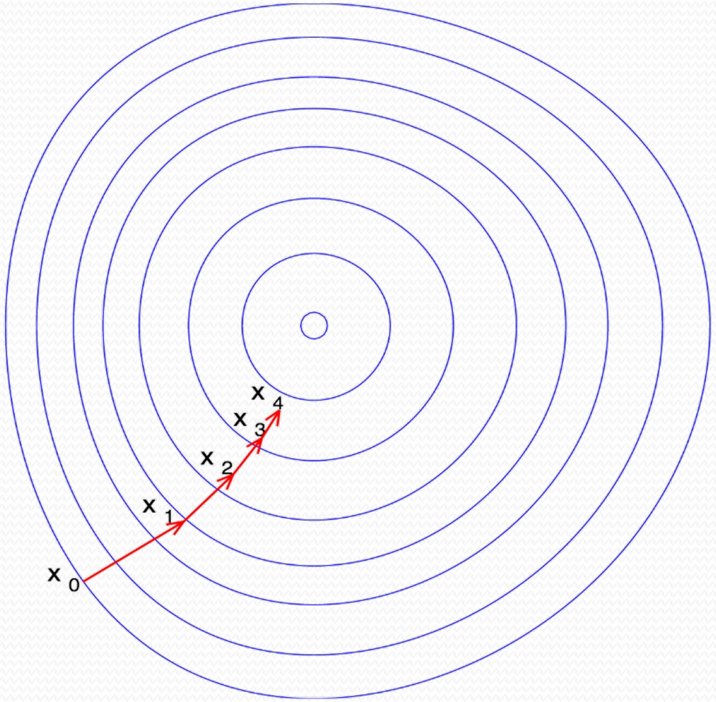
Applied Deep Learning

Training Neural Networks

Gradient Descent



$$\mathbf{x}_{n+1} = \mathbf{x}_n - \gamma_n \nabla F(\mathbf{x}_n)$$

Step size
or
Learning rate

Review: Chain Rule in One Dimension

- Suppose $f: \mathbb{R} \to \mathbb{R}$ and $g: \mathbb{R} \to \mathbb{R}$
- Define

$$h(x) = f(g(x))$$

- Then what is $h'(x) = dh/dx$?

$$h'(x) = f'\big(g(x)\big)g'(x)$$

Chain Rule in Multiple Dimensions

- Suppose $f: \mathbb{R}^m \to \mathbb{R}$ and $g: \mathbb{R}^n \to \mathbb{R}^m$, and $\mathbf{x} \in \mathbb{R}^n$
- Define

$$h(\mathbf{x}) = f(g_1(\mathbf{x}), \dots, g_m(\mathbf{x}))$$

- Then we can define partial derivatives using the multidimensional chain rule:

$$\frac{\partial f}{\partial x_i} = \sum_{l=1}^{m} \frac{\partial f}{\partial g_j} \frac{\partial g_j}{\partial x_i}$$

Automatic Differentiation (Differentiable Programming)

- Write an arbitrary program as consisting of basic operations f_1, …, f_n (e.g. +, -, *, cos, sin, …) that we know how to differentiate.
- Label the inputs of the program as $x_1, \ldots, x_n$, the intermediate results and parameters as $x_{n+1}, x_{n+2} \ldots$, and the final output x_N.
- Forward mode computation:

  For $i = n + 1, \ldots, N$

  $$x_i = f_i(\mathrm{x}_{\pi(i)})$$

  $\pi(i)$: Sequence of "parent" values

  (e.g. if $\pi(3) = (1,2)$, and f_3=+, then $x_3 = x_1 + x_2$)

- Backward mode automatic differentiation: apply the chain rule from the end of the program x_N back towards the beginning.

Simple Chain of Dependencies

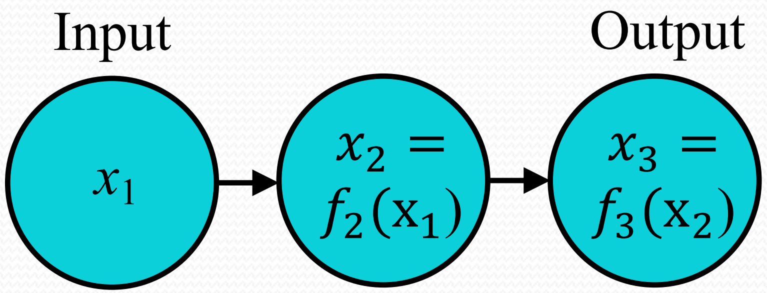
$\pi(i)$: Sequence of "parent" values
(e.g. if $\pi(3) = (1,2)$, and f_3=+, then $x_3=x_1+x_2$)

- Suppose $\pi(i) = i - 1$
- The (forward) computation:

  For $i = n + 1, \ldots, N$

  $$x_i = f_i(\mathrm{x}_{i-1})$$

For example:



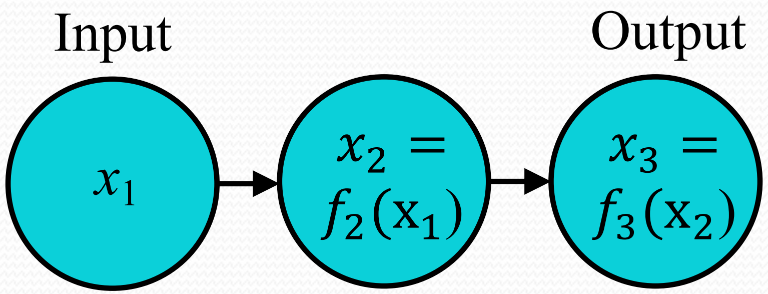
Simple Chain of Dependencies 2

- Suppose $\pi(i) = i - 1$
- The (forward) computation:

  For $i = n + 1, \ldots, N$

  $$x_i = f_i(\mathrm{x}_{i-1})$$

For example:



- What is $\frac{dx_N}{dx_i}$ in terms of $\frac{dx_N}{dx_{i+1}}$?

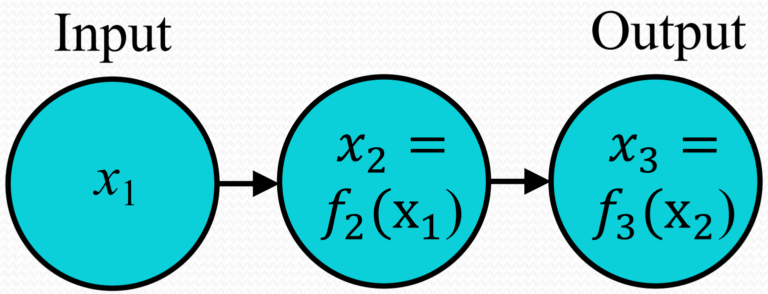
$$\frac{dx_N}{dx_i} = \frac{dx_N}{dx_{i+1}} \left(\frac{dx_{i+1}}{dx_i} \right)$$

Simple Chain of Dependencies 3

- Suppose $\pi(i) = i - 1$
- The computation:

$$\text{For } i = n+1, \ldots, N$$
$$x_i = f_i(\mathrm{x}_{i-1})$$

For example:

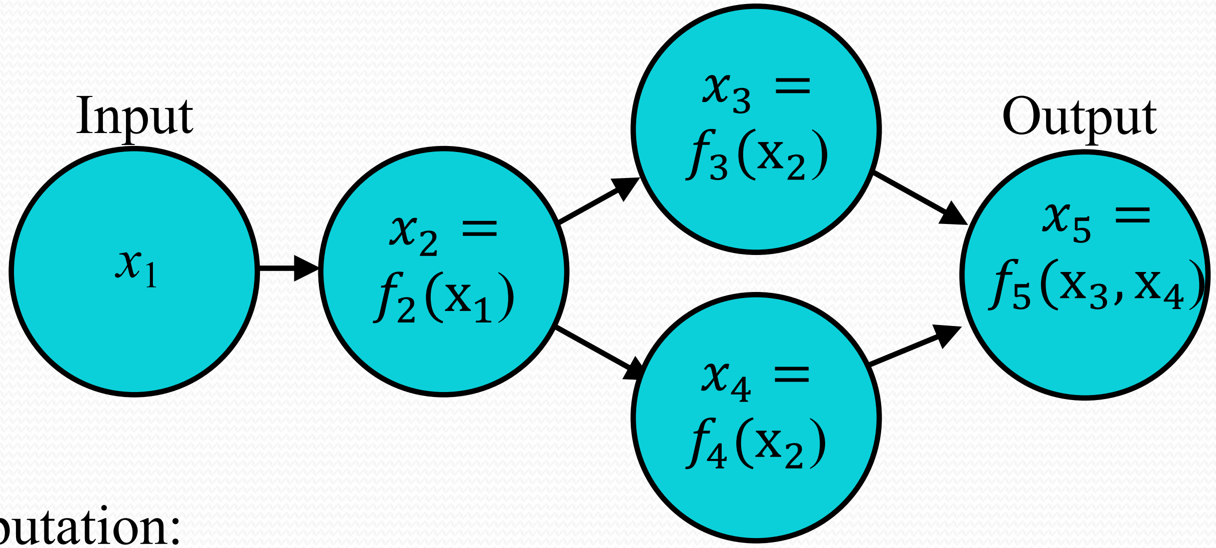


- What is $\frac{dx_N}{dx_i}$ in terms of $\frac{dx_N}{dx_{i+1}}$?

$$\frac{dx_N}{dx_i} = \frac{dx_N}{dx_{i+1}} \left(\frac{dx_{i+1}}{dx_i} \right)$$

- Conclusion: run the computation forwards. Then initialize $\frac{dx_N}{dx_N} = 1$ and work **backwards** through the computation to find $\frac{dx_N}{dx_i}$ for each i from $\frac{dx_N}{dx_{i+1}}$. This gives us the gradient of the output (x_N) with respect to every expression in our compute graph!

What if the Dependency Graph is More Complex?



- The computation:

  For $i = n + 1, \dots, N$

  $x_i = f_i(\mathrm{x}_{\pi(i)})$

  $\pi(i)$: Sequence of "parent" values
  (e.g. if $\pi(5) = (3,4)$, and f_5=+, then x_5=x_3+x_4)

- Solution: apply multi-dimensional chain rule.

Automatic Differentiation

- Computation:

$$x_i = f_i(\mathrm{x}_{\pi(i)})$$

- Multidimensional chain rule:

$$\frac{\partial f}{\partial x_i} = \sum_{j=1}^{m} \frac{\partial f}{\partial g_j} \frac{\partial g_j}{\partial x_i} \qquad f(g_1(\mathbf{x}), \dots, g_m(\mathbf{x}))$$

- Result:

$$\frac{\partial x_N}{\partial x_i} = \sum_{j: i \in \pi(j)} \frac{\partial x_N}{\partial x_j} \frac{\partial x_j}{\partial x_i}$$

Automatic Differentiation 2

- Algorithm: initialize:

$$\frac{dx_N}{dx_N} = 1$$

- For $i = N - 1, N - 2, \ldots, 1$, compute:

$$\frac{\partial x_N}{\partial x_i} = \sum_{j : i \in \pi(j)} \frac{\partial x_N}{\partial x_j} \frac{\partial x_j}{\partial x_i}$$

- Now we have differentiated the output of the program x_N with respect to the inputs $x_1, \ldots, x_n$, as well as the intermediate results and parameters.

Backpropagation Algorithm

- Apply **backward process of automatic differentiation** to a neural network's loss function.
- If we have one output neuron, squared error is:

$$E = \frac{1}{2}(t - y)^2$$

t is the target output for a training sample, and

y is the actual output of the output neuron.

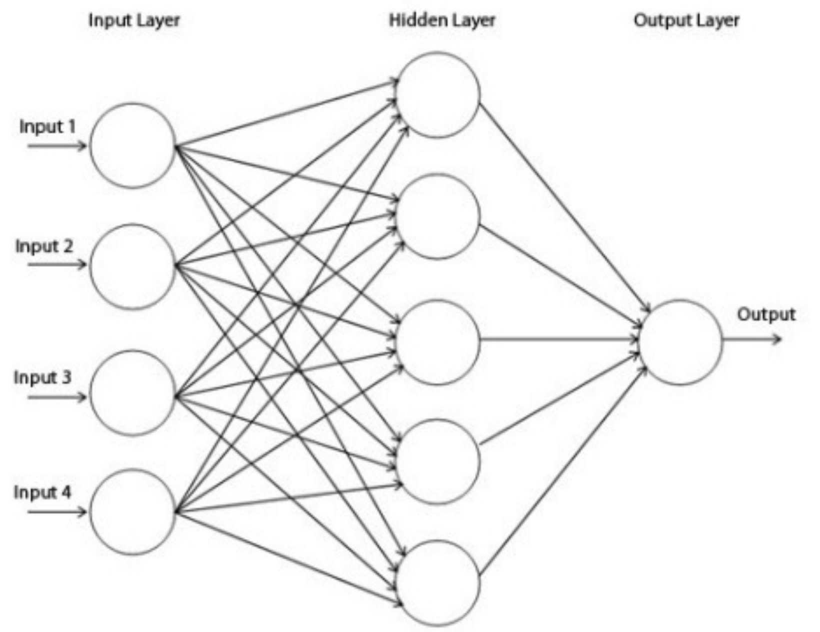
Backpropagation Algorithm 2

For each neuron j, its output o_j is defined as

$$o_j = \varphi(\mathrm{net}_j) = \varphi\left(\sum_{k=1}^{n} w_{kj} o_k\right).$$

The input net_j to a neuron is the weighted sum of outputs o_k of previous neurons. If

The variable w_{ij} denotes the weight between neurons i and j.



Backpropagation Algorithm 3

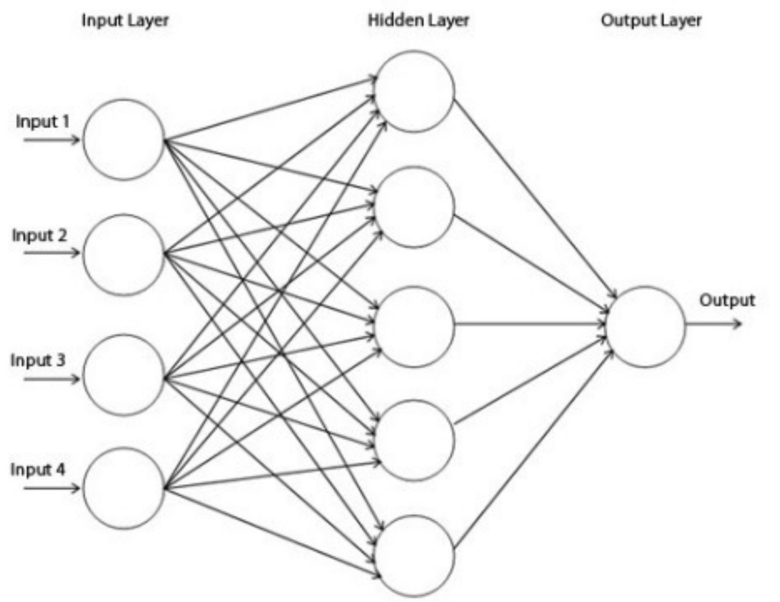
- Apply the chain rule twice:

$$\frac{\partial E}{\partial w_{ij}} = \frac{\partial E}{\partial o_j}\frac{\partial o_j}{\partial \mathrm{net_j}}\frac{\partial \mathrm{net_j}}{\partial w_{ij}}$$

- Last term :

$$o_j = \varphi(\mathrm{net}_j) = \varphi\left(\sum_{k=1}^{n} w_{kj} o_k\right).$$

$$\frac{\partial \mathrm{net_j}}{\partial w_{ij}} = \frac{\partial}{\partial w_{ij}}\left(\sum_{k=1}^{n} w_{kj} o_k\right) = o_i$$



Backpropagation Algorithm 4

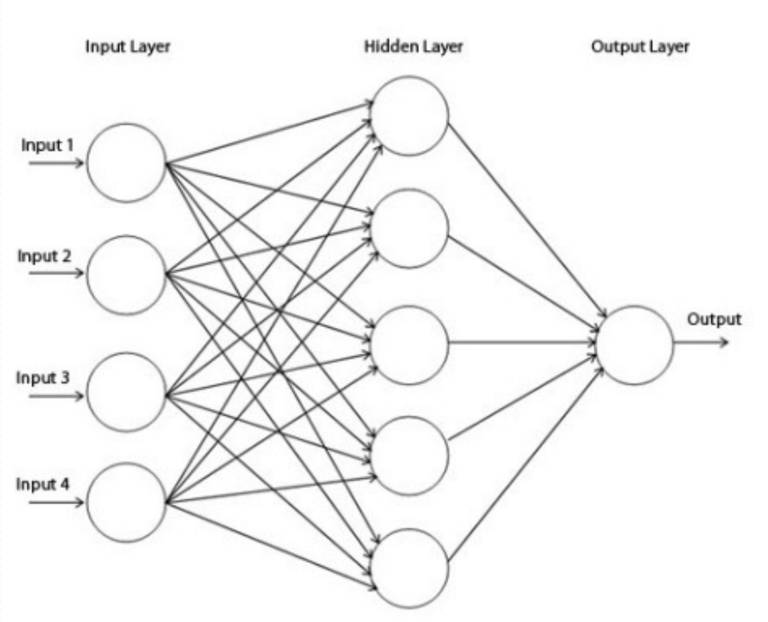
- Apply the chain rule twice:

$$\frac{\partial E}{\partial w_{ij}} = \frac{\partial E}{\partial o_j} \frac{\partial o_j}{\partial \mathrm{net_j}} \frac{\partial \mathrm{net_j}}{\partial w_{ij}}$$

- Second term :

$$o_j = \varphi(\mathrm{net}_j)$$

$$\frac{\partial o_j}{\partial net_j} = \varphi'(\mathrm{net}_j)$$



Backpropagation Algorithm 5

- Apply the chain rule twice:

$$E = \frac{1}{2}(t-y)^2$$

$$\frac{\partial E}{\partial w_{ij}} = \frac{\partial E}{\partial o_j}\frac{\partial o_j}{\partial \mathrm{net_j}}\frac{\partial \mathrm{net_j}}{\partial w_{ij}}$$

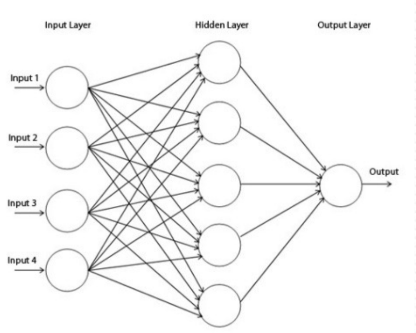
- If the neuron is in output layer, first term is easy:

$$o_j = y$$

$$\frac{\partial E}{\partial o_j} = \quad y\text{-}t$$

Backpropagation Algorithm 6

$$E = \frac{1}{2}(t - y)^2$$



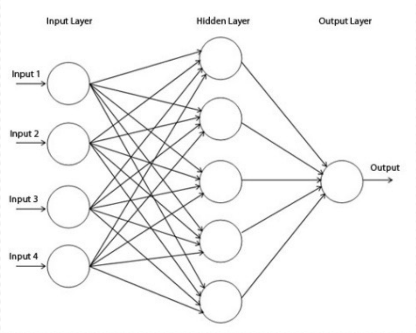
- Apply the chain rule twice:

$$\frac{\partial E}{\partial w_{ij}} = \frac{\partial E}{\partial o_j} \frac{\partial o_j}{\partial \mathrm{net_j}} \frac{\partial \mathrm{net_j}}{\partial w_{ij}}$$

- If the neuron is interior neuron, we use the chain rule from automatic differentiation.
- We need to know what expressions depend on the current neuron's output o_j? (the children of o_j in the computation graph)
- Answer: other neurons whose input sums, i.e. net_l for all neurons l, receiving inputs from the current neuron o_j.

Backpropagation Algorithm 7

$$E = \frac{1}{2}(t-y)^2$$



- Apply the chain rule twice:

$$\frac{\partial E}{\partial w_{ij}} = \frac{\partial E}{\partial o_j}\frac{\partial o_j}{\partial \text{net}_j}\frac{\partial \text{net}_j}{\partial w_{ij}}$$

- If the neuron is an interior neuron, chain rule:

$$\frac{\partial E}{\partial o_j} = \sum_{l\in L}\left(\frac{\partial E}{\partial \text{net}_l}\frac{\partial \text{net}_l}{\partial o_j}\right) = \sum_{l\in L}\left(\frac{\partial E}{\partial o_l}\frac{\partial o_l}{\partial \text{net}_l}w_{jl}\right)$$

All neurons receiving input from the current neuron j.

Backpropagation Algorithm 8

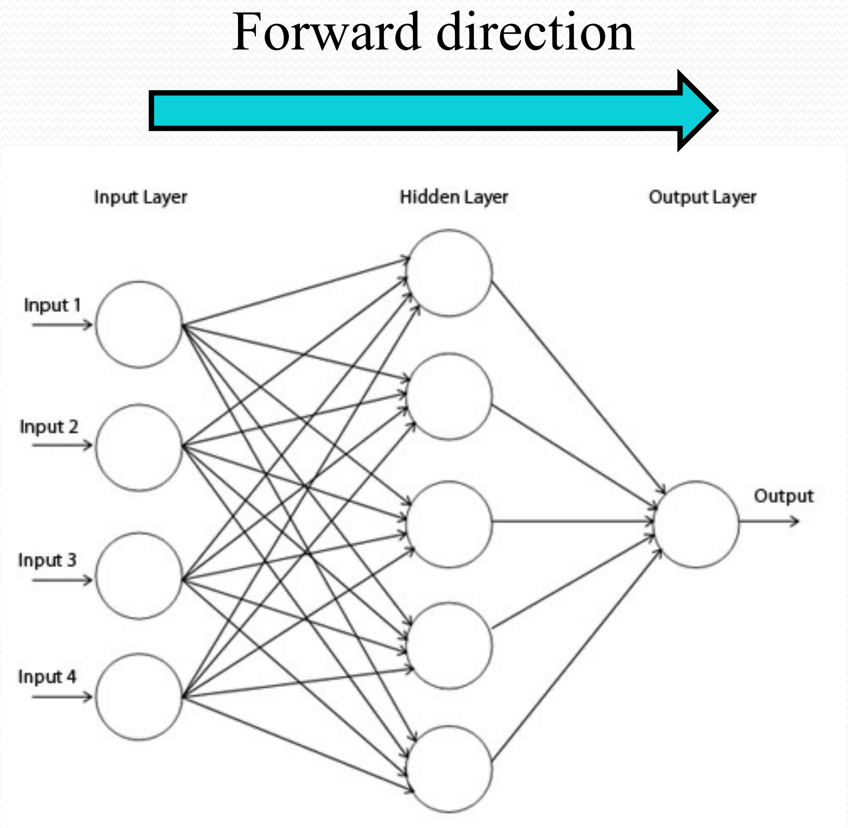
- Partial derivative of error E with respect to weight w_{ij}:

$$\frac{\partial E}{\partial w_{ij}} = \delta_j o_i$$

$$\delta_j = \frac{\partial E}{\partial o_j}\frac{\partial o_j}{\partial \mathrm{net}_j} = \varphi'(net_j)\times\begin{cases}(o_j - t_j) & \text{if } j \text{ is an output neuron}\\ \sum_{l\in L}\delta_l w_{jl} & \text{if } j \text{ is an interior neuron}\end{cases}$$

All neurons receiving input from the current neuron j.

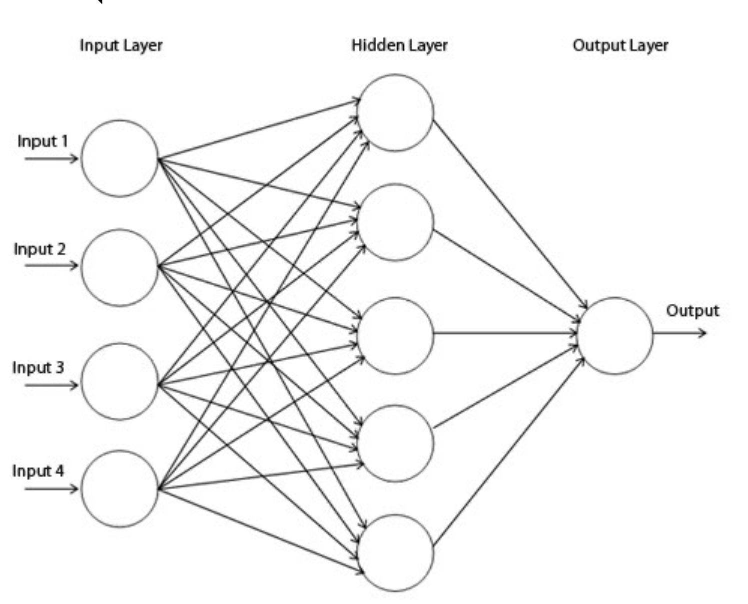
Backpropagation Algorithm 9



- Calculate network and error.

Backpropagation Algorithm (1960s-1980s)

Backward direction



- Backpropagate: from output to input, recursively compute

$$\frac{\partial E}{\partial w_{ij}} = \boldsymbol{\nabla}_w E$$

Backpropagation (auto differentiation)

- Differentiation on 100s million or more parameters. (Cannot be done manually. Mathematically simple but labor-wise challenging.)
- Numerical differentiation (not symbolic, i.e. analytical solution)
  - allows for functions such as relu, max, min, abs …
  - what about greater_than?

"Deep Learning est mort. Vive Differentiable Programming!"
"The important point is that people are now building a new kind of software by assembling networks of parameterized functional blocks and by training them from examples using some form of gradient-based optimization….It's really very much like a regular program, except it's parameterized, automatically differentiated, and trainable/optimizable." (Jan 5th, 2018)

-- Yann LeCun,
Director of Facebook AI Research and
the inventor of convolutional neural networks

Write Your Own PyTorch Operator

```
class Log1pExp(Function):
    @staticmethod
    def forward(ctx, x):
        # The forward pass can use ctx.
        e = my_exp(x)
        ctx.save_for_backward(e)
        output = my_log(1 + e)
        return output

    @staticmethod
    def backward(ctx, dy):
        e = ctx.saved_tensors
        if ctx.needs_input_grad[0]:
            dx = dy*(1 - 1 / (1 + e))
        else:
            dx = None
       return dx
```

To compute: `log(1 + exp(x))` and derivative:

```
x = torch.tensor(4.5)
y = Log1pExp.apply(x)

dydx = torch.autograd.grad(y, x)
```