$$f(x)=c+x \qquad \frac{df}{dx}=1$$

$$f(w,x) = \frac{1}{1+\exp(-(w_0 + w_1x_1 + w_2x_2))}$$

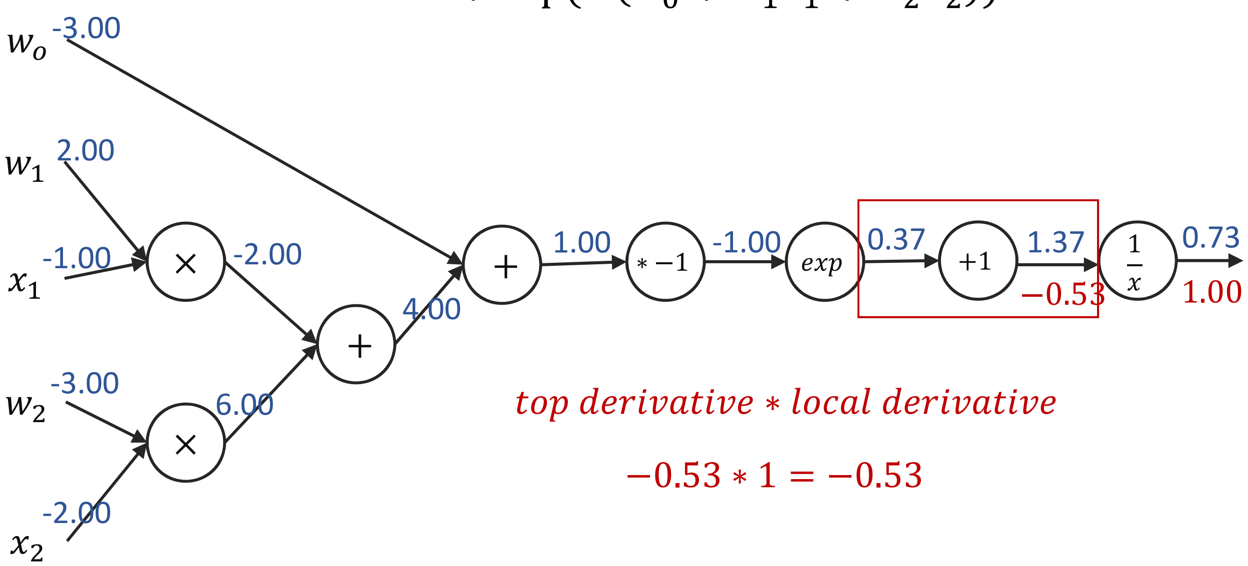


top derivative ∗ local derivative

$$1.00 * -\frac{1}{x^2} = 1.00 * -\frac{1}{1.37^2} = -0.53$$

49/120

$$f(w,x) = \frac{1}{1+\exp(-(w_0 + w_1x_1 + w_2x_2))}$$

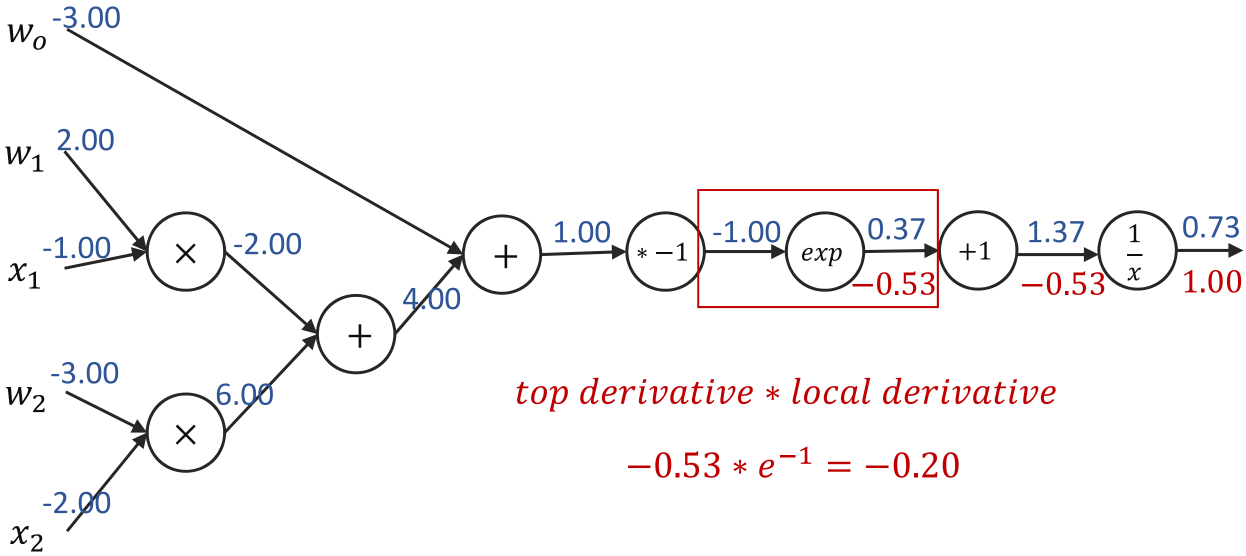


top derivative ∗ local derivative

$$-0.53 * 1 = -0.53$$

50/120

$$f(w,x) = \frac{1}{1 + \exp(-(w_0 + w_1 x_1 + w_2 x_2))}$$

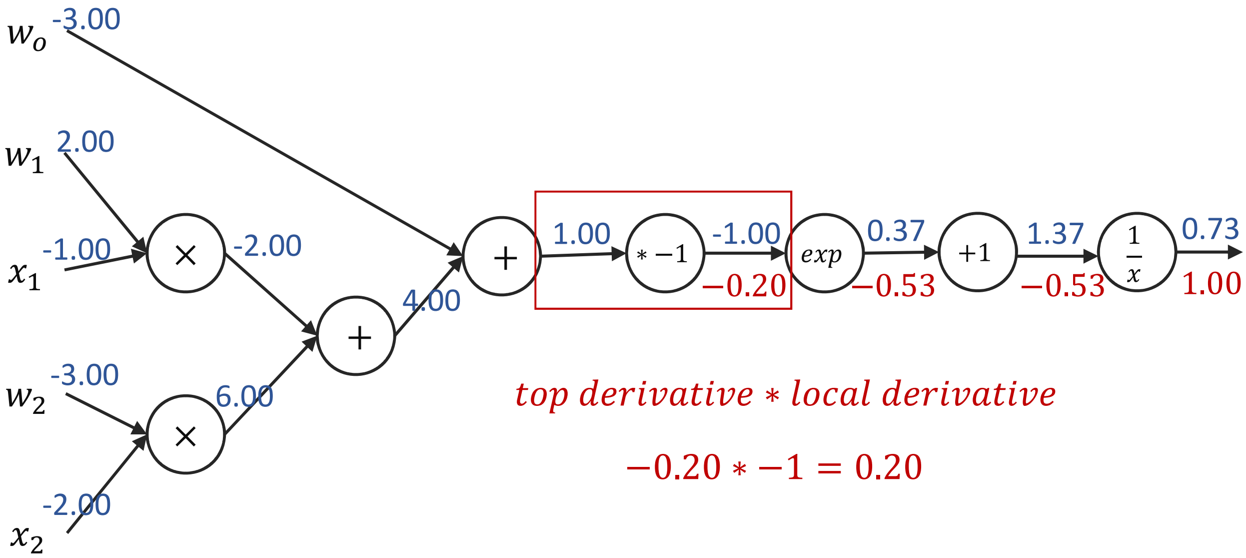


$$top\ derivative * local\ derivative$$

$$-0.53 * e^{-1} = -0.20$$

51/120

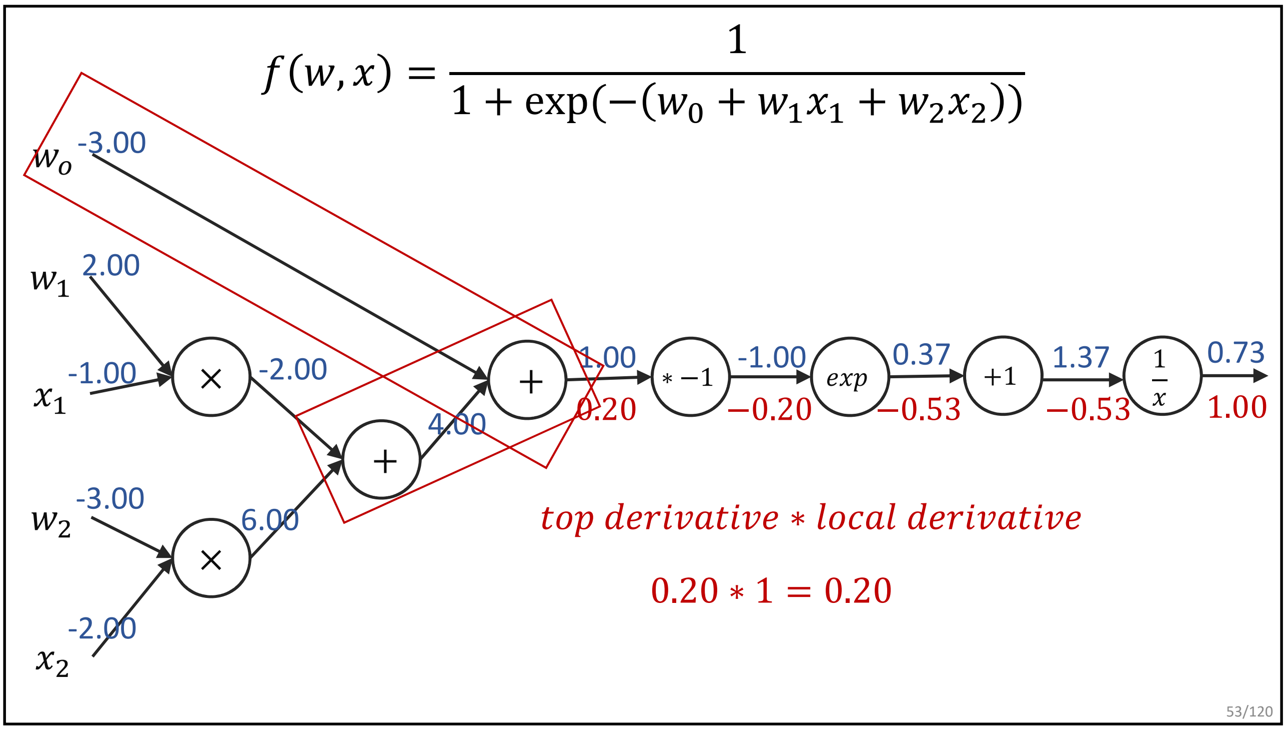
$$f(w,x) = \frac{1}{1 + \exp(-(w_0 + w_1 x_1 + w_2 x_2))}$$

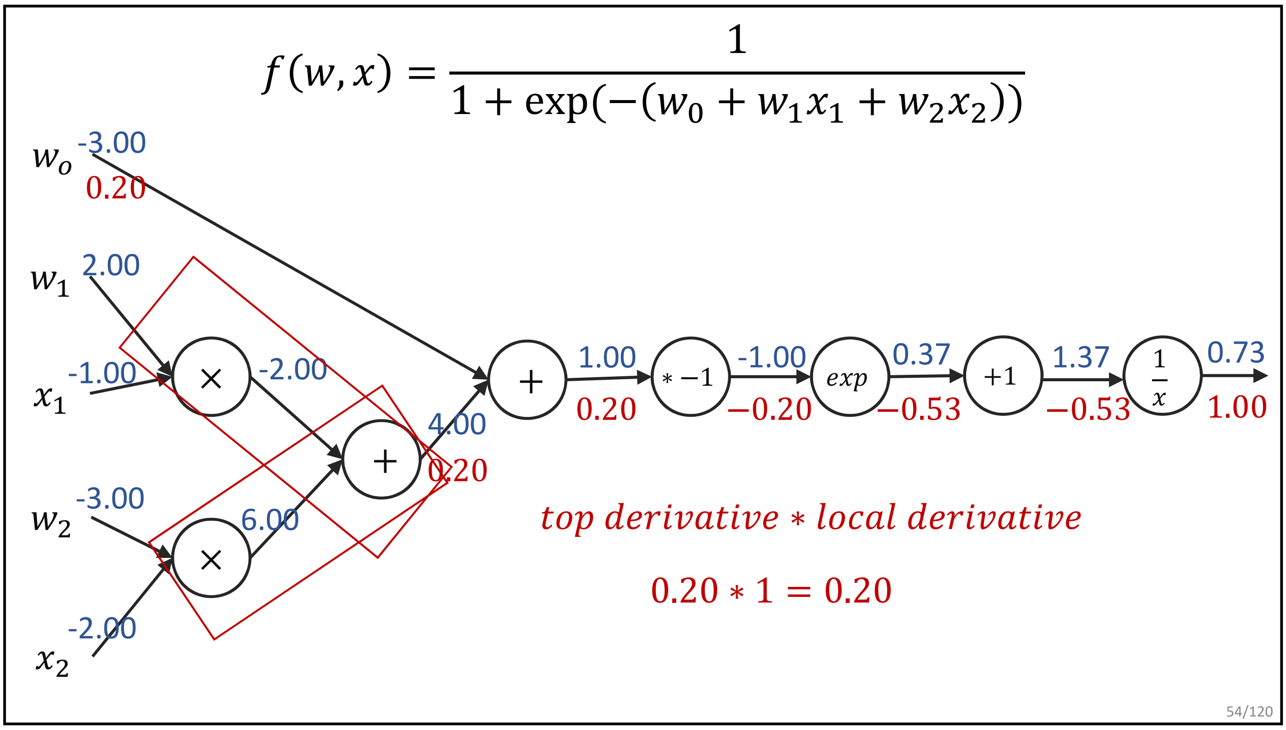


$$top\ derivative * local\ derivative$$

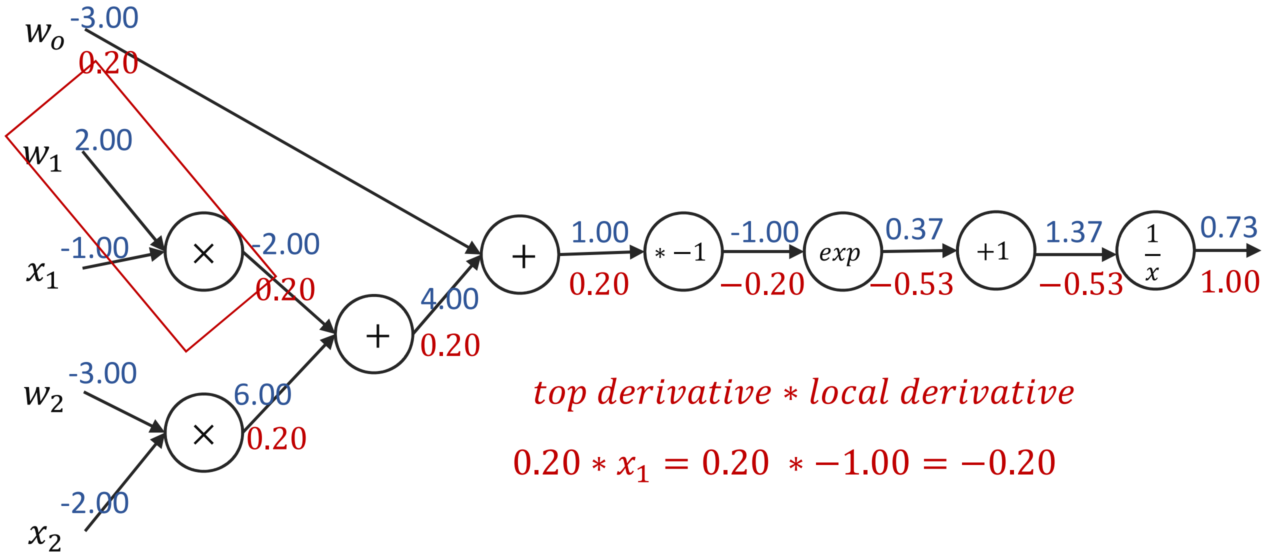
$$-0.20 * -1 = 0.20$$

52/120


$f(w,x) = \frac{1}{1+\exp(-(w_0 + w_1x_1 + w_2x_2))}$
w_o -3.00
w_1 2.00
x_1 -1.00
w_2 -3.00
x_2 -2.00
-2.00
6.00
4.00
1.00
0.20
$*-1$
-1.00
-0.20
exp
0.37
-0.53
$+1$
1.37
-0.53
$\frac{1}{x}$
0.73
1.00
$top\ derivative * local\ derivative$
$0.20 * 1 = 0.20$
53/120


$f(w,x) = \frac{1}{1+\exp(-(w_0 + w_1x_1 + w_2x_2))}$
w_o -3.00
0.20
w_1 2.00
x_1 -1.00
w_2 -3.00
x_2 -2.00
-2.00
6.00
4.00
0.20
1.00
0.20
$*-1$
-1.00
-0.20
exp
0.37
-0.53
$+1$
1.37
-0.53
$\frac{1}{x}$
0.73
1.00
$top\ derivative * local\ derivative$
$0.20 * 1 = 0.20$
54/120

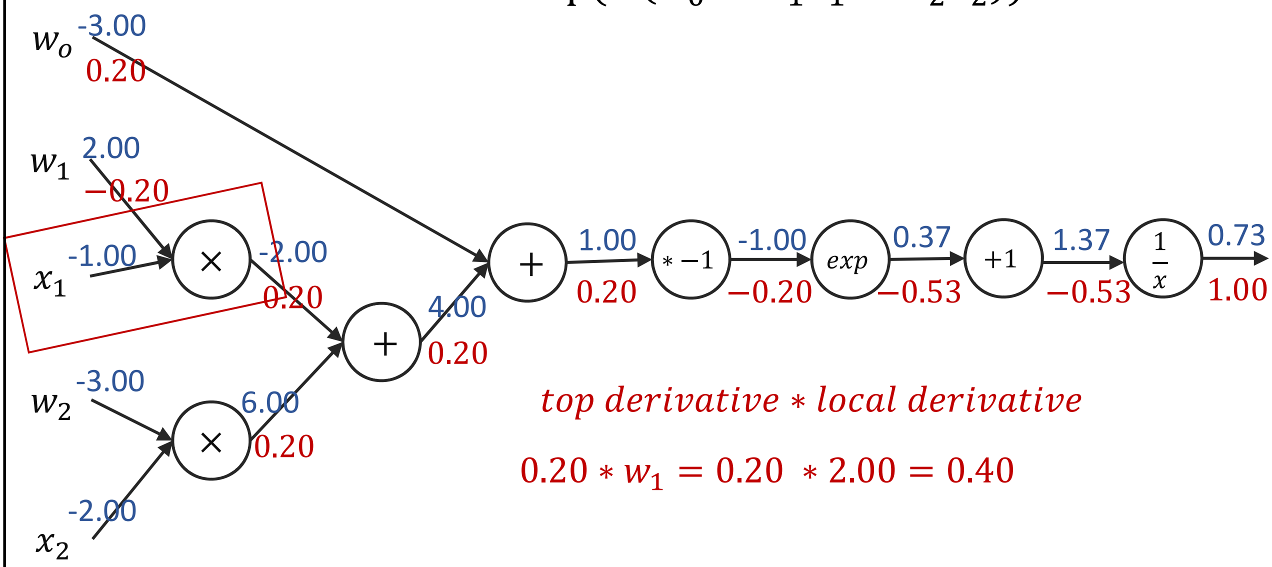
$$f(w,x) = \frac{1}{1+\exp(-(w_0 + w_1x_1 + w_2x_2))}$$



top derivative ∗ local derivative

$0.20 * x_1 = 0.20 * -1.00 = -0.20$

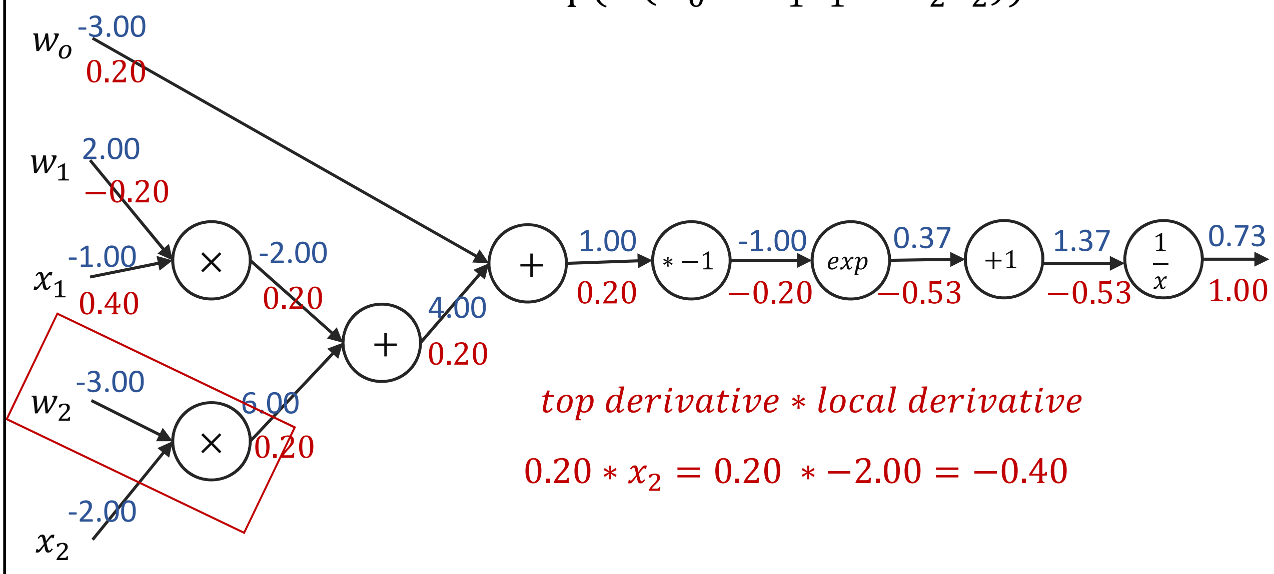
$$f(w,x) = \frac{1}{1+\exp(-(w_0 + w_1x_1 + w_2x_2))}$$



top derivative ∗ local derivative

$0.20 * w_1 = 0.20 * 2.00 = 0.40$

$$f(w,x) = \frac{1}{1+\exp(-(w_0 + w_1x_1 + w_2x_2))}$$

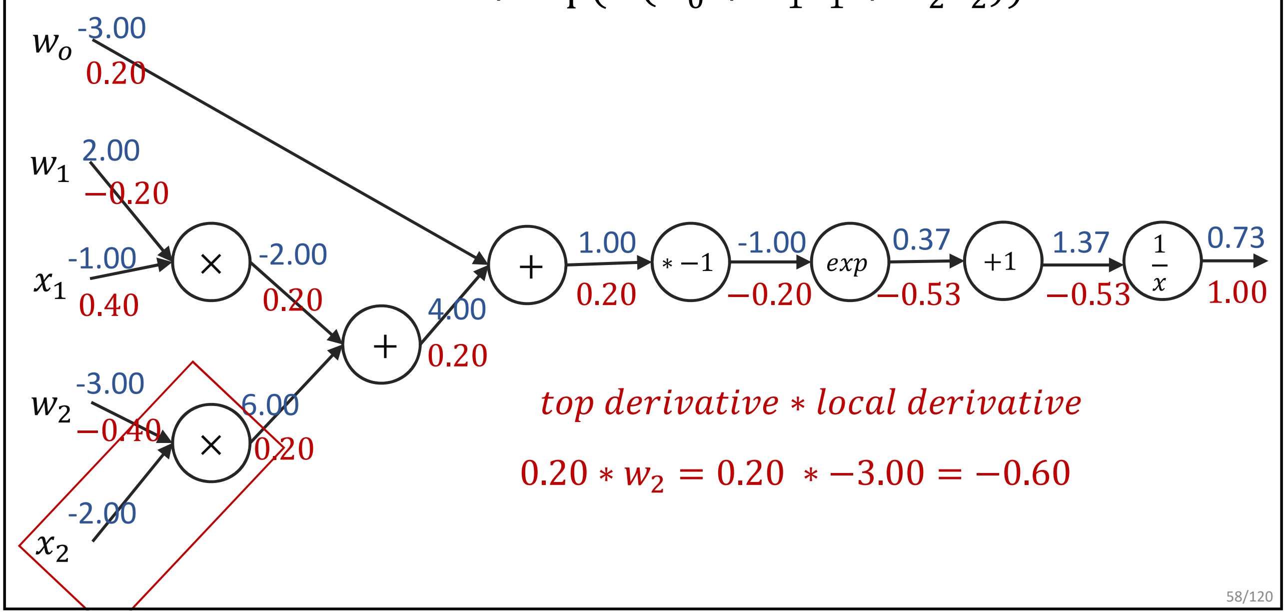


$top\ derivative * local\ derivative$

$$0.20 * x_2 = 0.20\ * -2.00 = -0.40$$

57/120

$$f(w,x) = \frac{1}{1+\exp(-(w_0 + w_1x_1 + w_2x_2))}$$

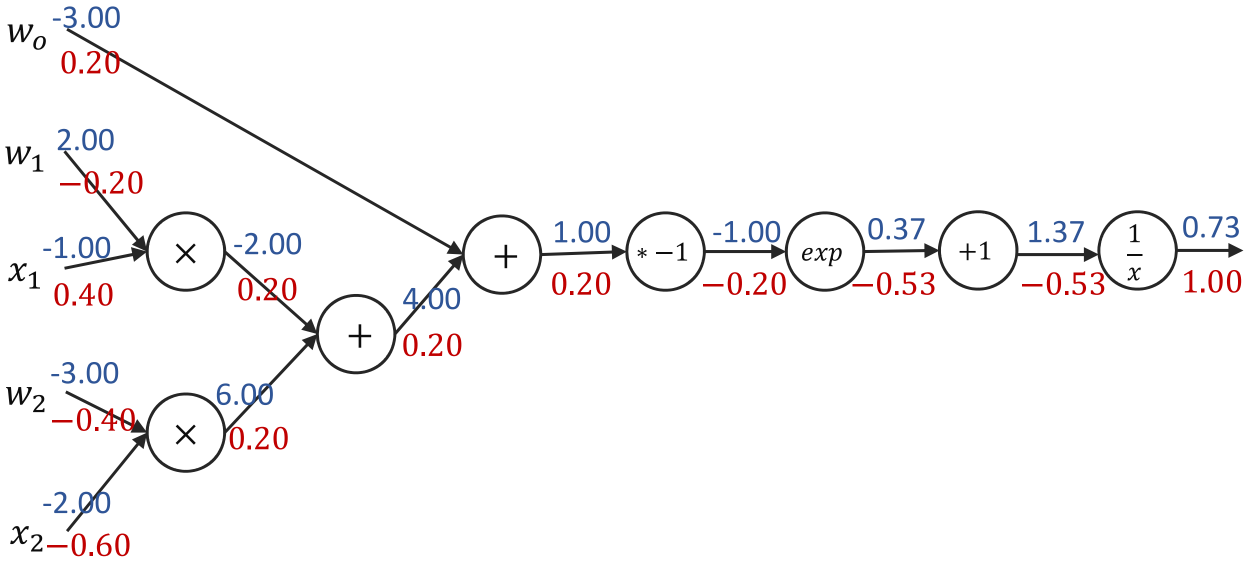


$top\ derivative * local\ derivative$

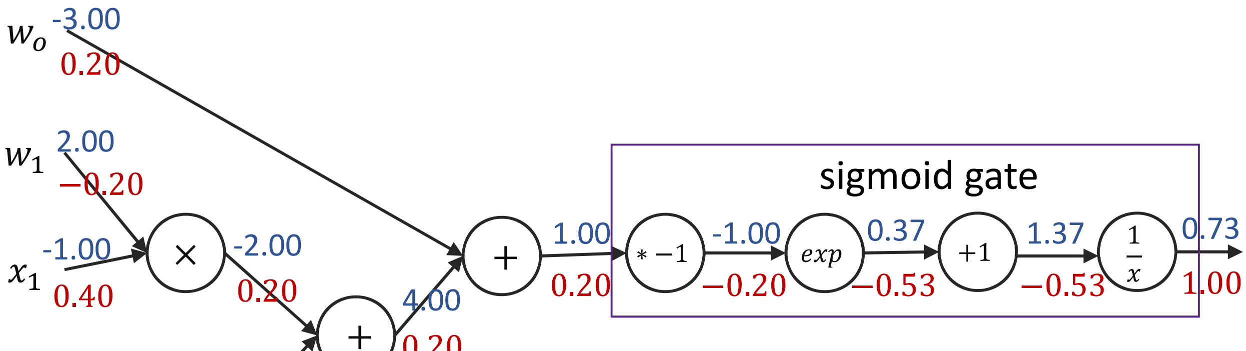
$$0.20 * w_2 = 0.20\ * -3.00 = -0.60$$

58/120

$$f(w,x) = \frac{1}{1+\exp(-(w_0 + w_1 x_1 + w_2 x_2))}$$



$$f(w,x) = \frac{1}{1+\exp(-(w_0 + w_1 x_1 + w_2 x_2))}$$



$$\frac{d\sigma(x)}{dx} = (1-\sigma(x))\sigma(x)$$

$$\frac{d\sigma(x)}{dx} = (1-0.73)*(0.73) = \sim 0.20$$

Neural Networks

Neural Networks

(Before) Logistic Function $\quad f = \text{activation}(\boldsymbol{Wx})$

$$x, w, b \rightarrow z = w^T x + b \rightarrow a = \sigma(z) \rightarrow \hat{y}$$

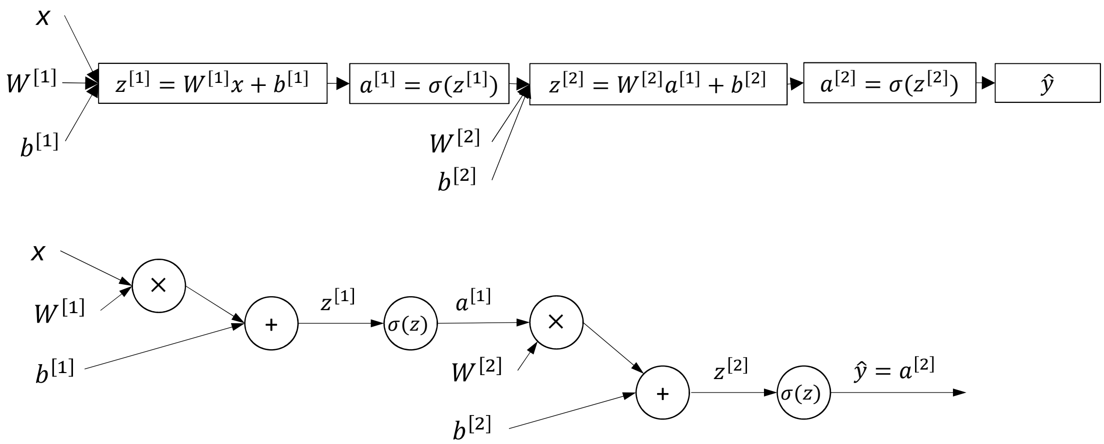
(Now) 2-layer Neural Network $\quad f = \boldsymbol{activation}(\boldsymbol{W_2}\text{activation}(\boldsymbol{W_1 x}))$

$$x, W^{[1]}, b^{[1]} \rightarrow z^{[1]} = W^{[1]}x + b^{[1]} \rightarrow a^{[1]} = \sigma(z^{[1]}) \rightarrow z^{[2]} = W^{[2]}a^{[1]} + b^{[2]} \rightarrow a^{[2]} = \sigma(z^{[2]}) \rightarrow \hat{y}$$

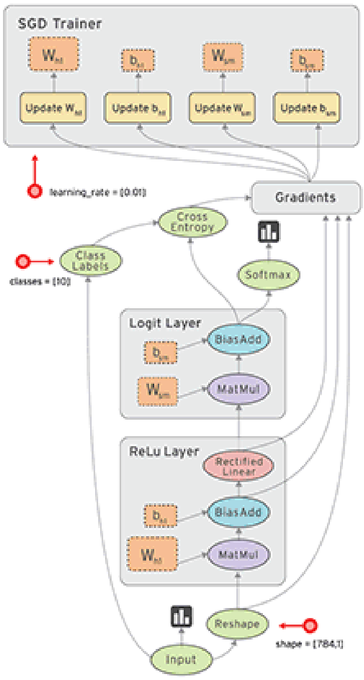
$$W^{[2]}, b^{[2]} \rightarrow z^{[2]} = W^{[2]}a^{[1]} + b^{[2]}$$

(or) 3-layer Neural Network $\quad f = \boldsymbol{activation}(\boldsymbol{W_3}(\text{activation}(\boldsymbol{W_2}\text{activation}(\boldsymbol{W_1 x}))))$

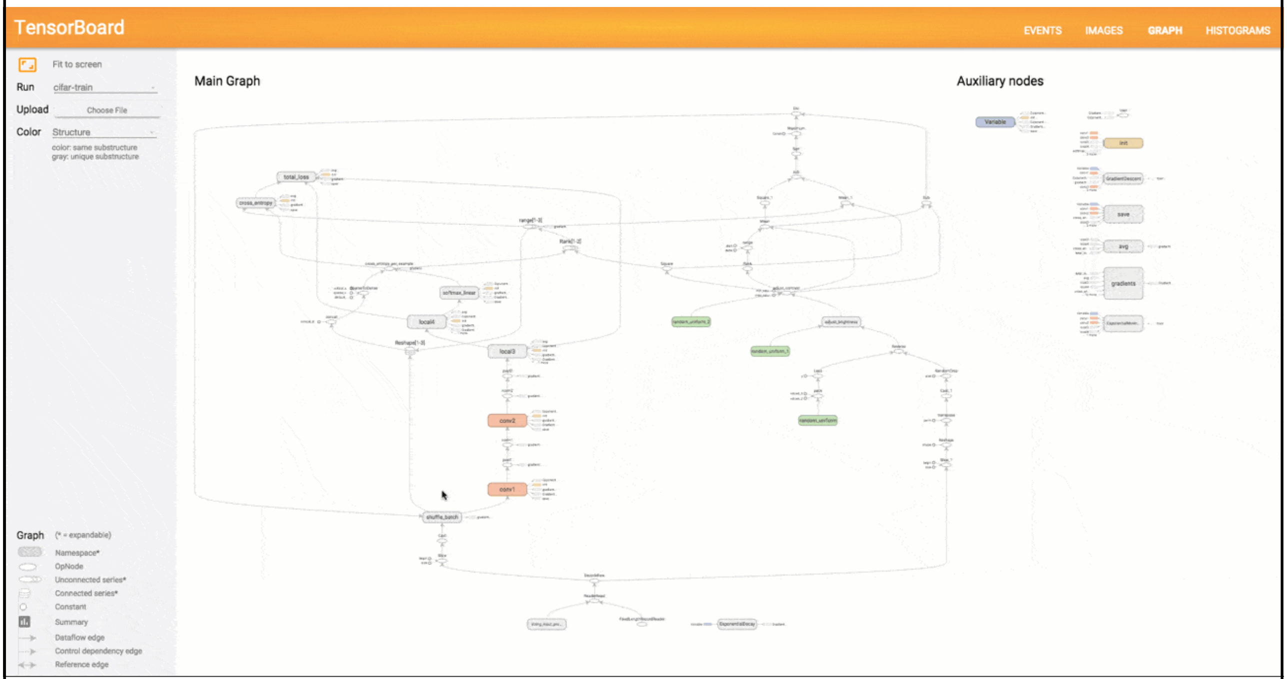
Neural Networks as Computational Graphs



63/120



64/120



65/120

Common Activation Functions / Non-linearities

66/120

Common Activation Functions / Non-linearities

What happens if we don’t use activation functions?

1-layer Neural Network $\boldsymbol{f} = (\boldsymbol{W}\boldsymbol{x})$

2-layer Neural Network $\boldsymbol{f} = \left(\boldsymbol{W}_2(\boldsymbol{W}_1\boldsymbol{x})\right)$

$$= [\boldsymbol{W}_2\boldsymbol{W}_1]\boldsymbol{x} = \boxed{\boldsymbol{W}'\boldsymbol{x}}$$

$$\boldsymbol{W}' = \boldsymbol{W}_2\boldsymbol{W}_1$$

We end up with just another linear model, the extra layers did not do anything

3-layer Neural Network $\boldsymbol{f} = (\boldsymbol{W}_3(\,(\boldsymbol{W}_2(\boldsymbol{W}_1\boldsymbol{x}))))$

$$= [\boldsymbol{W}_3\boldsymbol{W}_2\boldsymbol{W}_1]\boldsymbol{x} = \boxed{\boldsymbol{W}'\boldsymbol{x}}$$

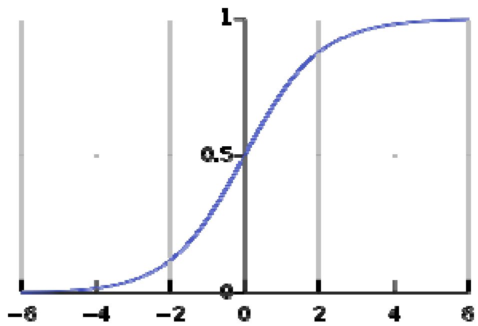
$$\boldsymbol{W}' = \boldsymbol{W}_3\boldsymbol{W}_2\boldsymbol{W}_1$$

67/120

Common Activation Functions / Non-linearities > Sigmoid function

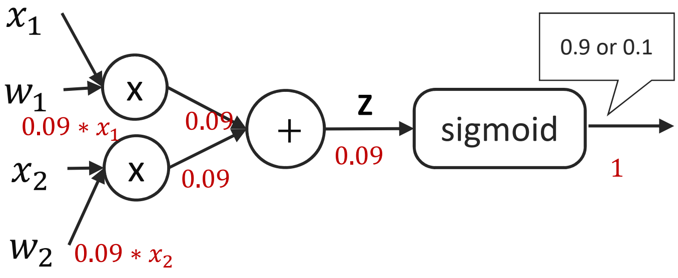
Sigmoid function

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



range $[0, 1]$

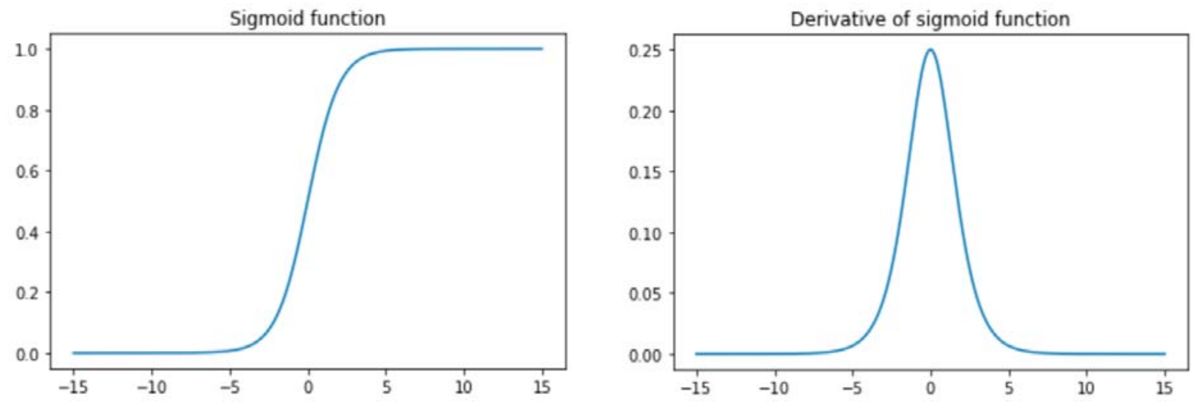
Saturated (near 0 or 1) neurons ‘kill’ the gradients

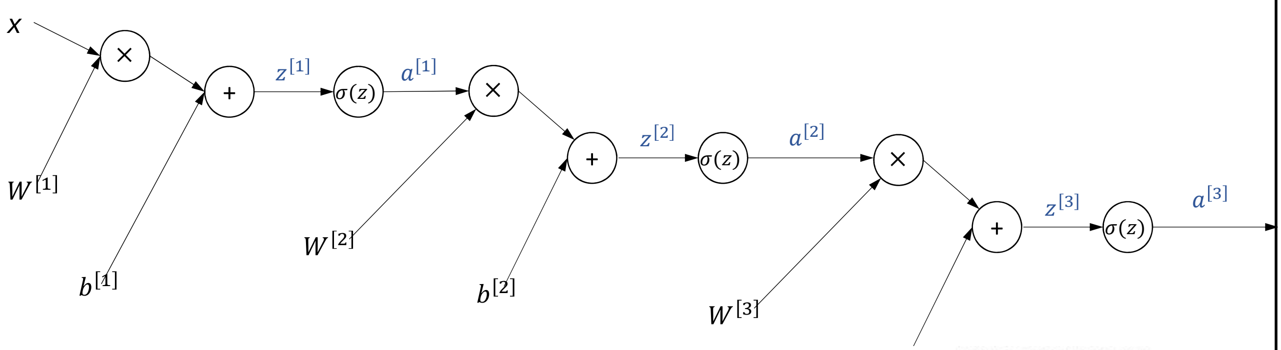


exp are expensive to compute

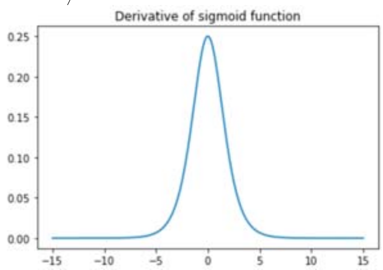
Outputs are not zero-centered *

68/120





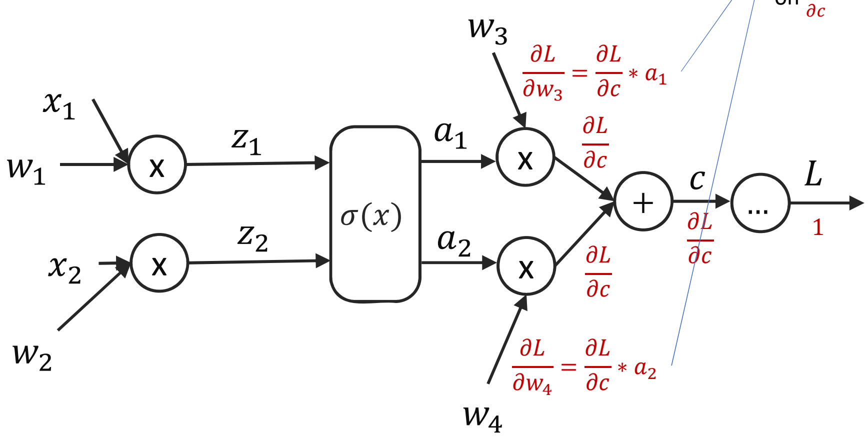
$$\frac{\partial L}{\partial W^{[1]}} = X\underbrace{\sigma'(z_1)}_{<0.25}W^{[2]}\underbrace{\sigma'(z_2)}_{<0.25}W^{[3]}\frac{\partial L}{\partial \sigma(z_3)}$$



Why do we want zero-centered activations? (and zero-mean data)

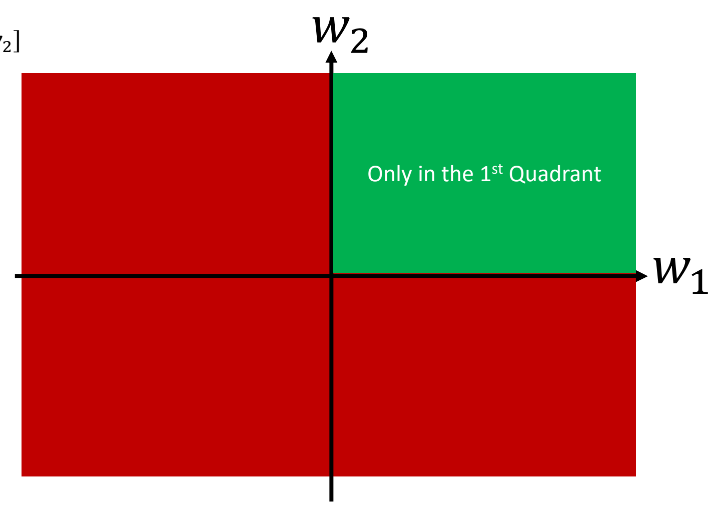
Consider what happens when the input x_i is always positive.

a_1 and a_2 are positive because of the sigmoid function so the gradients will either be both negative or both positive depending on $\frac{\partial L}{\partial c}$



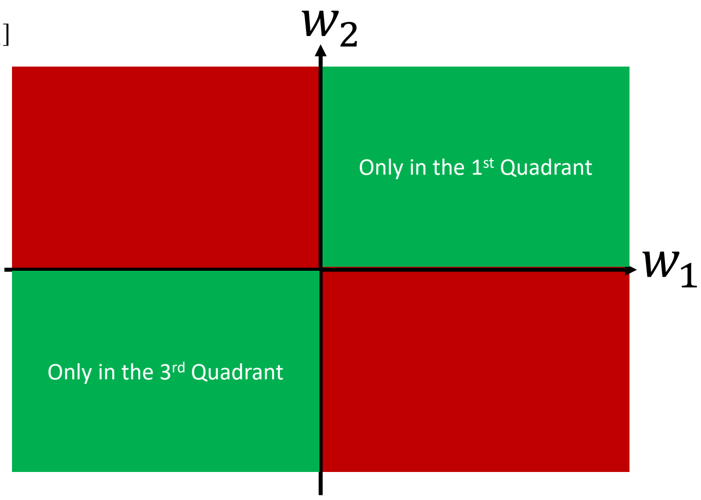
What directions can we represent if $w_1 > 0$ and $w_2 > 0$?

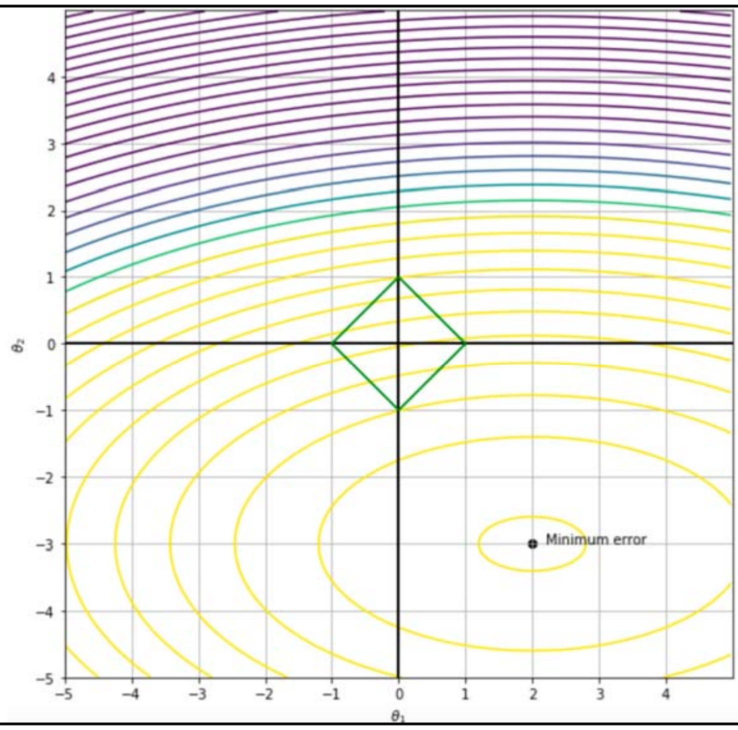
Let $w = [w_1, w_2]$



What directions can we represent if $w_1 < 0$ and $w_2 < 0$?

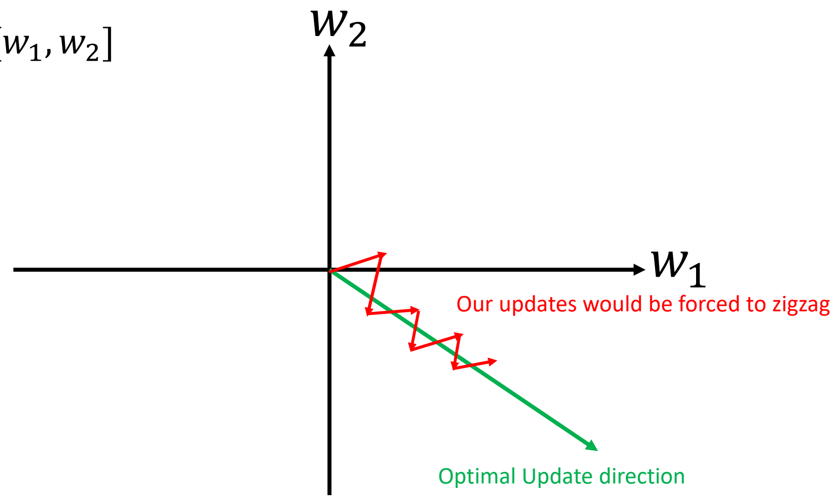
Let $w = [w_1, w_2]$





What if the gradients can only be all positive of all negative but optimal update direction is in the 4th quadrant?

Let $w = [w_1, w_2]$

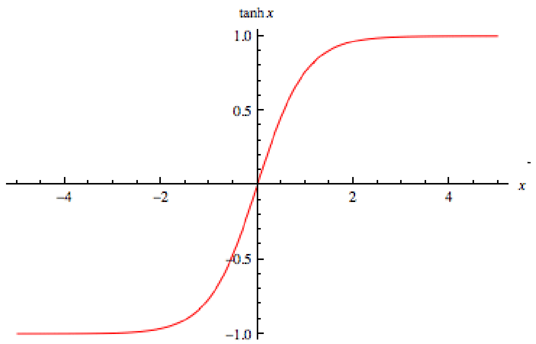


tanh (hyperbolic tangent)

$$\tanh(x) = \frac{e^{x} - e^{-x}}{e^{x} + e^{-x}} = \frac{e^{2x} - 1}{e^{2x} + 1} = \frac{1 - e^{-2x}}{1 + e^{-2x}}$$

Scaled version of the sigmoid function

$$\tanh(x) = 2\sigma(2x) - 1$$

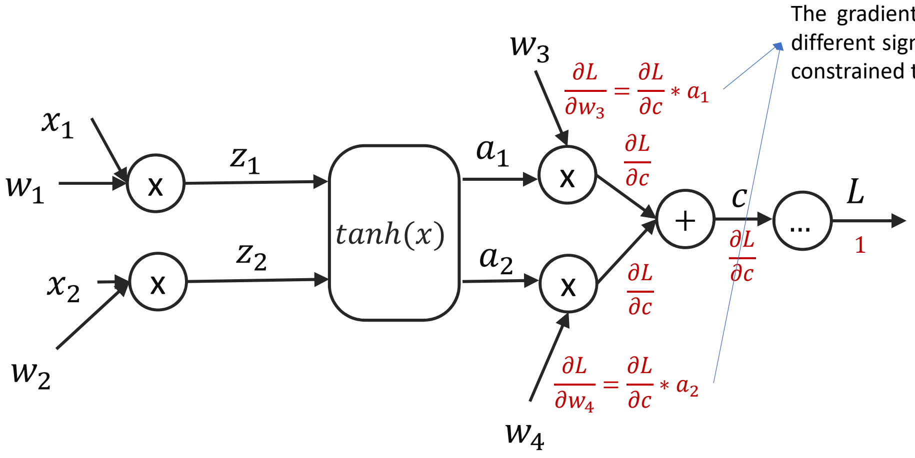


range $[-1, 1]$

Outputs are zero-centered

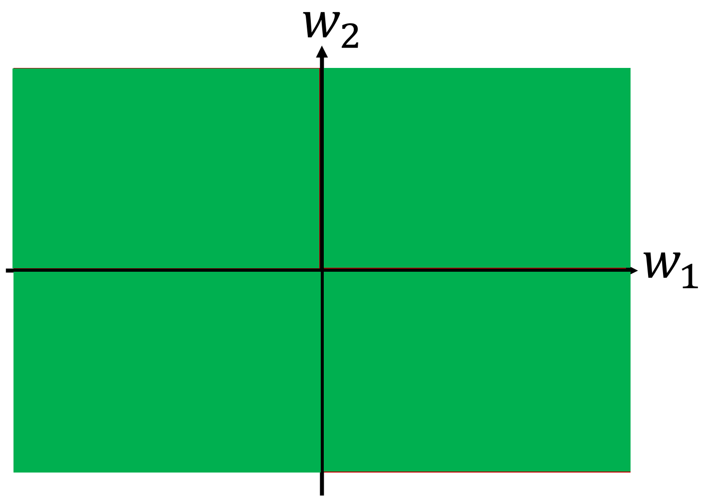
Saturated (near -1 or 1) neurons ‘kill’ the gradients

exp are expensive to compute



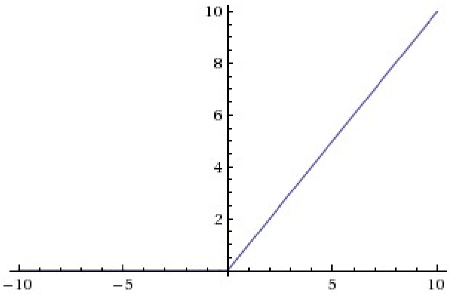
The gradients of w_3 and w_4 can now have different signs since a_1 and a_2 are no longer constrained to only be positive.

Since w_1 and w_2 can have different signs we can now update in any direction



ReLU (rectified linear unit)

$$\mathrm{ReLU(x)} = \max(0, x)$$



range $[0, \infty]$

Does not saturate if positive

Outputs are not zero-centered

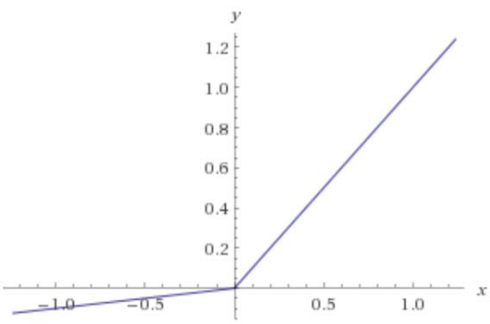
Neurons still die
- gradient is 0 when $x < 0$
- dead neurons won't activate/update

No exp, computationally efficient

In practice, converges faster (~6x faster)

leaky ReLU (leaky rectified linear unit)

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



Does not saturate

Outputs are not zero-centered

Will not die

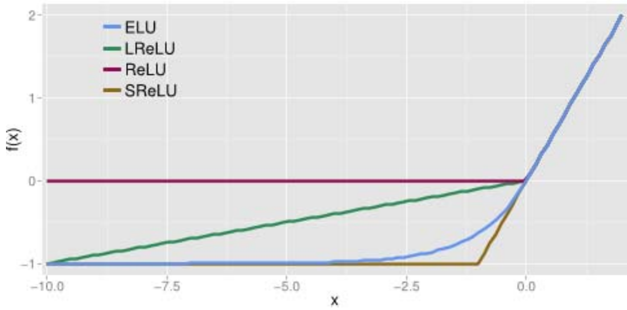
No exp, computationally efficient

Parametric Rectifier (PReLU)

$$f(x) = \max(\alpha x, x)$$

ELU (exponential linear unit)

$$f(x) = \begin{cases} x & \text{if } x > 0 \\ \alpha(\exp(x) - 1) & \text{if } x \le 0 \end{cases}$$



Does not saturate

Outputs are more zero-centered

Will not die

exp, slight annoyance

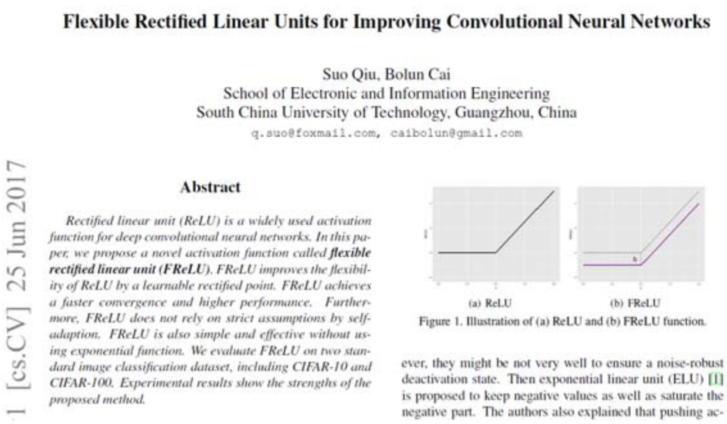
In Practice

- Use ReLU as a default activation function.
- Works very well in most problems
- Stops the propagation of “useless” information.
- Saturating on negative could be an advantage
- Simpler and faster than **Elu and LeakyRelu**

In Practice

- In practice sigmoid and tanh are not used for hidden layers.
- It saturates on positive and negative, which makes it hard to use.

Some Studies On Activations



Flexible Rectified Linear Units for Improving Convolutional Neural Networks

Suo Qiu, Bolun Cai
School of Electronic and Information Engineering
South China University of Technology, Guangzhou, China
q.suo@foxmail.com, caibolun@gmail.com

1 [cs.CV] 25 Jun 2017

Abstract

Rectified linear unit (ReLU) is a widely used activation function for deep convolutional neural networks. In this paper, we propose a novel activation function called ***flexible rectified linear unit (FReLU)****. FReLU improves the flexibility of ReLU by a learnable rectified point. FReLU achieves a faster convergence and higher performance. Furthermore, FReLU does not rely on strict assumptions by self-adaption. FReLU is also simple and effective without using exponential function. We evaluate FReLU on two standard image classification dataset, including CIFAR-10 and CIFAR-100. Experimental results show the strengths of the proposed method.*

(a) ReLU (b) FReLU

Figure 1. Illustration of (a) ReLU and (b) FReLU function.

ever, they might be not very well to ensure a noise-robust deactivation state. Then exponential linear unit (ELU) [1] is proposed to keep negative values as well as saturate the negative part. The authors also explained that pushing ac-

Some Studies On Activations

1 / 102

9 page paper with 90+ appendix

Self-Normalizing Neural Networks

5v5 [cs.]

neural networks (SNNs) to enable high-level abstract representations. While batch normalization requires explicit normalization, neuron activations of SNNs automatically converge towards zero mean and unit variance. The activation function of SNNs are "scaled exponential linear units" (SELUs), which induce self-normalizing properties. Using the Banach fixed-point theorem, we prove that activations close to zero mean and unit variance that are propagated through many

Some Studies On Activations

SHIFTING MEAN ACTIVATION TOWARDS ZERO WITH BIPOLAR ACTIVATION FUNCTIONS

Lars H. Eidnes
Trondheim
Norway
larseidnes@gmail.com

Arild Nøkland
Trondheim
Norway
arild.nokland@gmail.com

:ML] 18 Dec 2017

ABSTRACT

We propose a simple extension to the ReLU-family of activation functions that allows them to shift the mean activation across a layer towards zero. Combined with proper weight initialization, this alleviates the need for normalization layers. We explore the training of deep vanilla recurrent neural networks (RNNs) with up to 144 layers, and show that bipolar activation functions help learning in this setting. On the Penn Treebank and Text8 language modeling tasks we obtain competitive results, improving on the best reported results for non-gated networks. In experiments with convolutional neural networks without batch normalization, we find that bipolar activations produce a faster drop in training error, and results in a lower test error on the CIFAR-10 classification task. [1]

Some Studies On Activations

Deep ReLU Networks Have Surprisingly Few Activation Patterns

Boris Hanin
Facebook AI Research
Texas A&M University
bhanin@math.tamu.edu

David Rolnick
University of Pennsylvania
Philadelphia, PA USA
drolnick@seas.upenn.edu

Abstract

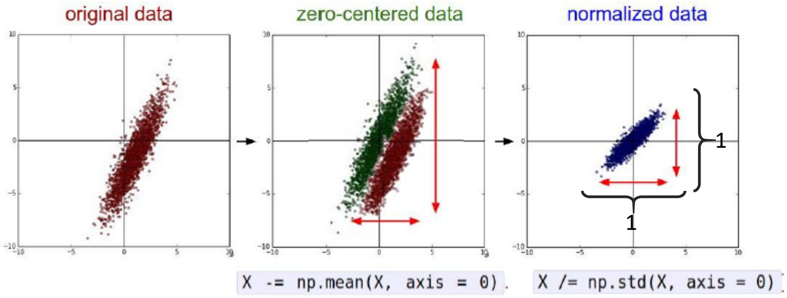
The success of deep networks has been attributed in part to their expressivity: per parameter, deep networks can approximate a richer class of functions than shallow networks. In ReLU networks, the number of activation patterns is one measure of expressivity; and the maximum number of patterns grows exponentially with the depth. However, recent work has showed that the practical expressivity of deep networks – the functions they can learn rather than express – is often far from the theoretical maximum. In this paper, we show that the average number of activation patterns for ReLU networks at initialization is bounded by the total number of

Common Data Pre-processing

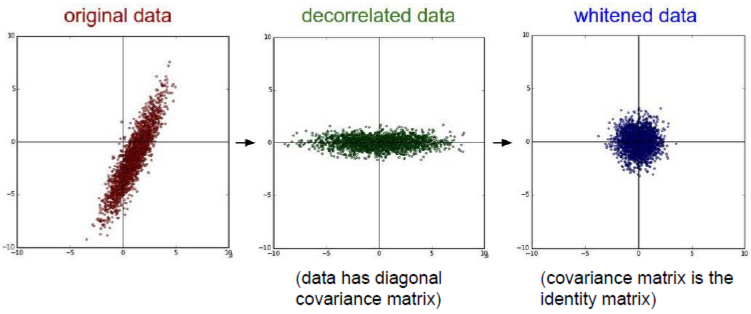
Standardization

sometimes also called normalization but different from min-max normalization

$$\frac{x-\mu}{\sigma}$$



PCA and whitening

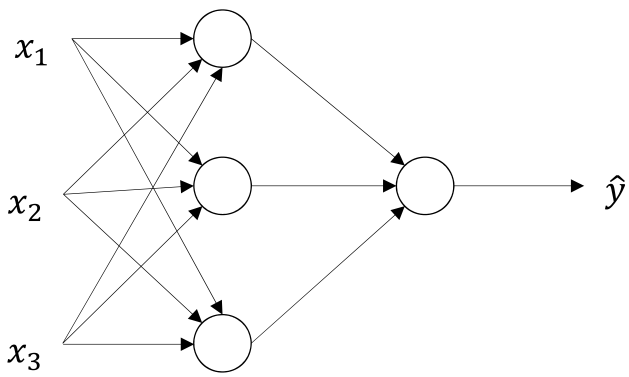


PCA and Whitening is not commonly used in images

Weights initialization

Weights initialization

What happens if we initialize weights to a constant value?



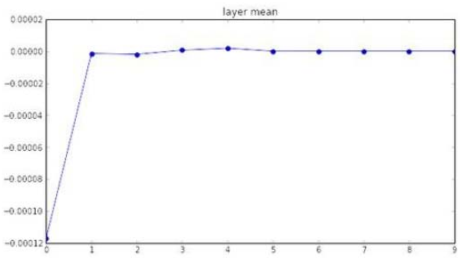
Weights initialization

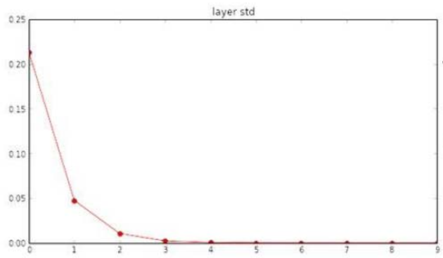
- Traditionally, the weights of the neural networks are initialized with small random numbers.
  - Gaussian with 0 mean and 1e-2 standard deviation.
  - This works okay for small networks but can lead to non-homogeneous distributions of activations across the layers of a network

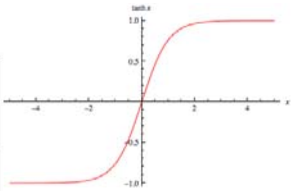

```
hidden layer 2 had mean -0.000001 and std 0.047551
hidden layer 3 had mean -0.000002 and std 0.010630
hidden layer 4 had mean 0.000001 and std 0.002378
hidden layer 5 had mean 0.000002 and std 0.000532
hidden layer 6 had mean -0.000000 and std 0.000119
hidden layer 7 had mean 0.000000 and std 0.000026
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
```

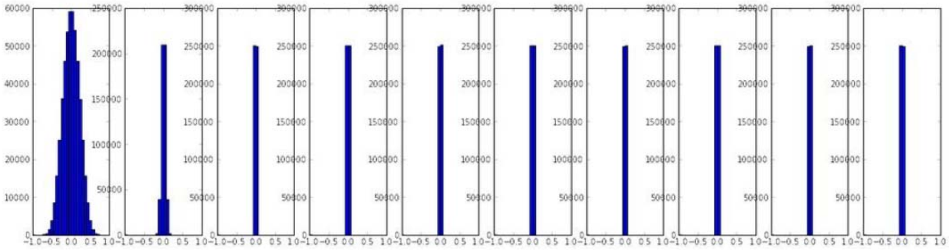
Initialize weights with mean=0 and std=0.01

tanh activation function









Assume input is mean=0 and std = 1

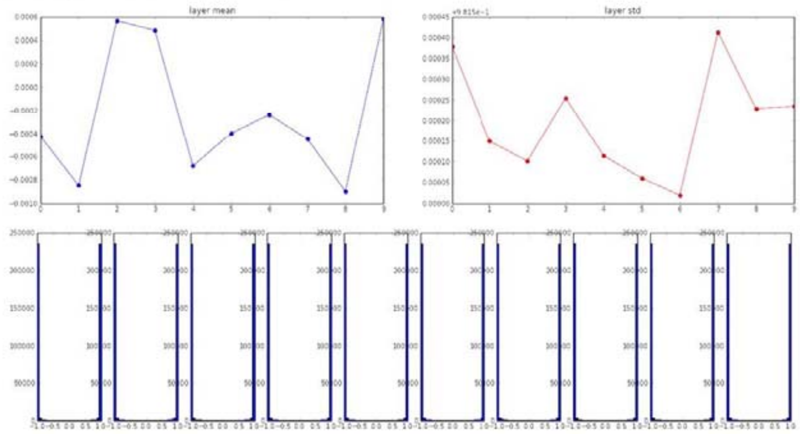
Weights initialization

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

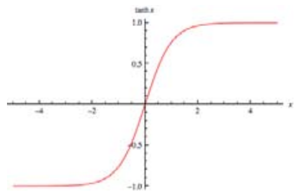
```
input layer had mean 0.001800 and std 1.001311
hidden layer 1 had mean -0.000430 and std 0.981879
hidden layer 2 had mean -0.000849 and std 0.981649
hidden layer 3 had mean 0.000566 and std 0.981601
hidden layer 4 had mean 0.000483 and std 0.981755
hidden layer 5 had mean -0.000682 and std 0.981614
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.



tanh activation function



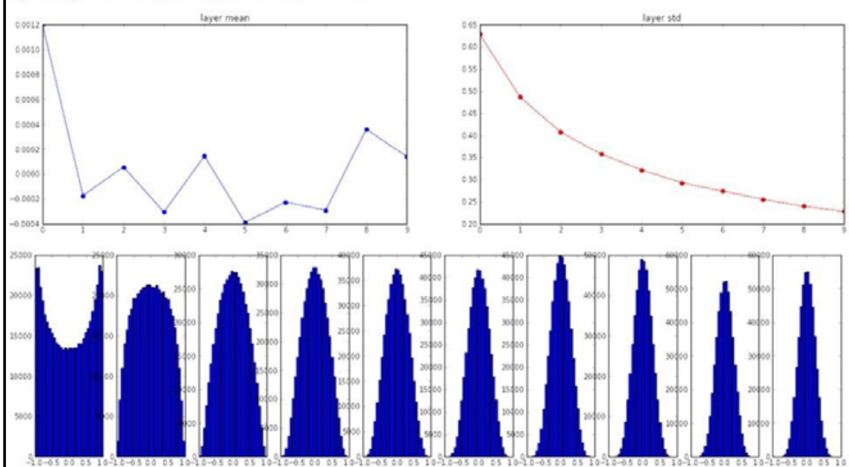
Assume input is mean=0 and std = 1

Weights 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
```

"Xavier initialization" [Glorot et al., 2010]



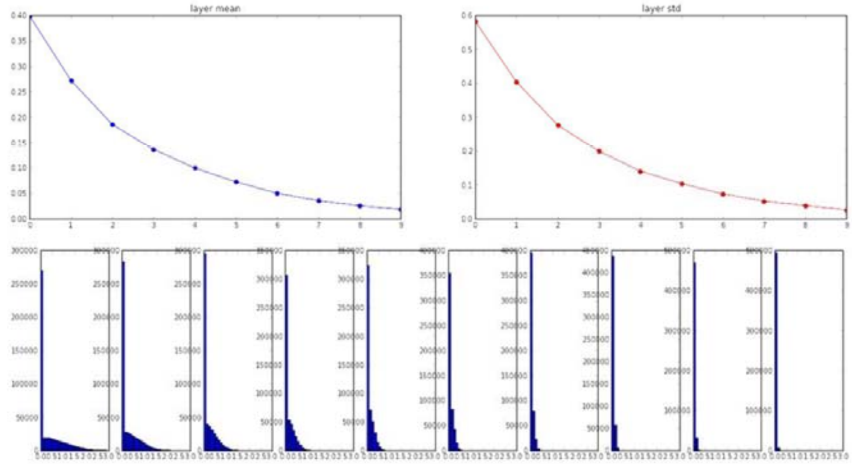
Reasonable initialization. (Mathematical derivation assumes linear activations)

Weights 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.

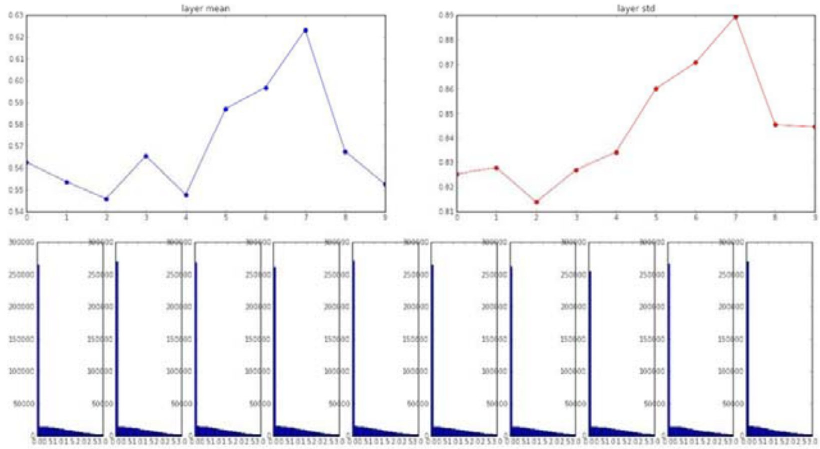


Weights initialization

```
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
```

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

He et al., 2015
(note additional /2)



Proper initialization is still an active area of research

- ***Understanding the difficulty of training deep feedforward neural networks***
  - by Glorot and Bengio, 2010
- ***Exact solutions to the nonlinear dynamics of learning in deep linear neural networks***
  - By Saxe et al, 2013
- ***Random walk initialization for training very deep feedforward networks***
  - By Sussillo and Abbott, 2014
- ***Delving deep into rectifiers: Surpassing human-level performance on ImageNet classification***
  - By He et al., 2015
- ***Data-dependent Initializations of Convolutional Neural Networks***
  - By Krähenbühl et al., 2015
- ***All you need is a good init***,
  - By Mishkin and Matas, 2015

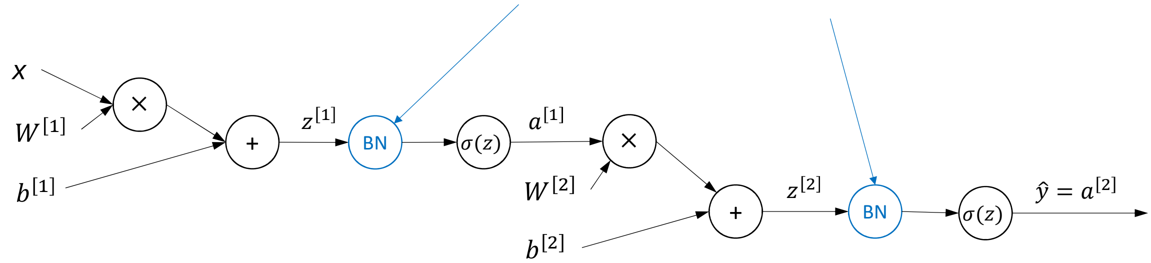
Batch Normalization

“you want unit Gaussian activations? Just make them so.”

Batch normalization, Ioffe and Szegedy, 2015

- Consider a batch of activations at some layer k:
  - $\hat{x}^{(k)} = \frac{x^{(k)} - E\left[x^{(k)}\right]}{\sqrt{\mathrm{Var}[x^{(k)}]}}$

Batch normalization is usually inserted before the activation functions



Normalize:

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

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

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

Note, the network can learn:

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

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

to recover the identity mapping.

Weights initialization > Batch normalization

Batch Normalization

- Improves Gradient Flow through the network
- Allows higher learning rates
- Reduces the strong dependence on initialization
- Has a regularizing effect

- Note: at test time the batch norm layer functions differently
- We should use the mean and std computed from the training data and not the test.

Input: Values of x over a mini-batch: $\mathcal{B} = \{x_{1\ldots m}\}$;
Parameters to be learned: γ, β
Output: $\{y_i = \mathrm{BN}_{\gamma,\beta}(x_i)\}$

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

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

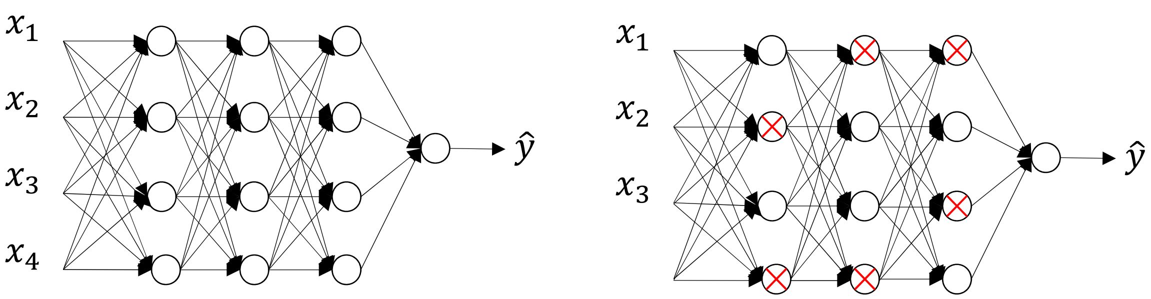
$$\widehat{x}_i \leftarrow \frac{x_i - \mu_{\mathcal{B}}}{\sqrt{\sigma_{\mathcal{B}}^2 + \epsilon}} \qquad \text{// normalize}$$

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

Weights initialization > Dropout

Dropout

Intuition: Can't rely on any single feature, so we have to spread out the weights. (closely related to ensemble methods)

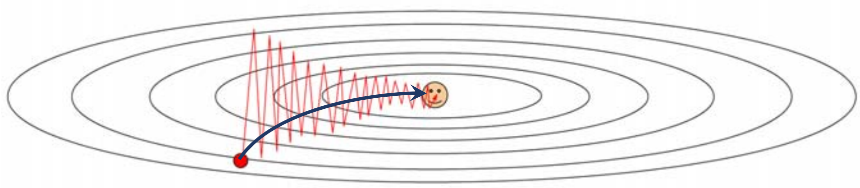


In practice we implement the "inverted dropout". We divide the weights by the keep-prob during training to maintain the magnitude / statistics of the activations. At test time, the activations are untouched.

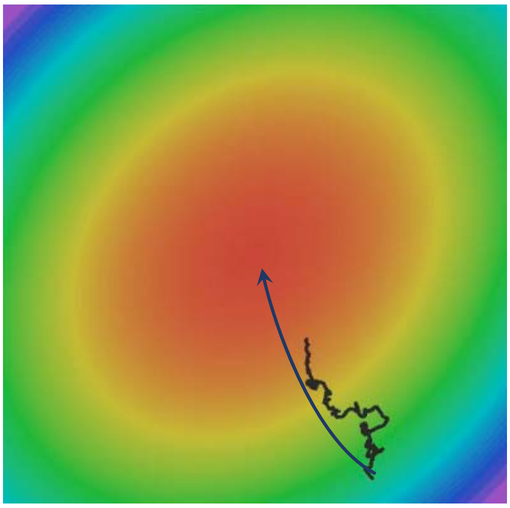
Speeding up the optimization

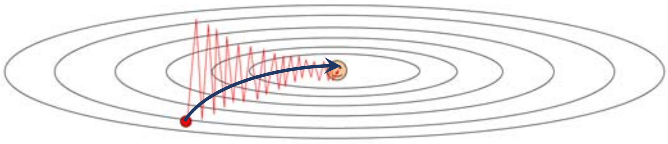
Optimization / Parameter Updates

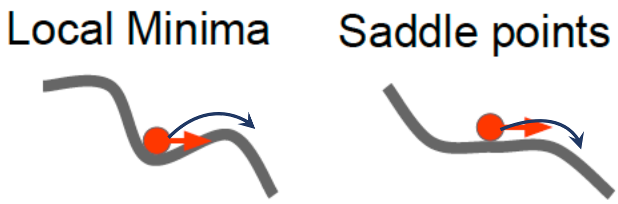
What if the loss function is steep vertically but shallow horizontally



We could be updating one feature more over another, so we move in zigzags







What will happen if learning rate is high? oscillations will be exaggerated.

(Digression) Exponentially Weighted (Moving) averages

Moving averages / moving mean

- Low pass filter (smoothing effect)

Given a set of data points / signal θ_i, where $v_0 = 0$.

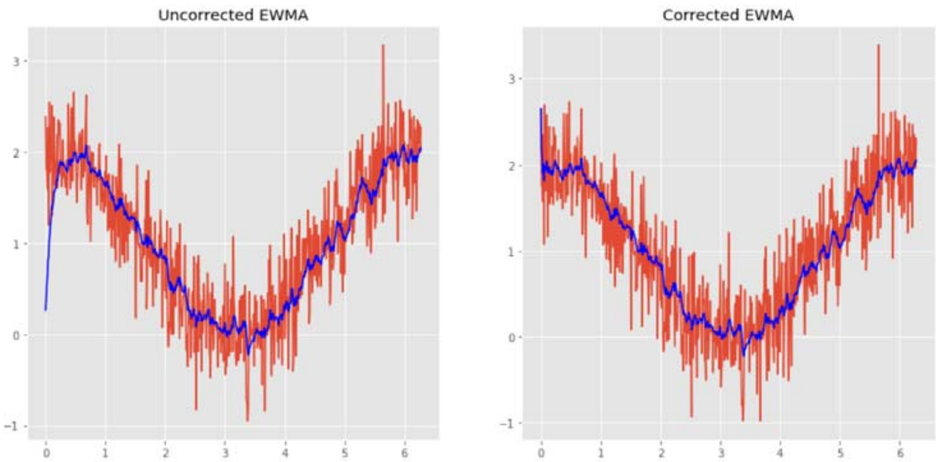
$$v_t = \beta\, v_{t-1} + (1-\beta)\theta_t$$

v_t - smoothened version of the signal.

$\frac{1}{1-\beta}$ - effective horizon / window

*Bias correction : $\frac{v_t}{1-\beta^t}$

(Digression) Exponentially Weighted (Moving) averages



Gradient Descent with Momentum

Problem: Oscillating updates makes learning slow and prevents us from using higher learning rates.

Idea: Apply Exponentially Weighted Moving Averages on the gradient updates / gradient directions.

Physics interpretation: allows velocity to build up along shallow directions and damped oscillations in steep directions due to quickly changing signs

On iteration t:

- Compute the gradients $\frac{\partial L}{\partial W}, \frac{\partial L}{\partial b}$ on the current mini-batch
- $V_{\partial W} = \beta V_{\partial W} + (1-\beta)\frac{\partial L}{\partial W}$ → bias corrected → $V_{\partial W} = \frac{V_{\partial W}}{1-\beta^t}$
- $V_{\partial b} = \beta V_{\partial b} + (1-\beta)\frac{\partial L}{\partial b}$ → bias corrected → $V_{\partial b} = \frac{V_{\partial b}}{1-\beta^t}$

When updating the weights, we now use the moving average instead of the actual gradient direction

- $W = W - \alpha V_{\partial W}$
- $b = b - \alpha V_{\partial b}$

$\beta = 0.9$ Usually works well in practice

Gradient Descent with Momentum

How do we make it trust its current trajectory? Add momentum!

$$V_{\partial W} = \beta V_{\partial W} + (1-\beta)\frac{\partial L}{\partial W}$$

$$V_{\partial b} = \beta V_{\partial b} + (1-\beta)\frac{\partial L}{\partial b}$$

$$W = W - \alpha V_{\partial W}$$
$$b = b - \alpha V_{\partial b}$$

	SGD	Momentum
1	100	
2	100	
3	1	
4	100	
5	1000	

Gradient Descent with RMSProp (Root Mean Squared Prop)

Or, maybe we can make the steps equal among the features…

What will happen over time?

RMSProp tends to halt when denominator grows big.

$$S_{\partial W} = \frac{\beta S_{\partial W} + (1-\beta)\partial W^2}{1-\beta^t}$$

$$S_{\partial b} = \beta S_{\partial b} + \frac{(1-\beta)\partial b^2}{1-\beta^t}$$

$$W = W - \alpha\frac{\partial W}{\sqrt{S_{\partial W}}+\epsilon}$$

$$b = b - \alpha\frac{\partial b}{\sqrt{S_{\partial b}}+\epsilon}$$

$\beta = 0.999$ Usually works well in practice

RMSProp-Root Mean Squared Prop

Fun Fact: RMSProp was first introduced in a slide of Geoff Hinton's Coursera class (lecture 6). It was then 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.

Coursera Slide

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).

Adam – Adaptive Moment Estimation

Combine momentum and RMSProp

Each Reduces oscillating updates and we combine both

Momentum: updates weights using running average.

RMSProp: Updates weights using normalized gradients

$$V_{\partial W} = \frac{\beta_1 V_{\partial W} + (1-\beta_1)\partial W}{1-\beta_1^t}$$

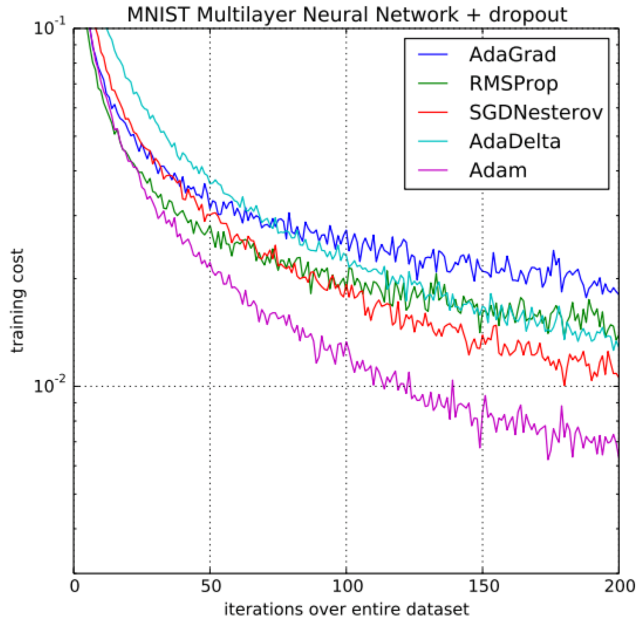
$$V_{\partial b} = \frac{\beta_1 V_{\partial b} + (1-\beta_1)\partial b}{1-\beta_1^t}$$

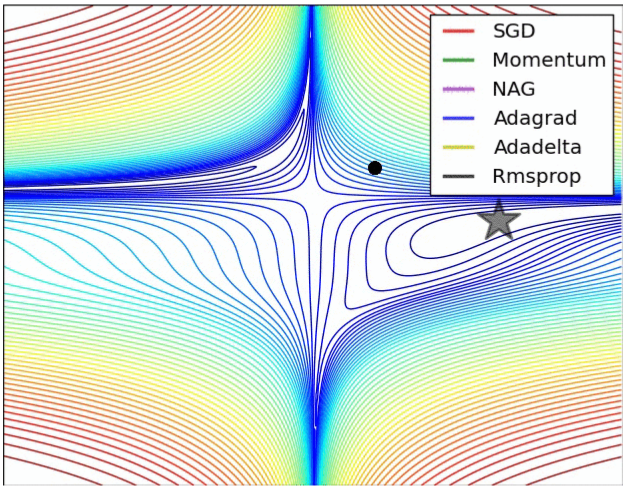
$$S_{\partial W} = \frac{\beta_2 S_{\partial W} + (1-\beta_2)\partial W^2}{1-\beta_2^t}$$

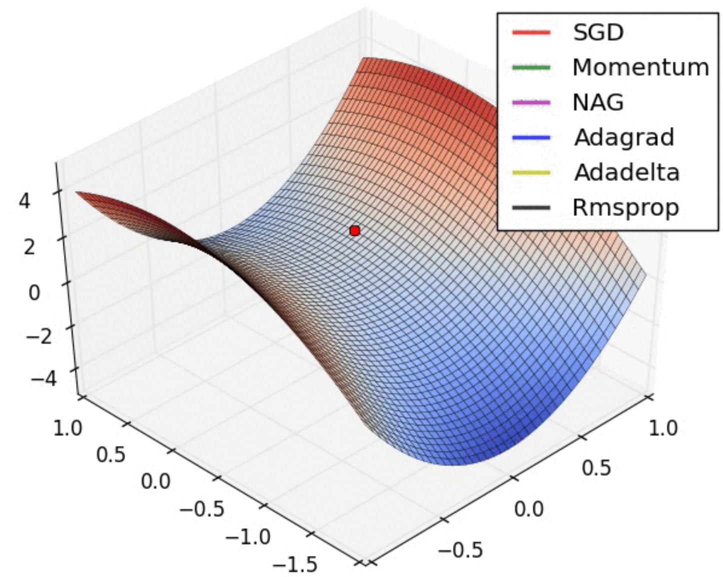
$$S_{\partial b} = \frac{\beta_2 S_{\partial b} + (1-\beta_2)\partial b^2}{1-\beta_2^t}$$

$$W = W - \alpha\frac{V_{\partial W}}{\sqrt{S_{\partial W}} + \epsilon}$$

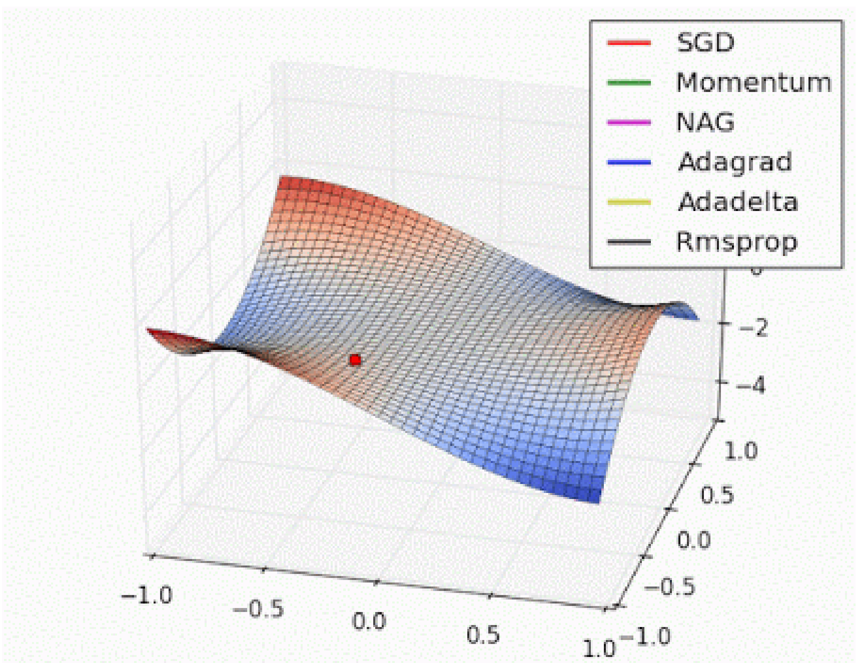
$$b = b - \alpha\frac{V_{\partial b}}{\sqrt{S_{\partial b}} + \epsilon}$$


MNIST Multilayer Neural Network + dropout
AdaGrad
RMSProp
SGDNesterov
AdaDelta
Adam
training cost
10^{-1}
10^{-2}
0
50
100
150
200
iterations over entire dataset


SGD
Momentum
NAG
Adagrad
Adadelta
Rmsprop



117/120



118/120

In practice

- Adam is a good default choice in most cases
  - $\beta_1 = 0.9$ (Good default choice)
  - $\beta_2 = 0.999$ (Good default choice)
  - α – still has to be tuned
  - $\epsilon = 10^{-8}$ (does not really affect performance much)
- Slowly Decay learning rate over time
  - Usually decay every epoch (0.95 or 0.9 are common choices)
  - Some people manually decay the learning rate by monitoring the training process
- As for the problem of local minima, recent researches have pointed out that it is much less of an issue in very high dimensional space (such as deep neural networks) points with gradients 0 are much more likely to be a saddle point. (Dauphin et al., 2014)
  - Unlikely to get stuck in bad local optima
  - Though plateaus can make learning slow

119/120

Recently proposed

Published as a conference paper at ICLR 2020

ON THE VARIANCE OF THE ADAPTIVE LEARNING RATE AND BEYOND

Liyuan Liu *
University of Illinois, Urbana-Champaign
ll2@illinois

Haoming Jiang †
Georgia Tech
jianghm@gatech.edu

Pengcheng He, Weizhu Chen
Microsoft Dynamics 365 AI
{penhe,wzchen}@microsoft.com

Xiaodong Liu, Jianfeng Gao
Microsoft Research
{xiaodl,jfgao}@microsoft.com

Jiawei Han
University of Illinois, Urbana-Champaign
hanj@illinois

G] 10 Mar 2020

ABSTRACT

The learning rate warmup heuristic achieves remarkable success in stabilizing training, accelerating convergence and improving generalization for adaptive stochastic optimization algorithms like RMSprop and Adam. Pursuing the theory behind warmup, we identify a problem of the adaptive learning rate – its variance is problematically large in the early stage, and presume warmup works as a variance reduction technique. We provide both empirical and theoretical evidence

120/120

Recently proposed

Published as a conference paper at ICLR 2019

A CLOSER LOOK AT DEEP LEARNING HEURISTICS: LEARNING RATE RESTARTS, WARMUP AND DISTILLATION

Akhilesh Gotmare*
Department of Computer Science
EPFL, Switzerland
akhilesh.gotmare@epfl.ch

Nitish Shirish Keskar, Caiming Xiong & Richard Socher
Salesforce Research
Palo Alto, US
{nkeskar, cxiong, rsocher}@salesforce.com

ABSTRACT

The convergence rate and final performance of common deep learning models have significantly benefited from heuristics such as learning rate schedules, knowledge distillation, skip connections, and normalization layers. In the absence of theoretical underpinnings, controlled experiments aimed at explaining these strategies can aid our understanding of deep learning landscapes and the training dynamics. Existing approaches for empirical analysis rely on tools of linear interpolation and visualizations with dimensionality reduction, each with their limitations. Instead, we revisit such analysis of heuristics through the lens of recently proposed methods for loss surface and representation analysis, viz., mode con-

Recently proposed

[stat.ML] 15 Apr 2019

How Does Batch Normalization Help Optimization?

Shibani Santurkar* MIT shibani@mit.edu
Dimitris Tsipras* MIT tsipras@mit.edu
Andrew Ilyas* MIT ailyas@mit.edu
Aleksander Mądry MIT madry@mit.edu

Abstract

Batch Normalization (BatchNorm) is a widely adopted technique that enables faster and more stable training of deep neural networks (DNNs). Despite its pervasiveness, the exact reasons for BatchNorm's effectiveness are still poorly understood. The popular belief is that this effectiveness stems from controlling the change of the layers' input distributions during training to reduce the so-called "internal covariate shift". In this work, we demonstrate that such distributional stability of layer inputs has little to do with the success of BatchNorm. Instead, we uncover a more fundamental impact of BatchNorm on the training process: it makes the optimization landscape significantly smoother. This smoothness induces a more predictive and stable behavior of the gradients, allowing for faster training.

1 Introduction

Recently proposed

Learning to learn by gradient descent by gradient descent

Marcin Andrychowicz[1], Misha Denil[1], Sergio Gómez Colmenarejo[1], Matthew W. Hoffman[1], David Pfau[1], Tom Schaul[1], Brendan Shillingford[1,2], Nando de Freitas[1,2,3]

[1]Google DeepMind [2]University of Oxford [3]Canadian Institute for Advanced Research

marcin.andrychowicz@gmail.com
{mdenil,sergomez,mwhoffman,pfau,schaul}@google.com
brendan.shillingford@cs.ox.ac.uk, nandodefreitas@google.com

cs.NE] 30 Nov 2016

Abstract

The move from hand-designed features to learned features in machine learning has been wildly successful. In spite of this, optimization algorithms are still designed by hand. In this paper we show how the design of an optimization algorithm can be cast as a learning problem, allowing the algorithm to learn to exploit structure in the problems of interest in an automatic way. Our learned algorithms, implemented by LSTMs, outperform generic, hand-designed competitors on the tasks for which

Recently proposed

YELLOWFIN and the Art of Momentum Tuning

Jian Zhang, Ioannis Mitliagkas, Christopher Ré
Department of Computer Science
Stanford University
{zjian,imit,chrismre}@cs.stanford.edu

June 13, 2017

1 [stat.ML] 12 Jun 2017

Abstract

Hyperparameter tuning is one of the big costs of deep learning. State-of-the-art optimizers, such as Adagrad, RMSProp and Adam, make things easier by adaptively tuning an individual learning rate for each variable. This level of fine adaptation is understood to yield a more powerful method. However, our experiments, as well as recent theory by Wilson et al. [1], show that hand-tuned stochastic gradient descent (SGD) achieves better results, at the same rate or faster. The hypothesis put forth is that adaptive methods converge to different minima [1]. Here we point out another factor: none of these methods tune their momentum parameter, known to be very important for deep learning applications [2]. Tuning the momentum parameter becomes even more important in asynchronous-parallel systems: recent theory [3] shows that asynchrony introduces momentum-like dynamics, and that tuning down algorithmic momentum is important for efficient parallelization

Recently proposed

Learning to Optimize

Ke Li **Jitendra Malik**
Department of Electrical Engineering and Computer Sciences
University of California, Berkeley
Berkeley, CA 94720
United States
{ke.li,malik}@eecs.berkeley.edu

1 [cs.LG] 6 Jun 2016

Abstract

Algorithm design is a laborious process and often requires many iterations of ideation and validation. In this paper, we explore automating algorithm design and present a method to *learn* an optimization algorithm, which we believe to be the first method that can automatically discover a better algorithm. We approach this problem from a reinforcement learning perspective and represent any particular optimization algorithm as a policy. We learn an optimization algorithm using guided policy search and demonstrate that the resulting algorithm outperforms existing hand-engineered algorithms in terms of convergence speed and/or the final objective value.

125/120

Recently proposed

Decoupled Neural Interfaces using Synthetic Gradients

Max Jaderberg [1] **Wojciech Marian Czarnecki** [1] **Simon Osindero** [1] **Oriol Vinyals** [1] **Alex Graves** [1] **David Silver** [1] **Koray Kavukcuoglu** [1]

3v2 [cs.LG] 3 Jul 2017

Abstract

Training directed neural networks typically requires forward-propagating data through a computation graph, followed by backpropagating error signal, to produce weight updates. All layers, or more generally, modules, of the network are therefore locked, in the sense that they must wait for the remainder of the network to execute forwards and propagate error backwards before they can be updated. In this work we break this constraint by decoupling modules by introducing a model of the future computation of the network graph. These models predict what the result of the modelled subgraph will produce using only local information. In particular we focus on modelling error gradients: by using the modelled *synthetic gradient* in place of true backpropagated error gradients we decouple subgraphs, and can update them independently and asyn-

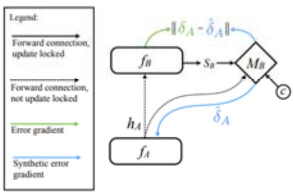


Figure 1. General communication protocol between A and B. After receiving the message h_A from A, B can use its model of A, M_B, to send back *synthetic gradients* $\hat{\delta}_A$ which are trained to approximate real error gradients δ_A. Note that A does not need to wait for any extra computation after itself to get the correct error gradients, hence decoupling the backward computation. The feedback model M_B can also be conditioned on any privileged information or context, c, available during training such as a label.

126/120

Next Homework: Logistic Regression

- It will be posted on tonight.
- Deadline will be next week Monday .