Backpropagation

Alexander Binder

University of Oslo (UiO)

February 9, 2021

UiO : Department of Informatics
University of Oslo

Learning goals

- be able to explain the differences: batch gradient descent vs stochastic gradient descent
- be able to use the chain rule on functions
- Backpropagation is an algorithm to compute derivatives in a neural network (w.r.t. network parameters, and inputs)
- Backpropagation inside amounts to applying chain rule top-down along the graph structure of the network
- the backprop comes with an efficient mechanism to reuse computed partial derivatives for computing new derivatives further down in the network.
- you should be able to derive the derivative of a loss of a neural network as a sum-product of derivatives of single neurons with respect to their inputs and parameters

Learning goals

- Autograd and when to use `with torch.no_grad():`
- be able to understand how the product nature of chain rule may lead to vanishing/exploding gradients in a neural net
- be able to reproduce the key initialization points for ReLU and PReLU networks:
  - biases to zero
  - drawn weights as random numbers (why random?)
  - drawn weights from a zero mean normal
  - what are the variances for ReLU and PReLU activations

batch vs stochastic gradient descent ...

Have a nested function composed of neural network layers $f^{(l)}(z, w)$.
$f^{(l)}(z, w)$ - i-th layer with parameters w and inputs to the layer z.

A single layer (fully connected layer):

$$z^{(l)} = f(w^{(l)} \cdot z^{(l-1)} + b) = f(w^{(l)}, z^{(l-1)})$$
$$z \in \mathbb{R}^{d_1},\ \ w^{(l)} \in \mathbb{R}^{d_2 \times d_1},\ \ b \in \mathbb{R}^{d_2}$$

$f(w^{(l)} \cdot z + b)$ is the layer definition, $f(w^{(l)}, z)$ is a notation to denote dependency on parameters $w^{(i)}$ and the input to the network layer z.

A three layer network can be described as:

$$y = f^{(3)}(w^{(3)}, f^{(2)}(w^{(2)}, f^{(1)}(w^{(1)}, x)))$$

An iterative writing would be:

$$z^{(l)} = f^{(l)}(w^{(l)}, z^{(l-1)})$$

A three layer network can be described as:

$$z^{(l)} = f^{(l)}(w^{(l)}, z^{(l-1)})$$
$$y = f^{(3)}(w^{(3)}, f^{(2)}(w^{(2)}, f^{(1)}(w^{(1)}, x)))$$

goal: to compute $\frac{\partial y}{\partial w_d^{(l)}}$ fast.

How to compute gradients generically for some function (if we had no autograd) ?

Finite difference method:

$$\begin{aligned}\frac{\partial f}{\partial w_d}(w) &= \lim_{\epsilon \to 0} \frac{f(x, w \setminus \{w_d\}, w_d + \epsilon) - f(x, w \setminus \{w_d\}, w_d)}{\epsilon} \\ &\approx \frac{f(x, w \setminus \{w_d\}, w_d + \epsilon) - f(x, w \setminus \{w_d\}, w_d)}{\epsilon}\end{aligned}$$

note: better is however:

$$\frac{\partial f}{\partial w_d}(w) \approx \frac{f(x, w \setminus \{w_d\}, w_d + \epsilon) - f(x, w \setminus \{w_d\}, w_d - \epsilon)}{\epsilon^2}$$

How can we compute the derivative $\frac{\partial L}{\partial w_d}$ of a loss $L(y, y_{gt})$ with respect to parameters w_d once we have $\frac{\partial y}{\partial w_d}$?

Outline

- 1-dim case: $x \in \mathbb{R}$, $g(x) \in \mathbb{R}$

$$h(x) = f \circ g(x)$$
$$\frac{\partial h}{\partial x}(x) = \frac{\partial f \circ g}{\partial x}(x) = f'(g(x))g'(x) = \frac{\partial f}{\partial z}(g(z))\frac{\partial g}{\partial x}(x)$$

- n-dim case Typically shown as:

$$f : \mathbb{R}^m \to \mathbb{R},\ f(z_1, \dots, z_m) \in \mathbb{R}$$
$$g : \mathbb{R}^d \to \mathbb{R}^m,\ g(x_1, \dots, x_d) \in \mathbb{R}^m$$
$$f \circ g(x) = f(g(x))$$
$$\frac{\partial f \circ g}{\partial x_k}(x) = \sum_{r=1}^{m} \frac{\partial f}{\partial z_r}(g(x))\frac{\partial g_r}{\partial x_k}(x)$$

Contains a non-intuitive summing: derivatives over input components of f and over output components of g.

- n-dim case Typically shown as:

$$f : \mathbb{R}^m \to \mathbb{R},\ f(z_1, \dots, z_m) \in \mathbb{R},\ \ g : \mathbb{R}^d \to \mathbb{R}^m,\ \ g(x_1, \dots, x_d) \in \mathbb{R}^m$$

$$\frac{\partial f \circ g}{\partial x_k}(x) = \sum_{r=1}^{m} \frac{\partial f}{\partial z_r}(g(x)) \frac{\partial g_r}{\partial x_k}(x)$$

  Why this summing of columns with rows?

- A derivative is a linear mapping of directions h onto directional derivatives. Represented by vector/matrix-vector/matrix multiplication $\star$:

$$f(x) \xleftrightarrow{\sim} Df(x)[\cdot],\ \ Df(x)[h] = \nabla f(x) \star h$$

- concatenation of two functions – derivative $\xleftrightarrow{\sim}$ concatenation of two linear mappings

$$f \circ g(x) \xleftrightarrow{\sim} D(f \circ g)(x)[h] = Df(g(x))[v], v = Dg(x)[h]$$

$$\frac{\partial f \circ g}{\partial x_k}(x) = \sum_{r=1}^{m} \frac{\partial f}{\partial z_r}(g(x)) \frac{\partial g_r}{\partial x_k}(x)$$

Why this summing?

- A derivative of one function is a linear mapping of directions h:

$$f(x) \overset{\sim}{\longleftrightarrow} Df(x)[\cdot],\ Df(x)[h] = \nabla f(x) \star h$$

- concatenation of two functions – derivative $\overset{\sim}{\longleftrightarrow}$ concatenation of two linear mappings:

$$f \circ g(x) \overset{\sim}{\longleftrightarrow} D(f \circ g)(x)[h] = Df(g(x))[v], v = Dg(x)[h]$$

- since linear mapping $\overset{\sim}{\longleftrightarrow}$ vector/matrix-vector/matrix multiplication (between the gradient and direction vector h),
  the concatenation of two linear mappings $\overset{\sim}{\longleftrightarrow}$ matrix-multiply as well between the matrices defining both gradients and the direction vector h:

$$\begin{aligned} D(f \circ g)(x)[h] &= Df(g(x))[v] = \nabla f(g(x)) \star v, v = Dg(x)[h] = \nabla g(x) \star h \\ &\approx \text{"}\nabla f(g(x)) \star \nabla g(x) \star h\text{"} \\ D(f \circ g)(x)[\cdot] &\approx \text{"}\nabla f(g(x)) \star \nabla g(x) \end{aligned}$$

$$\frac{\partial f \circ g}{\partial x_k}(x) = \sum_{r=1}^{m} \frac{\partial f}{\partial z_r}(g(x)) \frac{\partial g_r}{\partial x_k}(x)$$

Why this column-row type summing?

- A derivative of one function is a linear mapping of directions h. concatenation of two functions – derivative will be a concat of two linear mappings

$$f(x) \overset{\sim}{\longleftrightarrow} Df(x)[\cdot],\;\; Df(x)[h] = \nabla f(x) \star h$$

$$f \circ g(x) \overset{\sim}{\longleftrightarrow} D(f \circ g)(x)[h] = Df(g(x))[v],\, v = Dg(x)[h]$$

- since linear mapping $\overset{\sim}{\longleftrightarrow}$ vector/matrix-vector/matrix multiplication (between the gradient and direction vector h),
  the concatenation of two linear mappings $\overset{\sim}{\longleftrightarrow}$ matrix-multiply (between the matrices defining both gradients) as well:

$$\begin{aligned} D(f \circ g)(x)[h] &= Df(g(x))[v] = \nabla f(g(x)) \star v,\, v = Dg(x)[h] = \nabla g(x) \star h \\ &\approx \text{"}\nabla f(g(x)) \star \nabla g(x) \star h\text{"} \end{aligned}$$

$$D(f \circ g)(x)[\cdot] \approx \text{"}\nabla f(g(x)) \star \nabla g(x)$$

- explanation for why ...? matrix multiply causing multiplication of columns (left side) with rows (right side)

$$f : \mathbb{R}^m \to \mathbb{R},\ f(z_1, \ldots, z_m) \in \mathbb{R}$$

$$g : \mathbb{R}^d \to \mathbb{R}^m,\ g(x_1, \ldots, x_d) \in \mathbb{R}^m$$

$$f \circ g(x) = f(g(x))$$

$$\frac{\partial f \circ g}{\partial x_k}(x) = \sum_{r=1}^{m} \frac{\partial f}{\partial z_r}(g(x)) \frac{\partial g_r}{\partial x_k}(x)$$

$$v_r = \nabla f(g(x))_r := \frac{\partial f}{\partial z_r}(g(x))$$

$$J_{kr} := \frac{\partial g_r}{\partial x_k}(x)$$

$$\Rightarrow \frac{\partial f \circ g}{\partial x_k}(x) = \sum_{r=1}^{m} J_{kr} v_r = (J \star v)_k$$

See: the k-th partial derivative is the k-th component of matrix-vector product $J \star v$, $\star$ explicit for matrix multiplication

$$f : \mathbb{R}^m \to \mathbb{R},\ f(z_1, \ldots, z_m) \in \mathbb{R}$$

$$g : \mathbb{R}^d \to \mathbb{R}^m,\ g(x_1, \ldots, x_d) \in \mathbb{R}^m$$

$$f \circ g(x) = f(g(x))$$

$$\frac{\partial f \circ g}{\partial x_k}(x) = \sum_{r=1}^{m} \frac{\partial f}{\partial z_r}(g(x)) \frac{\partial g_r}{\partial x_k}(x)$$

$$v_r = \nabla f(g(x))_r := \frac{\partial f}{\partial z_r}(g(x))$$

$$J_{kr} := \frac{\partial g_r}{\partial x_k}(x)$$

$$(!) \Rightarrow \frac{\partial f \circ g}{\partial x_k}(x) = \sum_{r=1}^{m} J_{kr} v_r = (J \star v)_k$$

$$\text{vectorize: } \Rightarrow \begin{pmatrix} \frac{\partial f \circ g}{\partial x_1}(x) \\ \vdots \\ \frac{\partial f \circ g}{\partial x_d}(x) \end{pmatrix} = J \star v = \text{ by definition } J \star \nabla f(g(x))$$

chain rule as matrix multiplication

$$
\begin{aligned}
f : \mathbb{R}^m \to \mathbb{R},\ f(z_1, \dots, z_m) \in \mathbb{R} \\
g : \mathbb{R}^d \to \mathbb{R}^m,\ g(x_1, \dots, x_d) \in \mathbb{R}^m \\
\frac{\partial f \circ g}{\partial x_k}(x) = \sum_{r=1}^{m} \frac{\partial f}{\partial z_r}(g(x)) \frac{\partial g_r}{\partial x_k}(x) = \sum_{r=1}^{m} J_{kr} v_r = (J \star v)_k \\
\Rightarrow \begin{pmatrix} \frac{\partial f \circ g}{\partial x_1}(x) \\ \vdots \\ \frac{\partial f \circ g}{\partial x_d}(x) \end{pmatrix} = J \star \nabla f(g(x))
\end{aligned}
$$

- by definition:

$$
\begin{pmatrix} \frac{\partial f \circ g}{\partial x_1}(x) \\ \vdots \\ \frac{\partial f \circ g}{\partial x_d}(x) \end{pmatrix} = \nabla (f \circ g)(x)
$$

- what is J?

$$f : \mathbb{R}^m \to \mathbb{R},\ f(z_1,\dots,z_m) \in \mathbb{R}$$
$$g : \mathbb{R}^d \to \mathbb{R}^m,\ g(x_1,\dots,x_d) \in \mathbb{R}^m$$
$$\frac{\partial f \circ g}{\partial x_k}(x) = \sum_{r=1}^{m} \frac{\partial f}{\partial z_r}(g(x)) \frac{\partial g_r}{\partial x_k}(x) = \sum_{r=1}^{m} J_{kr} v_r = (J \star v)_k$$
$$\Rightarrow \begin{pmatrix} \frac{\partial f \circ g}{\partial x_1}(x) \\ \vdots \\ \frac{\partial f \circ g}{\partial x_d}(x) \end{pmatrix} = \nabla(f \circ g)(x) = J \star \nabla f(g(x))$$

- what is J? $J_{kr} \overset{\text{by def}}{=} \frac{\partial g_r}{\partial x_k}(x)$.

$$\Rightarrow J = J^{(g)} = \begin{pmatrix} \frac{\partial g_1}{\partial x_1}(x), \dots, \frac{\partial g_m}{\partial x_1}(x) \\ \frac{\partial g_1}{\partial x_2}(x), \dots, \frac{\partial g_m}{\partial x_2}(x) \\ \vdots \\ \frac{\partial g_1}{\partial x_d}(x), \dots, \frac{\partial g_m}{\partial x_d}(x) \end{pmatrix} = \underbrace{(\nabla g_1(x), \dots, \nabla g_m(x))}_{\text{Jacobi-matrix (or its transpose)}}$$

Gradient vs Jacobi-Matrix

- function f maps onto real numbers ($f(x) \in \mathbb{R}$)
  $\tilde{\leftrightarrow}$ apply ∇, Derivative: gradient

$$f(x_1, \ldots, x_d) \in \mathbb{R} \Rightarrow \nabla f(x) = \begin{pmatrix} \frac{\partial f}{\partial x_1}(x) \\ \vdots \\ \frac{\partial f}{\partial x_d}(x) \end{pmatrix} \text{ is the gradient}$$

- function $g = (g_1, \ldots, g_m)$ maps onto vectors ($g(x) \in \mathbb{R}^m$)
  $\tilde{\leftrightarrow}$ apply ∇ to each component g_k, Derivative: Jacobi-matrix.

Jacobi-Matrix $J^{(g)}$

is just a name for the matrix $(\nabla g_1(x), \ldots, \nabla g_m(x))$ of concatenated gradients for each output component g_i of $g = (g_1, \ldots, g_m)$ (applied ∇ to each output component)
The definition is transposed sometimes!

Next step: How can the chain rule be efficiently implemented using linear algebra?

$$
\begin{aligned}
f : \mathbb{R}^m \to \mathbb{R},\ f(z_1, \dots, z_m) \in \mathbb{R} \\
g : \mathbb{R}^d \to \mathbb{R}^m,\ g(x_1, \dots, x_d) \in \mathbb{R}^m \\
\frac{\partial f \circ g}{\partial x_k}(x) = \sum_{r=1}^{m} \frac{\partial f}{\partial z_r}(g(x)) \frac{\partial g_r}{\partial x_k}(x)
\end{aligned}
$$

$$
\Rightarrow \begin{pmatrix} \frac{\partial f \circ g}{\partial x_1}(x) \\ \vdots \\ \frac{\partial f \circ g}{\partial x_d}(x) \end{pmatrix} = \Big(\nabla g_1(x), \dots, \nabla g_m(x)\Big) \star \begin{pmatrix} \frac{\partial f}{\partial x_1}(g(x)) \\ \vdots \\ \frac{\partial f}{\partial x_d}(g(x)) \end{pmatrix}
$$

$$
\Rightarrow \nabla(f \circ g)(x) = \Big(\nabla g_1(x), \dots, \nabla g_m(x)\Big) \star \nabla f(g(x))
$$

the result (I): chain rule in matrix form

chain rule n-dim case as matrix multiplications

assume:

$$f : \mathbb{R}^m \mapsto \mathbb{R}, f(x) \in \mathbb{R}$$

and if ∇f is a column vector and if g is a row vector

$$g : \mathbb{R}^d \mapsto \mathbb{R}^m, g(x) = (g_1, \ldots, g_m) \in \mathbb{R}^m$$

then: $\nabla(f \circ g)(x) = (\nabla g_1(x), \nabla g_2(x), \ldots, \nabla g_m(x)) \star \nabla f(g(x))$

$= \text{(inner function } g \text{ gradients)} \star \text{(outer function } f \text{ gradients)}$

Mind here that the shapes must be correctly in this form – gradients are assumed to be column vectors: $g.shape = (1, m)$. $(\nabla g_i).shape = (d, 1)$ is a column vector.

If instead we would have $g.shape = (m, 1)$, $(\nabla f).shape = (1, m)$, and gradients are row vectors instead: $(\nabla g_i).shape = (1, d)$, then J and ∇f are transposed to the above (!), then one has to use ... see next slides if gradients are row vectors instead:

If instead we would have $g.shape = (m, 1)$, $(\nabla f).shape = (1, m)$, and gradients are row vectors instead: $(\nabla g_i).shape = (1, d)$, then J and ∇f are transposed to the above (!), then one has to use

$$\nabla(f \circ g)(x) = \nabla f(g(x)) \star \begin{pmatrix} \nabla g_1(x) \\ \nabla g_2(x) \\ \vdots \\ \nabla g_m(x) \end{pmatrix} = \nabla f(g(x)) \star J^{(g)}$$

How to remember that? consider the shapes:

$$f : \mathbb{R}^m \mapsto \mathbb{R}, f(x) \in \mathbb{R},\ \nabla f(x) \in \mathbb{R}^m$$
$$g : \mathbb{R}^d \mapsto \mathbb{R}^m, g(x) \in \mathbb{R}^m,\ \nabla g(x) = J^{(g)} \in \mathbb{R}^{m*d}$$

if gradients are columns: $\nabla f \circ g \sim (d, 1)$, $\nabla f \sim (m, 1)$

$$\begin{aligned} (d,1) &\stackrel{\sim}{=} (d,m) \star (m,1) \text{ only one way for } J \\ \nabla(f \circ g)(x) &= (\nabla g_1(x), \nabla g_2(x), \ldots, \nabla g_m(x)) \star \nabla f(g(x)) \end{aligned}$$

if gradients are rows: $\nabla f \circ g \sim (1, d)$, $\nabla f \sim (1, m)$

$$\begin{aligned} (1,d) &\stackrel{\sim}{=} (1,m) \star (m,d) \text{ transpose of the above} \\ \nabla(f \circ g)(x) &= \nabla f(g(x)) \star \begin{pmatrix} \nabla g_1(x) \\ \nabla g_2(x) \\ \vdots \\ \nabla g_m(x) \end{pmatrix} \end{aligned}$$

if gradients are row vectors instead:

chain rule n-dim case as matrix multiplications (transposed version)

$$f : \mathbb{R}^m \mapsto \mathbb{R}, f(x) \in \mathbb{R}$$

$$\text{if } g : \mathbb{R}^d \mapsto \mathbb{R}^m, g(x) = \begin{pmatrix} g_1 \\ \vdots \\ g_m \end{pmatrix} \in \mathbb{R}^m \text{ is a column vector and if}$$

$$\nabla f = \text{ is a row vector } \left(\frac{\partial f}{\partial x_1}, \ldots, \frac{\partial f}{\partial x_m} \right)$$

$$\text{then: } \Rightarrow \nabla (f \circ g)(x) = \nabla f(g(x)) \star J = \nabla f(g(x)) \star \begin{pmatrix} \nabla g_1(x) \\ \nabla g_2(x) \\ \vdots \\ \nabla g_m(x) \end{pmatrix}$$

$$= (\text{outer function } f \text{ gradients}) \star (\text{inner function } g \text{ gradients})$$

Now: Chain rule for more than two functions using linear algebra (without any neural networks)?

assume: gradients are column vectors here. Then:

$$\nabla(f \circ g)(x) = (\nabla g_1(x), \nabla g_2(x), \ldots, \nabla g_m(x)) \star \nabla f(g(x))$$
$$= (\text{inner function } g \text{ gradients}) \star (\text{outer function } f \text{ gradients})$$

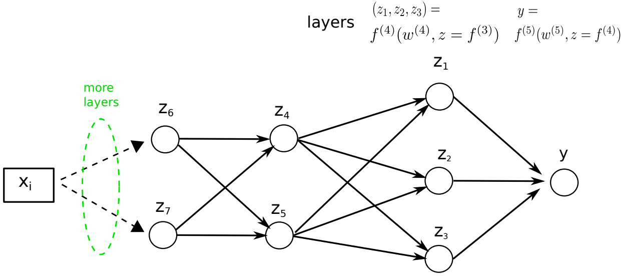
This chains to more than two functions:

$$\nabla(f \circ g \circ t)(x) = \nabla t(x) \star \nabla(f \circ g)(t(x)) =$$
$$= (\nabla t_1(x), , \ldots, \nabla t_n(x)) \star (\nabla g_1(t(x)), \ldots, \nabla g_m(t(x))) \star \nabla f(g(t(x)))$$
$$= (\text{inner gradients}) \star (\text{mid gradients}) \star (\text{outer gradients})$$
$$= (\text{inner gradients})_{|x} \star (\text{mid gradients})_{|t(x)} \star (\text{outer gradients})_{|g(t(x))}$$

Backpropagation computes gradients via chainrule along the (directed) edges in the neural network graph.
Backprop = chainrule on a DAG

What is backprop?

Executing chainrule along a directed graph.

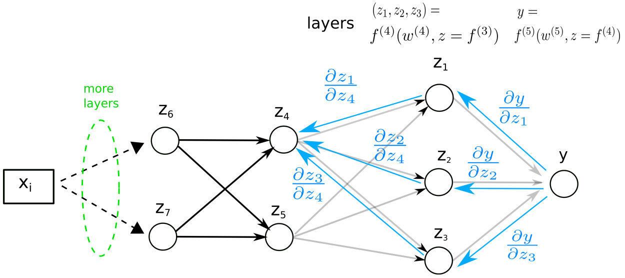


$$\frac{\partial y}{\partial z_1} = \text{ ? ... compute directly}$$

$$y = \sum_i w_i z_i + b$$

$$\frac{\partial y}{\partial z_i} = w_i \text{ for } i = 1, 2, 3$$

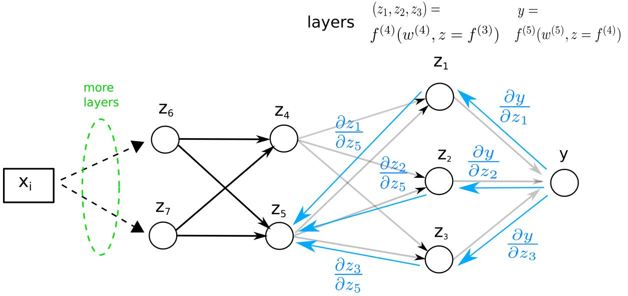
Executing chainrule along a directed graph.



$\frac{\partial y}{\partial z_4} =???$

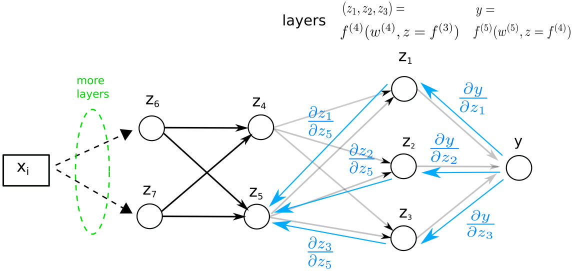
$$\frac{\partial y}{\partial z_4} = \frac{\partial y}{\partial z_1}\frac{\partial z_1}{\partial z_4} + \frac{\partial y}{\partial z_2}\frac{\partial z_2}{\partial z_4} + \frac{\partial y}{\partial z_3}\frac{\partial z_3}{\partial z_4}$$

Executing chainrule along a directed graph.



$\frac{\partial y}{\partial z_5} = ???$ $\quad \frac{\partial y}{\partial z_5} = \frac{\partial y}{\partial z_1}\frac{\partial z_1}{\partial z_5} + \frac{\partial y}{\partial z_2}\frac{\partial z_2}{\partial z_5} + \frac{\partial y}{\partial z_3}\frac{\partial z_3}{\partial z_5}$

- Note the flow: each edge $z_k \to z_i$ has a partial derivative $\frac{\partial z_i}{\partial z_k}$ flowing backwards
- each path (e.g.$z_4 \to y$) is the product of its associated edge terms $\frac{\partial z_i}{\partial z_k}$.



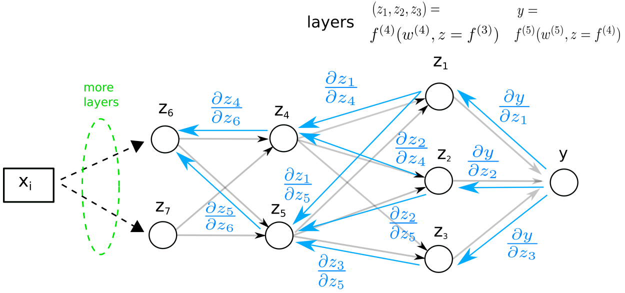
Learning goals

- Note the flow: each edge $z_k \to z_i$ has a partial derivative $\frac{\partial z_i}{\partial z_k}$ flowing backwards
- each path (e.g.$z_4 \to y$) is the product of its associated edge terms $\frac{\partial z_i}{\partial z_k}$
- In particular it is the product of the last edge with the product at the last node :)

Now you can compute backpropagation on your own on any DAG graph

What is backprop?

Executing chainrule along a directed graph.



$\frac{\partial y}{\partial z_6}$ =??? The flow principle continues: $\frac{\partial y}{\partial z_6} = \frac{\partial y}{\partial z_4}\frac{\partial z_4}{\partial z_6} + \frac{\partial y}{\partial z_5}\frac{\partial z_5}{\partial z_6}$

- how to use it: walk backwards and compute layer by layer
- at first $\frac{\partial y}{\partial z_1}, \frac{\partial y}{\partial z_2}, \frac{\partial y}{\partial z_3}$
- then $\frac{\partial y}{\partial z_4}, \frac{\partial y}{\partial z_5}$ from the previous $\frac{\partial y}{\partial z_i}$
- then $\frac{\partial y}{\partial z_6}, \frac{\partial y}{\partial z_7}$ from the previous $\frac{\partial y}{\partial z_i}$

Backpropagation first version (ignores layer structure)

1. Start at the top by finding the set of neurons z_l such that the loss function directly depends on their inputs $L = L(z_l)$

(!) the graph structure of the neural net tells you which neurons z_k give input to z_l

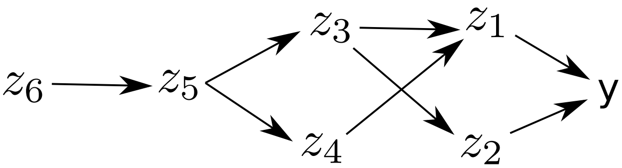
2. Then use for walking downwards (against directions of the forward computation flow):

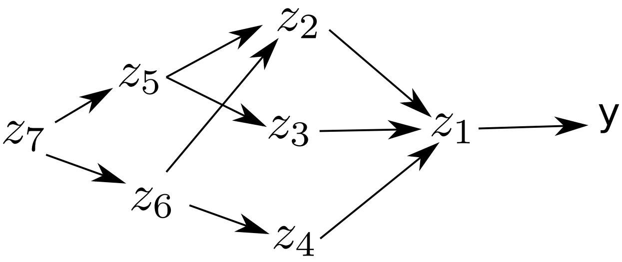
$$\frac{dL}{dz_k} = \sum_{l\text{: s.t. } k \text{ gives input to } l} \frac{dE}{dz_l} \frac{\partial z_l}{\partial z_k}$$

3. next: repeat step 2. for the z_k until you reach the bottom / or until you have covered all paths backwards from E to your weight of interest w_k

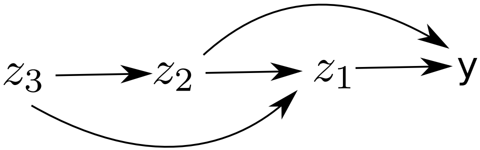
5 . Finish at the bottom by $\frac{dL}{dw_k} = \frac{dL}{dz_k} \frac{\partial z_k}{\partial w_k}$

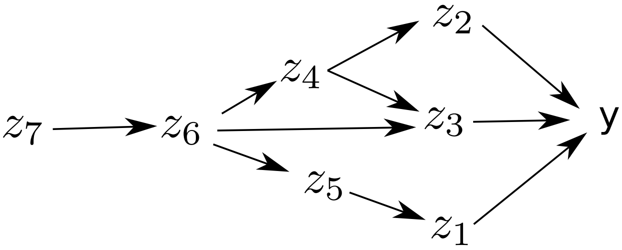
Exercise example for backprop




z_2
z_5
z_7
z_3
z_1
y
z_6
z_4

Exercise example for backprop





in practice walking down is done layer by layer

Backpropagation

1. Start at the top by finding the set of neurons z_l such that the loss function directly depends on their inputs $L = L(z_l)$. Let the last layer index be M.
2. for $i \in range(M-1, -1, step=-1)$

$$\forall z_k \in \text{Layer}_i : \frac{dL}{dz_k} = \sum_{l\text{: s.t. } k \text{ gives input to } l} \frac{dE}{dz_l} \frac{\partial z_l}{\partial z_k}$$

3. next: repeat step 2. for the z_k until you reach the bottom / or until you have covered all paths backwards from E to your weight of interest w_k

4 Finish at the bottom by $\frac{dL}{dw_k} = \frac{dL}{dz_k} \frac{\partial z_k}{\partial w_k}$

Efficient Backprop via matrix multiplications

Important for implementation: backprop is implemented efficiently by writing the chain rules as matrix multiplications in matrix algebra.

- Remember: $f^{(l)} = f^{(l)}(w^{(l)}, f^{(l-1)})$, $f^{(l-1)} = f^{(l-1)}(w^{(l-1)}, f^{(l-2)})$
- assume we have already computed the gradient with respect to inputs to layer l: $\nabla^{f^{(l-1)}}(L \cdots \circ f^{(l)})$.
- want: $\nabla^{f^{(l-2)}}(L \cdots \circ f^{(l)} \circ f^{(l-1)})$
  By slide 26:

$$\nabla^{f^{(l-2)}}(L \cdots \circ f^{(l)} \circ f^{(l-1)}) = \nabla^{f^{(l-2)}} f^{(l-1)} \star \nabla^{f^{(l-1)}}(L \cdots \circ f^{(l)})$$

- have now: $\nabla^{f^{(l-2)}}(L \cdots \circ f^{(l-1)})$
- iterate further down (+ reusing)

$$\nabla^{f^{(l-3)}}(L \cdots \circ f^{(l-1)} \circ f^{(l-2)}) = \nabla^{f^{(l-3)}} f^{(l-2)} \star \nabla^{f^{(l-2)}}(L \cdots \circ f^{(l-1)})$$

Efficient Backprop via matrix multiplications

- Remember for the network $f^{(l)} = f^{(l)}(w^{(l)}, f^{(l-1)})$
- One thing to finish: we need gradients with respect to the trainable parameters $\nabla^{w^{(l-1)}}(L \cdots \circ f^{(l)} \circ f^{(l-1)})$

$$f^{(l)} = f^{(l)}(w^{(l)}, f^{(l-1)})$$

$$\text{and so } f^{(l-1)} = f^{(l-1)}(w^{(l-1)}, f^{(l-2)})$$

$$\nabla^{f^{(l-2)}}(L \cdots \circ f^{(l)} \circ f^{(l-1)}) = \nabla^{f^{(l-2)}} f^{(l-1)} \star \nabla^{f^{(l-1)}}(L \cdots \circ f^{(l)})$$

$$\nabla^{w^{(l-1)}}(L \cdots \circ f^{(l)} \circ f^{(l-1)}) = \nabla^{w^{(l-1)}} f^{(l-1)} \star \nabla^{f^{(l-1)}}(L \cdots \circ f^{(l)})$$

Backpropagation

The last two equations are the iterative version of backprop in matrix algebra ... BUT: mind the shapes (gradients are column or row vectors)

Why is this efficient ?

Exemplary three layer network:

$$z^{(l)} = f^{(l)}(w^{(l)}, z^{(l-1)})$$
$$y = f^{(3)}(w^{(3)}, f^{(2)}(w^{(2)}, f^{(1)}(w^{(1)}, x)))$$

goal: to compute $\nabla^{w_d} y$ – using chain rule, $\nabla^{(z)} y$ denotes which vector z of variables the gradient of y is computed for.

$$\nabla^{w^{(3)}} y \rightarrow \text{ compute directly}$$
$$\nabla^{w^{(2)}} y = \nabla^{w^{(2)}}(f^{(3)} \circ f^{(2)}) = \nabla^{w^{(2)}} f^{(2)} \star \nabla^{f^{(2)}} f^{(3)}$$
$$= \nabla^{w^{(2)}} f^{(2)} \star \nabla^{f^{(2)}} f^{(3)}(w^{(3)}, f^{(2)}(\ldots))$$
$$\nabla^{w^{(1)}} y = \nabla^{w^{(1)}}(f^{(3)} \circ f^{(2)} \circ f^{(1)}) = \nabla^{w^{(1)}} f^{(1)} \star \nabla^{f^{(1)}}(f^{(3)} \circ f^{(2)})$$
$$= \nabla^{w^{(1)}} f^{(1)} \star \nabla^{f^{(1)}} f^{(2)} \star \nabla^{f^{(2)}} f^{(3)}(w^{(3)}, f^{(2)}(\ldots))$$

Observation

We need to have the Jacobi-matrix $\nabla^{f^{(k-1)}} f^{(k)}(w^{(k)}, f^{(k-1)}(\ldots))$ for computing the gradients for all parameters $w^{(k-1)}, w^{(k-2)}, \ldots, w^{(1)}$ in layers closer to the input. Backprop: compute it once, reuse it for all layers.

Single layer gradients

Piece of cake

$$f^{(l-1)} = g(f^{(l-2)} \star w^{(l-1)} + b^{(l-1)}),\ g(\cdot) \text{ activation}$$

Note $f^{(l-1)}$ is a $(1, m)$-shaped vector, and $w^{(l-1)}$ is a matrix (n, m) , and $f^{(l-2)}$ is a $(1, n)$-shaped vector.

Consider a single output neuron $f^{(l-1)}[0, k]$ of the layer $f^{(l-1)}$:

$$\text{then: } f^{(l-1)}[0, k] = g(f^{(l-2)} \star w^{(l-1)}[:, k] + b^{(l-1)}[0, k])$$

$$\nabla^{f^{(l-2)}} f^{(l-1)}[0, k] = ?$$

$$\nabla^{w^{(l-1)}[:, {\color{red}k}]} f^{(l-1)}[0, {\color{red}k}] = ? \text{ the two red must match}$$

$$\Rightarrow \frac{\partial f^{(l-1)}[0, k]}{\partial f^{(l-2)}[0, d]} = g'(f^{(l-2)} \star w^{(l-1)}[:, k] + b^{(l-1)}[0, k]) w^{(l-1)}[d, k]$$

$$\Rightarrow \frac{\partial f^{(l-1)}[0, k]}{\partial w^{(l-1)}[d, k]} = g'(f^{(l-2)} \star w^{(l-1)}[:, k] + b^{(l-1)}[0, k]) f^{(l-2)}[0, d]$$

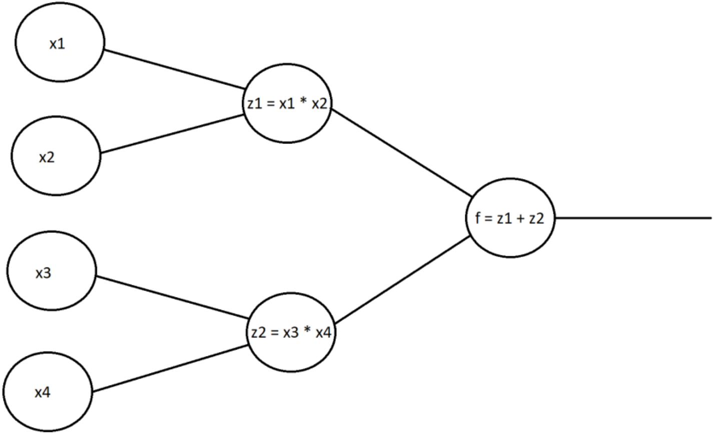
$$\text{and } \frac{\partial f^{(l-1)}[0, k]}{\partial b^{(l-1)}[0, k]} = g'(f^{(l-2)} \star w^{(l-1), [:, k]} + b^{(l-1)}[0, k])$$

Outline

A directed-graph representation of computations done.

What is a computational graph?

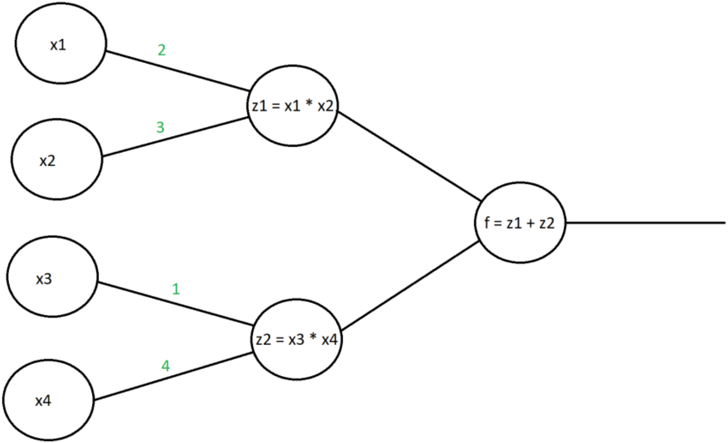
$f(\vec{x}) = x_1 * x_2 + x_3 * x_4 \qquad f(\vec{x}) = z_1 + z_2$



Forward pass: the actual computation

Forward propagation

$f(\vec{x}) = x_1 * x_2 + x_3 * x_4 \qquad f(\vec{x}) = z_1 + z_2$

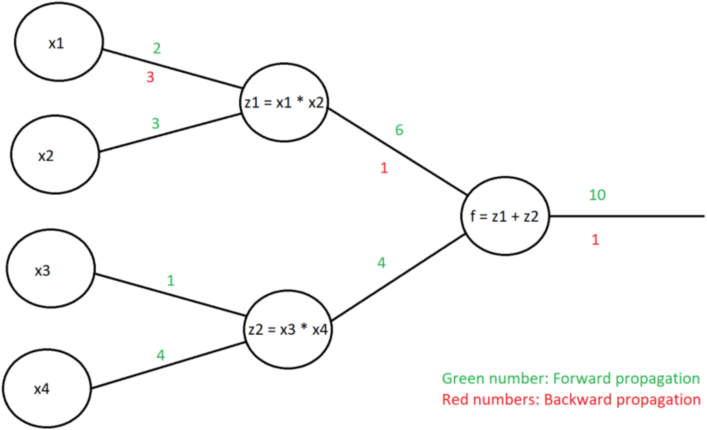


Backward pass: computing derivates

Backward propagation

What if we want to get the derivative of *f* with respect to the *x1*?

$$f(\vec{x}) = x_1 * x_2 + x_3 * x_4 \qquad f(\vec{x}) = z_1 + z_2 \qquad \frac{\partial f(\vec{x})}{\partial x_1} = \frac{\partial f}{\partial z_1}\frac{\partial z_1}{\partial x_1} = x_2$$



What ? Automatic differentiation with respect to variables used in computations.
You can define a sequence of computations, then call `.backward()` or `torch.autograd.grad(...)`. see `autograf2.py`, `print_computationalgraph.py`

When ?

- If tensors are leaf tensors and have the `requires_grad=True` flag set, then they are marked for tracking operations along the computation sequence for later gradient computation.
- leaf tensor: not created as the result of an operation but defined by you as an input.

https://pytorch.org/tutorials/beginner/blitz/autograd_tutorial.html#sphx-glr-beginner-blitz-autograd-tutorial-py

...

if `e` is a tensor with 1 element, then `e.backward()` computes the gradient of `e` with respect to all its inputs that were involved in computing `e`.

see `print_computationalgraph.py`: the whole backward graph if e is a tensor of $n \geq 2$ elements, then the gradient of e is a matrix, the Jacobi-matrix. Example for 3 elements:

$$e = (e_1, e_2, e_3)$$

$$de/dx = \begin{pmatrix} \frac{de_1}{dx_1} & \frac{de_2}{dx_1} & \frac{de_3}{dx_1} \\ \frac{de_1}{dx_2} & \frac{de_2}{dx_2} & \frac{de_3}{dx_2} \\ \vdots & \vdots & \vdots \\ \frac{de_1}{dx_8} & \frac{de_2}{dx_8} & \frac{de_3}{dx_8} \\ \vdots & \vdots & \vdots \\ \frac{de_1}{dx_D} & \frac{de_2}{dx_D} & \frac{de_3}{dx_D} \end{pmatrix}$$

if e is a tensor of $n \geq 2$ elements, then the gradient of e is a matrix, the Jacobi-matrix.
In this case: (for an example where e has 3 elements)
`e.backward(torch.tensor([-5,2,6]) )` computes the D-dim weighted gradient vector

$$\frac{de_1}{dx} * (-5) + \frac{de_2}{dx} * 2 + \frac{de_3}{dx} * 6$$
$$= \begin{pmatrix} \frac{de_1}{dx_1} * (-5) + \frac{de_2}{dx_1} * 2 + \frac{de_3}{dx_1} * 6 \\ \frac{de_1}{dx_2} * (-5) + \frac{de_2}{dx_2} * 2 + \frac{de_3}{dx_2} * 6 \\ \vdots \\ \frac{de_1}{dx_8} * (-5) + \frac{de_2}{dx_8} * 2 + \frac{de_3}{dx_8} * 6 \\ \vdots \\ \frac{de_1}{dx_D} * (-5) + \frac{de_2}{dx_D} * 2 + \frac{de_3}{dx_D} * 6 \end{pmatrix}$$

This is an inner product between the jacobi matrix and a vector that has as many elements as e in the forward pass.

Autograd

- Autograd tracks the graph of computations
- Tracked computations will be used to compute a gradient automatically
- use `with torch.no_grad():` environment to **not** record computations for gradient calculations for some larger block of code that is reused
- use case for `with torch.no_grad():`: everything outside of handling training data, e.g. computing validation or test predictions/ scores.[a]

[a] Why you dont want to track gradient computations in this case?

`with torch.no_grad():`

- use `with torch.no_grad():` environment to **not** record computations for gradient calculations for some larger block of code that is reused – use case: everything outside of handling training data, e.g. computing validation or test scores.[a]
- source of mistakes: every computation outside of a `with torch.no_grad():` environment will be used to compute gradients, and in the end to update parameters (e.g. predict on validation data).
- outlook: for GAN-training `sometensor.detach()` prevents the gradient flowing from `sometensor` to all those modules/variables used to compute sometensor.

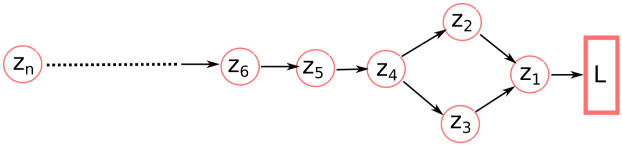
[a] Why you dont want to track gradient computations in this case?

Note: If you have a tensor with attached gradient, then the `.data` stores the tensor values, and `.grad.data` the gradient values

```
vals=x.data.numpy() #exports function values to numpy
g_vals=x.grad.data.numpy() #exports gradient values to numpy
```

Outline

Here I like to show that for classical neural networks the size of the gradient can become very small for layers close to the input.
Consider the following NN:



$$\frac{\partial L}{\partial z_4} = \frac{\partial L}{\partial z_1}\frac{\partial z_1}{\partial z_2}\frac{\partial z_2}{\partial z_4} + \frac{\partial L}{\partial z_1}\frac{\partial z_1}{\partial z_3}\frac{\partial z_3}{\partial z_4}$$

$$\frac{\partial L}{\partial z_5} = \frac{\partial L}{\partial z_1}\frac{\partial z_1}{\partial z_2}\frac{\partial z_2}{\partial z_4}\frac{\partial z_4}{\partial z_5} + \frac{\partial L}{\partial z_1}\frac{\partial z_1}{\partial z_3}\frac{\partial z_3}{\partial z_4}\frac{\partial z_4}{\partial z_5}$$

$$\frac{\partial L}{\partial z_6} = \frac{\partial L}{\partial z_1}\frac{\partial z_1}{\partial z_2}\frac{\partial z_2}{\partial z_4}\frac{\partial z_4}{\partial z_5}\frac{\partial z_5}{\partial z_6} + \frac{\partial L}{\partial z_1}\frac{\partial z_1}{\partial z_3}\frac{\partial z_3}{\partial z_4}\frac{\partial z_4}{\partial z_5}\frac{\partial z_5}{\partial z_6}$$

$$\frac{\partial L}{\partial z_n} = \frac{\partial L}{\partial z_1}\frac{\partial z_1}{\partial z_2}\frac{\partial z_2}{\partial z_4}\frac{\partial z_4}{\partial z_5}\frac{\partial z_5}{\partial z_6}\cdot\ldots\cdot\frac{\partial z_{n-1}}{\partial z_n} + \frac{\partial L}{\partial z_1}\frac{\partial z_1}{\partial z_3}\frac{\partial z_3}{\partial z_4}\frac{\partial z_4}{\partial z_5}\frac{\partial z_5}{\partial z_6}\cdot\ldots\cdot\frac{\partial z_{n-1}}{\partial z_n}$$

The partial derivatives are sums of chains of products. For neurons close to the input these chains are longer.

Now lets consider a classical neural network neuron with a sigmoid activation

$$z_1 = tanh(w_{12}z_2 + b)$$
$$\frac{\partial z_1}{\partial z_2} = tanh'(w_{12}z_2 + b)w_{12} = (1 - tanh^2(w_{12}z_2 + b))w_{12}$$

$(1 - tanh^2(w_{12}z_2 + b))$ is 1 at zero, otherwise quickly dropping to zero. Weights are usually initialized to be random values close to zero. Most of the time, such a derivative will be smaller than 1 in absolute value.

Multiplying long chains of values in $(-1, +1)$ quickly drops to zero:

$$0.5^4 = 0.0625,\ 0.5^{10} \approx 0.001,\ 0.5^{20} \approx 0.000001, 0.1^{10} \text{ etc}$$

In theory also gradient explosion may occur, if weights are set to large values.

The implication? Gradient updates far from the output can get very small.

one challenge in deep learning

In total this raises three problems:

- the gradients may become very small. Small gradients $\rightarrow$ small weight updates, slow learning
- the sizes of gradients will highly vary between neurons in a NN. So update speeds will vary across the network
- if used sigmoids as activations, a saturated sigmoid will result in a gradient close to zero, thus killing gradients along the whole chain downwards(see ReLU, leaky ReLU as alternatives)

One key challenge in deep learning is to maintain gradient flow so as to be able to update weights quickly, and at approximately the same speeds across the whole network

Outline

http://neuralnetworksanddeeplearning.com/chap3.html#weight_initialization
https://www.youtube.com/watch?v=6by6Xas_Kho

have to initialize neural network values so that gradient flows well at initialization. How to?

- symmetry breaking
- right scale of weights

Current standard for convolution layers in ReLU-type networks is Kaiming He et al. 2015 https://arxiv.org/pdf/1502.01852.pdf

Initialization for ReLU networks

- set biases to zero $b = 0$
- initialize weights as random values for symmetry breaking
- conv layers: draw weights from a normal with standard deviation equal to $\sigma = \sqrt{\frac{2}{n}}$

  $w_d \sim N(0, \sigma^2)$
- later: use transfer learning instead of training with random init

Initialization for pReLU networks

- set biases to zero $b = 0$
- initialize weights as random values for symmetry breaking
- conv layers: draw weights from a normal with standard deviation equal to $\sigma = \sqrt{\frac{2}{(1+a)^2 n}}$ where a is the negative slope
  $w_d \sim N(0, \sigma^2)$
- later: use transfer learning instead of training with random init

what is n for conv layers?

- either kernelsize(h)$*$ kernelsize(w) $*$ input_channels
- or kernelsize(h)$*$ kernelsize(w) $*$ output_channels

what is n for linear layers ?

- either input_channels
- or output_channels

if we use pReLu/ leaky ReLU with negative slope a:

$$g(z) = \mathbb{1}[z > 0] - a\mathbb{1}[z < 0]$$

conv layers: $\sigma = \sqrt{\frac{2}{(1+a^2)n}}$

$$w_d \sim N(0, \sigma^2)$$

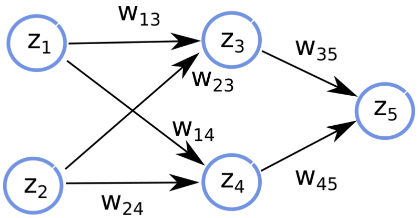
https://pytorch.org/docs/stable/nn.init.html

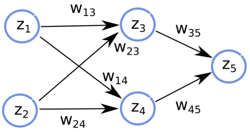
```
torch.nn.init.kaiming_normal_(tensor, a=0, mode='fan_in', nonlinearity='leaky_relu')
```

The weights of different neurons should be initialized with asymmetric values. Reason: allow different neurons to learn to become detectors for different structures.

Next: show that non-randomized initialization can result in symmetries. Then different neurons keep same weights – same function.

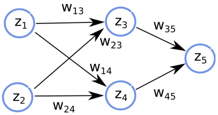
Consider a fully symmetrically initialized neural network:





- If $w_{13} = w_{14}$ and $w_{23} = w_{24}$, then the neuron activations of z_3 and z_4 are the same. If now also $w_{35} = w_{45}$, then we would have identically gradient updates for w_{13} versus w_{14}, as well as for w_{23} versus w_{24}.
- then: the weights of w_{13} versus w_{14} change in the same way, during learning it stays $w_{13} = w_{14}$ and z_3 vs z_4 nevers learns something different from z_1

Neural Net initialization: symmetry breaking



$$\frac{\partial L}{\partial w_{13}} = \frac{\partial L}{\partial z_5}\frac{\partial z_5}{\partial z_3}\frac{\partial z_3}{\partial w_{13}}, \frac{\partial L}{\partial w_{14}} = \frac{\partial L}{\partial z_5}\frac{\partial z_5}{\partial z_4}\frac{\partial z_4}{\partial w_{14}}$$

$$\frac{\partial z_5}{\partial z_3} = \sigma'(w_{35}z_3 + w_{45}z_4)w_{35}$$

$$\frac{\partial z_5}{\partial z_4} = \sigma'(w_{35}z_3 + w_{45}z_4)w_{34}$$

$$w_{35} = w_{34} \Rightarrow \frac{\partial z_5}{\partial z_3} = \frac{\partial z_5}{\partial z_4} \; !!$$

$$\frac{\partial z_3}{\partial w_{13}} = \sigma'(w_{13}z_1 + w_{23}z_2)z_1$$

$$\frac{\partial z_4}{\partial w_{14}} = \sigma'(w_{14}z_1 + w_{24}z_2)z_1$$

$$w_{13} = w_{14}, w_{23} = w_{24} \Rightarrow \frac{\partial z_3}{\partial w_{13}} = \frac{\partial z_4}{\partial w_{14}}$$

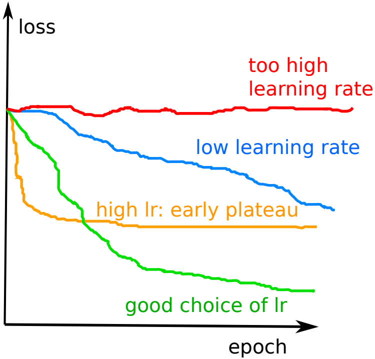
$$\Rightarrow \frac{\partial L}{\partial w_{13}} = \frac{\partial L}{\partial w_{14}}$$

- the right scale
- idea: initialize so that the average variance of outputs is constant in all the layers, and there is no value explosion during the forward pass
- similar argument holds for variance of gradients in the backward pass

see lec_initialization2.pdf for the derivation of the Kaiming-He Intializer

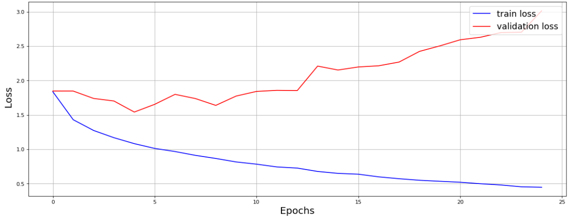
...

Outline



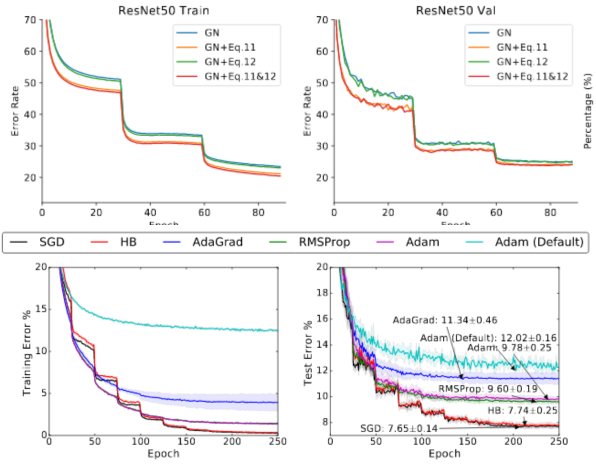
- too high lr: if non-convergence visible on the training loss already
- high lr: if have early plateau on train loss. learning rate decrease scheme can help!
- good choice of lr if visible on train loss. On val loss it can go up due to overfitting even when lr is set optimally

Typical results after training



- Gap between training error and validation error.
- Need regularization (or more data) to avoid overfitting.

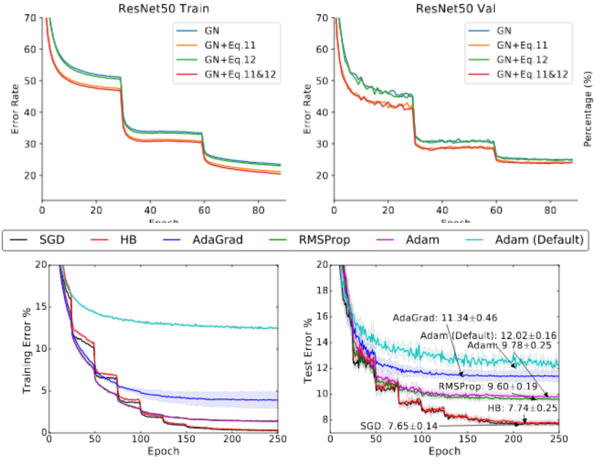
What is happening at the jumps?



(a) CIFAR-10 (Train) **(b)** CIFAR-10 (Test)

source: a paper from next lecture https://arxiv.org/abs/1903.10520 and https://arxiv.org/abs/1705.08292

The effects of learning rate decrease are drastic!



(a) CIFAR-10 (Train) **(b)** CIFAR-10 (Test)

source: a paper from next lecture https://arxiv.org/abs/1903.10520 and https://arxiv.org/abs/1705.08292

Outline

Recap: chain rule

- 1-dim case: $x \in \mathbb{R}$, $g(x) \in \mathbb{R}$

$$h(x) = f \circ g(x)$$
$$\frac{\partial h}{\partial x}(x) = \frac{\partial f \circ g}{\partial x}(x) = f'(g(x))g'(x) = \frac{\partial f}{\partial z}(g(z))\frac{\partial g}{\partial x}(x)$$

- n-dim case:
  - recap: derivatives tell you about directional derivatives $Dg(x)[h] = \nabla g(x)^T h$
  - recap: derivatives define linear mappings $L[\cdot] = Dg(x)[\cdot]$: directions h onto slopes $Dg(x)[h]$
  - derivative of chained functions $\leftrightarrow$ chaining of linear mappings

$$Df \circ g(x)[h] = Df(g(x))[Dg(x)[h]]$$

- meaning ?
  - compute the directional derivative $Dg(x)[h]$ of the inner mapping g in direction h at point x
  - plug it into the linear mapping $Df(g(x))[\cdot]$ for the directional derivative of the outer mapping f

chain rule n-dim case

for 2 functions f, g the chainrule of their concatenation $f \circ g(x)$ is given as the chaining of their linear mappings $Df(g(x))[\cdot]$ and $Dg(x)[\cdot]$ used to compute the directional derivatives for f (in point $g(x)$) and g (in point x):

$$\begin{aligned} f : \mathbb{R}^m \to \mathbb{R},\ & f(z_1, \ldots, z_m) && \in \mathbb{R} \\ g : \mathbb{R}^d \to \mathbb{R}^m,\ & g(x_1, \ldots, x_d) \\ & = (g_1(x_1, \ldots, x_d), \ldots, g_m(x_1, \ldots, x_d)) && \in \mathbb{R}^m \\ Df \circ g(x)[h] & = Df(g(x))[\,Dg(x)[h]\,] \end{aligned}$$

$$\frac{\partial f \circ g}{\partial x_k} = \sum_{r=1}^{m} \frac{\partial f}{\partial z_r}(g(x)) \frac{\partial g_r}{\partial x_k}(x)$$

The fact that it is a concatenation of two linear mappings $\leftrightarrow$ must correspond to matrix multiplication of two matrices, explains why you have to do that summing.

$$
\begin{aligned}
f : \mathbb{R}^m \to \mathbb{R},\ & f(z_1, \dots, z_m) && \in \mathbb{R} \\
g : \mathbb{R}^d \to \mathbb{R}^m,\ & g(x_1, \dots, x_d) \\
& = (g_1(x_1, \dots, x_d), \dots, g_m(x_1, \dots, x_d)) && \in \mathbb{R}^m \\
Df \circ g(x)[h] &= Df(g(x))[\, Dg(x)[h]\,]
\end{aligned}
$$

How does this translate into linear algebra?

$$
\begin{aligned}
Df(g(x))[u] = \sum_{r=1}^{m} \frac{\partial f}{\partial z_r}(g(x))\, u_r \qquad &= (u_1, \dots, u_m) \begin{pmatrix} \frac{\partial f}{\partial z_1}(x) \\ \vdots \\ \frac{\partial f}{\partial z_m}(x) \end{pmatrix} \\
= \begin{pmatrix} u_1 \\ \vdots \\ u_m \end{pmatrix}^T \begin{pmatrix} \frac{\partial f}{\partial z_1}(x) \\ \vdots \\ \frac{\partial f}{\partial z_m}(x) \end{pmatrix} \qquad &= u^T \nabla f(g(x))
\end{aligned}
$$

Towards chain rule via matrix multiplications

directional derivative via matrix multiplications

$$Df(g(x))[u] = \sum_{r=1}^{m} \frac{\partial f}{\partial z_r}(g(x))\, u_r \qquad = (u_1, \ldots, u_m) \begin{pmatrix} \frac{\partial f}{\partial z_1}(x) \\ \vdots \\ \frac{\partial f}{\partial z_m}(x) \end{pmatrix}$$

$$= \begin{pmatrix} u_1 \\ \vdots \\ u_m \end{pmatrix}^T \begin{pmatrix} \frac{\partial f}{\partial z_1}(x) \\ \vdots \\ \frac{\partial f}{\partial z_m}(x) \end{pmatrix} \qquad = u^T \nabla f(g(x))$$

- $g_i(x) = g_i(x_1, \ldots, x_d) \in \mathbb{R}$, ∇g_i is defined same as ∇f for f.

$Dg(x)[h]$ can be represented by what structure?

$$
\begin{aligned}
g(x) &= (g_1(x), \ldots g_m(x)) \\
Dg(x)[h] &= (Dg_1(x)[h], \ldots, Dg_m(x)[h]) \\
&= (h^T \nabla g_1(x), h^T \nabla g_2(x), \ldots, h^T \nabla g_m(x)) \\
&= \left((h_1, \ldots, h_d) \begin{pmatrix} \frac{\partial g_1}{\partial x_1} \\ \vdots \\ \frac{\partial g_1}{\partial x_d} \end{pmatrix}, (h_1, \ldots, h_d) \begin{pmatrix} \frac{\partial g_2}{\partial x_1} \\ \vdots \\ \frac{\partial g_2}{\partial x_d} \end{pmatrix}, \ldots, (h_1, \ldots, h_d) \begin{pmatrix} \frac{\partial g_m}{\partial x_1} \\ \vdots \\ \frac{\partial g_m}{\partial x_d} \end{pmatrix} \right) \\
&= (h_1, \ldots, h_d) \left(\begin{pmatrix} \frac{\partial g_1}{\partial x_1} \\ \vdots \\ \frac{\partial g_1}{\partial x_d} \end{pmatrix}, \begin{pmatrix} \frac{\partial g_2}{\partial x_1} \\ \vdots \\ \frac{\partial g_2}{\partial x_d} \end{pmatrix}, \ldots, \begin{pmatrix} \frac{\partial g_m}{\partial x_1} \\ \vdots \\ \frac{\partial g_m}{\partial x_d} \end{pmatrix} \right) \\
&= (h_1, \ldots, h_d)(\nabla g_1(x), \nabla g_2(x), \ldots, \nabla g_m(x)) \\
&= h^T (\nabla g_1(x), \nabla g_2(x), \ldots, \nabla g_m(x))
\end{aligned}
$$

All I have been showing:

$$\begin{aligned} g(x) &= (g_1(x), \ldots g_m(x)) \\ Dg(x)[h] &= (Dg_1(x)[h], \ldots, Dg_m(x)[h]) \\ &= (h^T \nabla g_1(x), h^T \nabla g_2(x), \ldots, h^T \nabla g_m(x)) \\ &= h^T (\nabla g_1(x), \nabla g_2(x), \ldots, \nabla g_m(x)) \end{aligned}$$

$Dg(x)[h]$ can be represented by what structure for $g(x) = (g_1(x), \ldots g_m(x)$?

See the analogy:

$$\begin{aligned} f(x) &= f(x) \\ g(x) &= (g_1(x), \ldots g_m(x)) \\ Df(x)[u] &= u^T \nabla f(x) \\ Dg(x)[h] &= h^T (\nabla g_1(x), \nabla g_2(x), \ldots, \nabla g_m(x)) \end{aligned}$$

This is the Jacobi-matrix. To remember it, simply remember it as $(\nabla g_1(x), \ldots, \nabla g_m(x))$ where every gradient is a column vector

$$(\nabla g_1(x), \ldots, \nabla g_m(x)) = \begin{pmatrix} \frac{\partial g_1}{\partial x_1}, & \frac{\partial g_2}{\partial x_1}, \ldots, & \frac{\partial g_m}{\partial x_1} \\ \frac{\partial g_1}{\partial x_2}, & \frac{\partial g_2}{\partial x_2}, \ldots, & \frac{\partial g_m}{\partial x_2} \\ & \vdots & \\ \frac{\partial g_1}{\partial x_d}, & \frac{\partial g_2}{\partial x_d}, \ldots, & \frac{\partial g_m}{\partial x_d} \end{pmatrix}$$

Now we can derive the final result

$$D(f \circ g)(x)[h] = Df(g(x))[\, Dg(x)[h]\,]$$

$$Df(g(x))[u] = u^T \nabla f(x) \text{ where } u \text{ is a column vector}$$

$$\text{and } u^T \text{ is a row vector}$$

$$Dg(x)[h] = h^T(\nabla g_1(x), \nabla g_2(x), \ldots, \nabla g_m(x)) \text{ is a row vector!!}$$

- This implies that you have to plug in $h^T(\nabla g_1(x), \nabla g_2(x), \ldots, \nabla g_m(x))$ as u^T and **not as u**! Therefore:

$$\Rightarrow D(f \circ g)(x)[h] = Df(g(x))[\, Dg(x)[h]\,]$$
$$= h^T(\nabla g_1(x), \nabla g_2(x), \ldots, \nabla g_m(x)) \nabla f(g(x))$$

chain rule n-dim case as matrix multiplications

$$f : \mathbb{R}^m \mapsto \mathbb{R}, f(x) \in \mathbb{R}$$

$$g : \mathbb{R}^d \mapsto \mathbb{R}^m, g(x) = (g_1, \ldots, g_m) \in \mathbb{R}^m$$

$$D(f \circ g)(x)[h] = h^T(\nabla g_1(x), \nabla g_2(x), \ldots, \nabla g_m(x))\nabla f(g(x))$$

$$= h^T(\text{inner function } g \text{ gradients})(\text{outer function } f \text{ gradients})$$

$$
\begin{aligned}
D(f \circ g)(x)[h] &= h^T (\nabla g_1(x), \nabla g_2(x), \ldots, \nabla g_m(x)) \nabla f(g(x)) \\
&= h^T (\text{inner function gradients})(\text{outer function gradients})
\end{aligned}
$$

This chains to more than two functions:

$$
\begin{aligned}
&D(f \circ g \circ t)(x)[h] = \\
&D(f \circ g)(t(x))[\,Dt(x)[h]\,] = \\
&= h^T (\nabla t_1(x), , \ldots, \nabla t_n(x))(\nabla g_1(t(x)), \ldots, \nabla g_m(t(x))) \nabla f(g(t(x))) \\
&= h^T (\text{inner f gradients})(\text{mid f gradients})(\text{outer f gradients}) \\
&= h^T (\text{inner f gradients})_{|x} (\text{mid f gradients})_{|t(x)} (\text{outer f gradients})_{|g(t(x))}
\end{aligned}
$$

Questions?!