Deep Learning 1

2025-2026 – Pascal Mettes

Lecture 2

AutoDiff

Where are we

Lecture	Title
1	Intro and history of deep learning
3	Deep learning optimization I
5	Convolutional deep learning
7	Graph deep learning
9	Multi-modal deep learning
11	What doesn't work in deep learning
13	Q&A

Lecture	Title
2	AutoDiff
4	Deep learning optimization II
6	Attention-based deep learning
8	From supervised to unsupervised deep learning
10	Generative deep learning
12	Non-Euclidean deep learning
14	Deep learning for videos

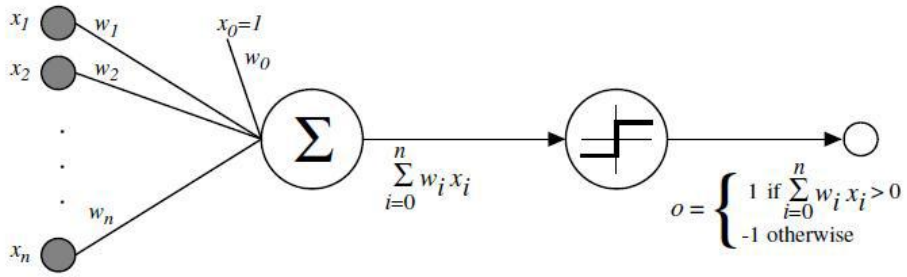
This lecture

Forward and backward propagation

AutoDiff

AutoGrad

The story so far



Then

Now

How do deep neural networks do it?

Deep learning in one slide

A family of parametric, non-linear and hierarchical representation learning functions, which are massively optimized with stochastic gradient descent to encode domain knowledge, i.e. domain invariances, stationarity.

$$a_L\left(x; \theta_{1,\ldots,\mathrm{L}}\right) = h_L\left(h_{L-1}(\ldots h_1(x, \theta_1), \theta_{L-1}), \theta_L\right)$$

x:input, θ_l: parameters for layer l, $a_l = h_l(x, \theta_l)$: (non-)linear function

Given training corpus $\{X, Y\}$ find optimal parameters

$$\theta^* \leftarrow \arg\min_\theta \sum_{(x,y)\subseteq(X,Y)} \ell\left(y, a_L\left(x; \theta_{1,\ldots,\mathrm{L}}\right)\right)$$

Deep feedforward networks

A composite of functions:

$$\mathrm{y} = f(x;\theta) = a_L\left(x;\theta_{1,\ldots,\mathrm{L}}\right) = h_L\ (h_{L-1}(\ldots(h_1(x,\theta_1),\ldots),\theta_{L-1}),\theta_L)$$

where θ_l denotes the parameters in the l-th layer.

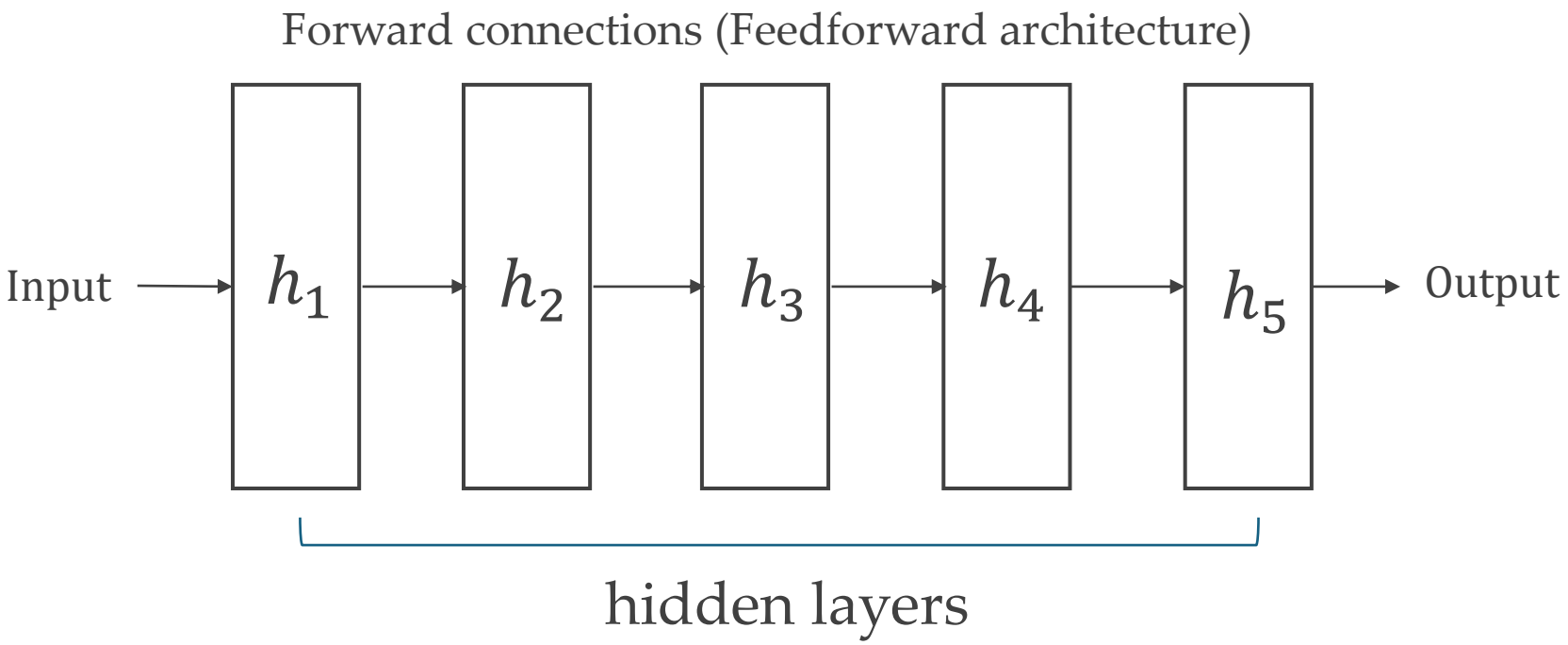
We can simplify the notation to

$$a_L = f(x;\theta) = h_L \circ h_{L-1} \circ \cdots \circ h_1 \circ x$$

where each functions h_l is parameterized by parameters θ_l.

Neural networks as blocks

With the last notation, we can visualize networks as blocks:



Neural network modules

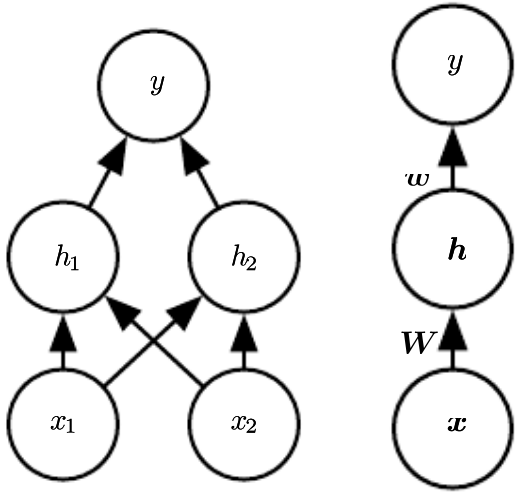
Module ⇔ Building block ⇔ Transformation ⇔ Function

A module receives as input either data x or another module's output

A module returns an output a based on its activation function $h(\dots)$

A module may or may not have trainable parameters w

Examples: f = Ax, f= exp(x)

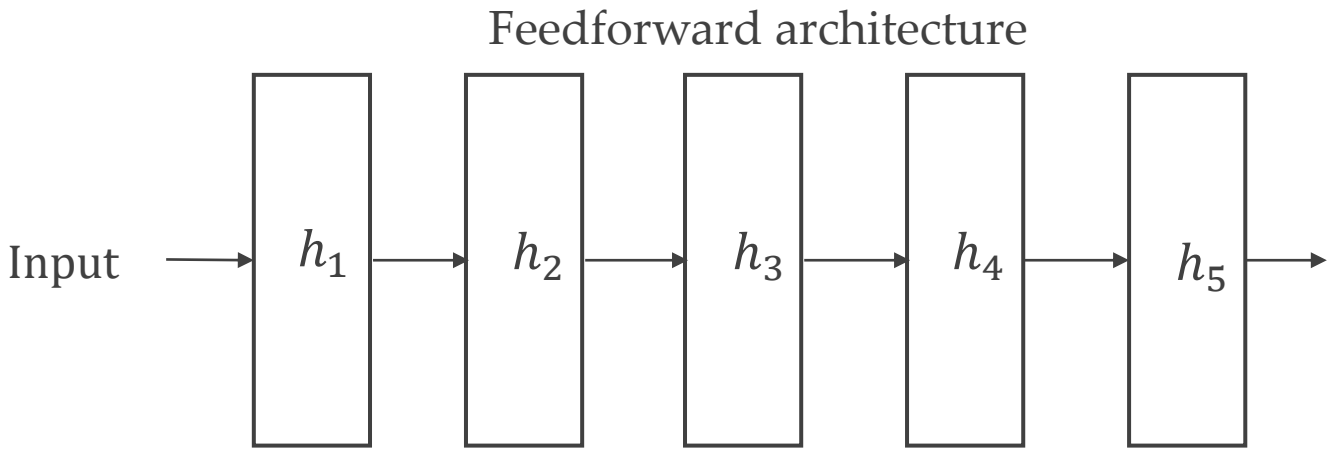


Requirements

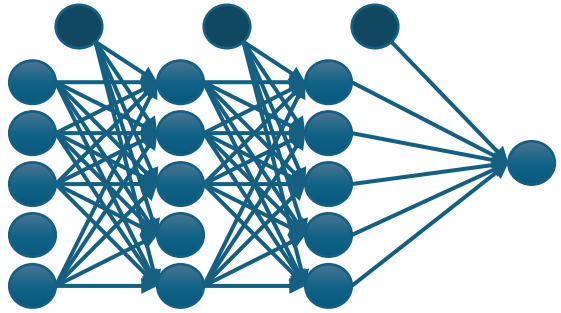
(1) Activations must be **1st-order differentiable (almost) everywhere.**

(2) Take special care when there are cycles in the architecture of blocks.

Most models are feedforward networks (e.g., CNNs, Transformers).



Training goal and overview



We have a dataset of inputs and outputs.

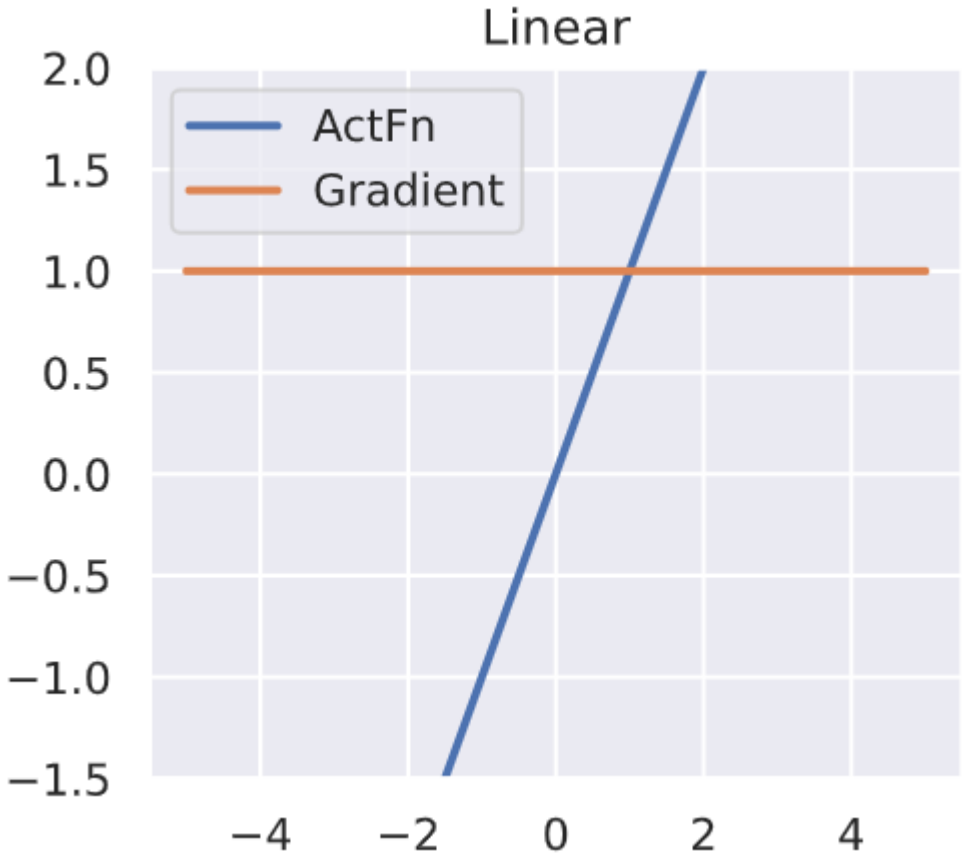
Initialize all weights and biases with random values.

Learn weights and biases through “forward-backward” propagation.

- Forward step: Map input to predicted output.
- Loss step: Compare predicted output to ground truth output.
- Backward step: Correct predictions by propagating gradients.

The linear / fully-connected layer

$$\boldsymbol{x} \in \mathbb{R}^{1\times M}, \boldsymbol{w} \in \mathbb{R}^{N\times M}$$
$$h(x; w) = \boldsymbol{x} \cdot \boldsymbol{w}^T + b$$
$$\frac{dh}{d\boldsymbol{x}} = \boldsymbol{w}$$



Identity activation function.

No activation saturation.

Hence, strong & stable gradients.

Reliable learning with linear modules.

Forward propagation

When using linear layers, essentially repeated application of perceptrons:

1. Start from the input, multiply with weights, sum, add bias.
2. Repeat for all following layers until you reach the end.

There is one main new element (next to the multiple layers):

Activation functions after each layer.

Why have activation functions?

Each hidden/output neuron is a linear sum.

A combination of linear functions is a linear function!

$$v(x) = ax + b$$
$$w(z) = cz + d$$
$$w\big(v(x)\big) = c(ax + b) + d = (ac)x + (cb + d)$$

Activation functions transforms the outputs of each neuron.

This results in non-linear functions.

Activation functions

Defines how the weighted sum of the input is transformed into an output in a layer of the network.

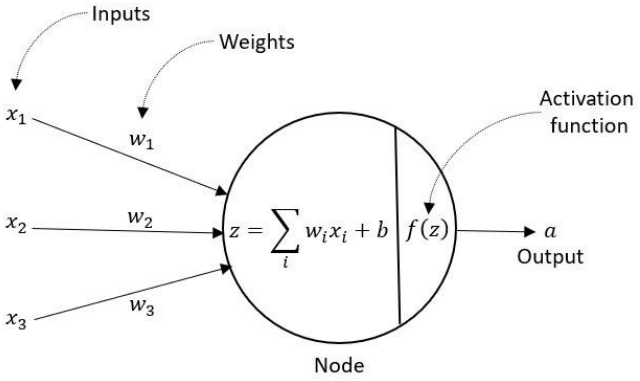
If output range limited, then called a "*squashing function*."

The choice of activation function has a large impact on the capability and performance of the neural network.

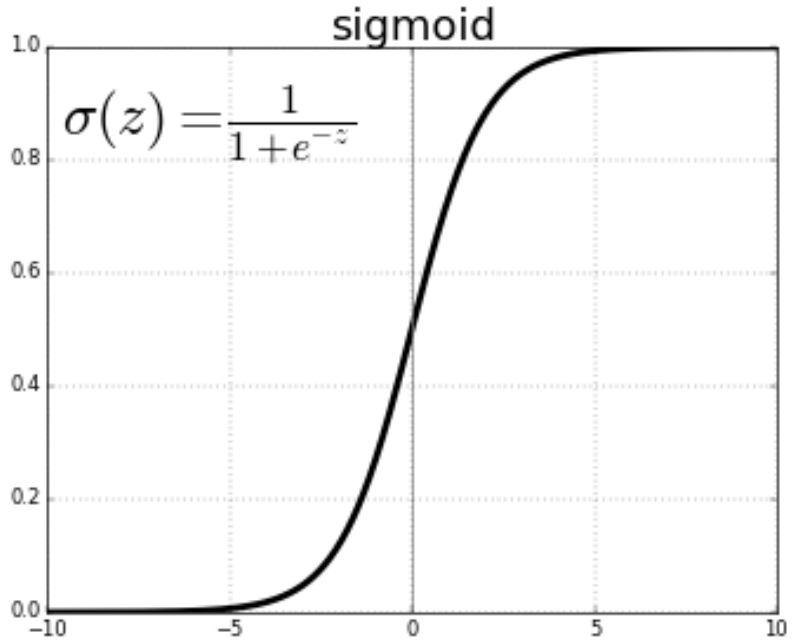
Different activation functions may be combined, but rare.

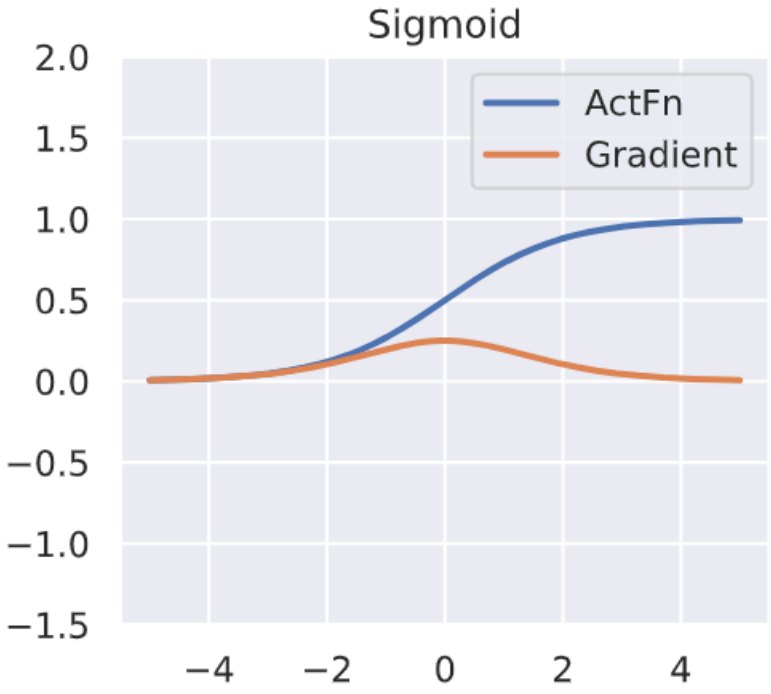
All hidden layers typically use the same activation function.

Need to be differentiable at most points.



The sigmoid activation





The problem with sigmoid activation is vanishing gradient this because sigma is between 0 to 1 and when i multiply them together to compute the derivative it shrinks

Range: (0,1)

Differentiable: $\frac{d}{dz}\sigma(z) = \sigma(z)(1-\sigma(z))$

The tanh activation

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

$$\frac{\partial h}{\partial x} = 1 - tanh^2(x)$$

$tanh(x)$ has better output range $[-1, +1]$.

Data centered around 0 (not 0.5) → stronger gradients

Less "positive" bias for next layers (mean 0, not 0.5)

Both saturate at the extreme → 0 gradients.

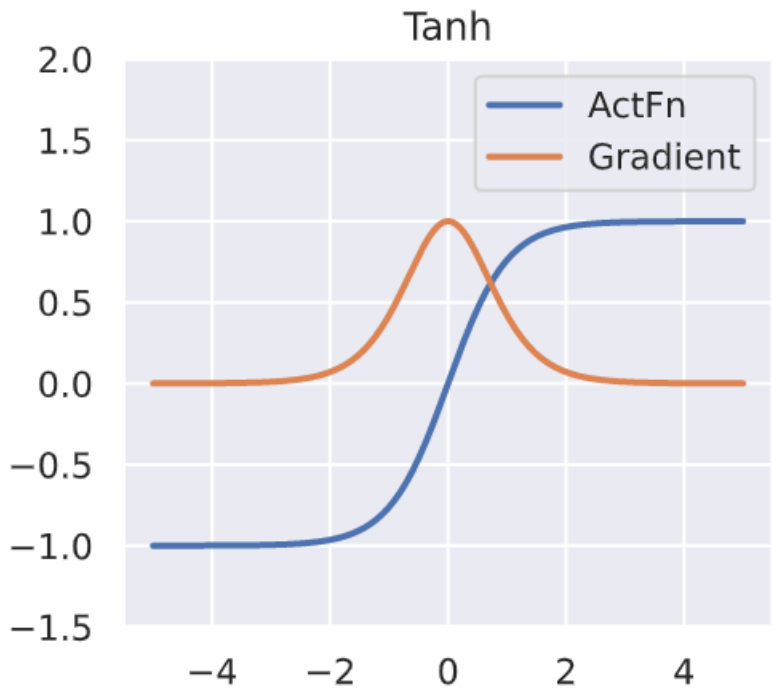
Easily become "overconfident" (0 or 1 decisions)

Undesirable for middle layers

Gradients $\ll 1$ *with chain multiplication*

$tanh(x)$ better for middle layers.

Sigmoids for outputs to emulate probabilities.



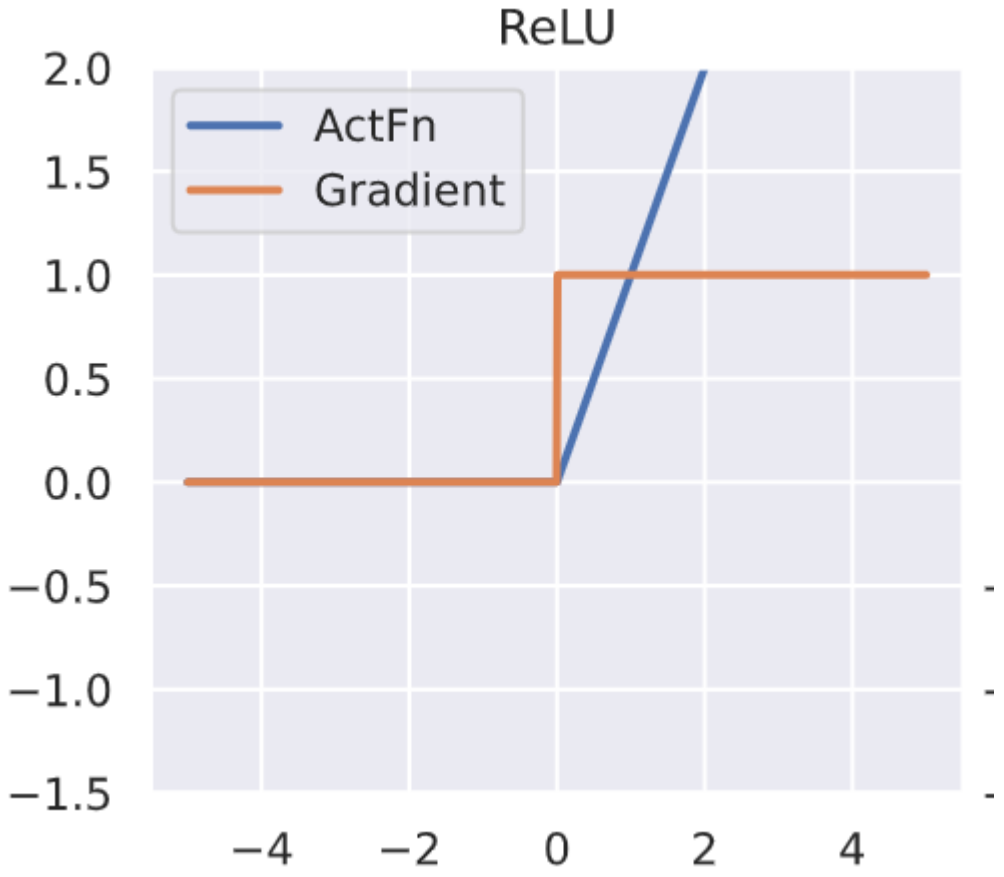
The rectified linear unit (ReLU)

ReLU

$$h(x) = \max(0, x)$$

$$\frac{\partial h}{\partial w} = \begin{cases} 1 \text{ when } x > 0 \\ 0, \text{when } x \leq 0 \end{cases}$$

Note that it is not differentiable in 0 because from the definition of differentiability, to be differentiable the derivatire has to be zero from the right and from the left



Advantages of ReLU

Sparse activation: In randomly initialized network, ~50% active.

Better gradient propagation: Fewer vanishing gradient problems compared to sigmoidal activation functions that saturate in both directions.

Efficient computation: Only comparison, addition and multiplication.

Limitations of ReLU

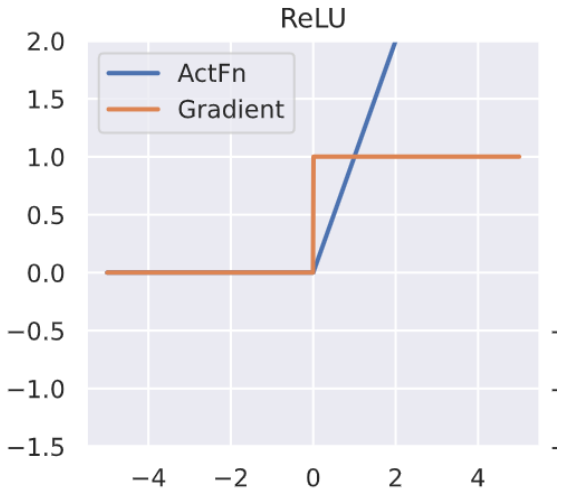
Non-differentiable at zero; however, it is differentiable anywhere else, and the value of the derivative at zero can be arbitrarily chosen to be 0 or 1.

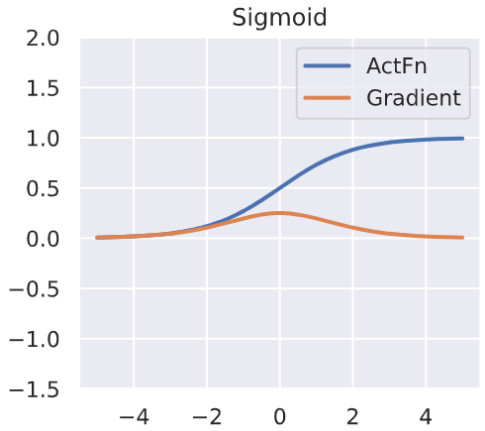
Not zero-centered.

Unbounded.

Dead neurons problem: neurons can sometimes be pushed into states in which they become inactive for essentially all inputs. Higher learning rates might help.

Also known as Dying Relu problem, Since the gradient of ReLU at zero is also zero, backpropagation cannot update the weights, and the neuron remains stuck in this inactive state forever. This can cause large portions of the network to become non-functional, reducing model capacity and potentially halting learning.





ReLU vs sigmoid
Which one is more non-linear?

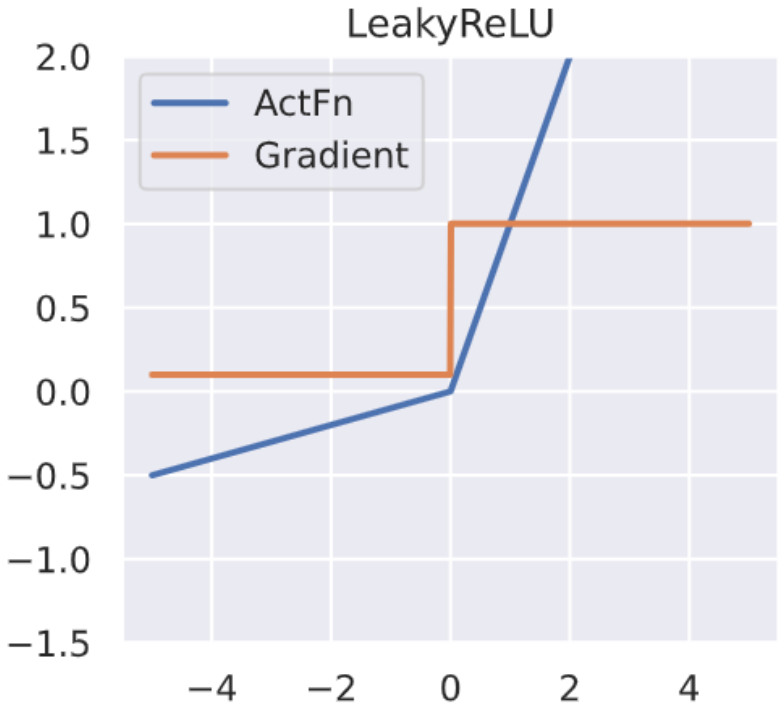
In very large range the sigmoid becomes almost a straight line parallel to the x-axis, while Relu maintains its non linearity

Leaky Relu has been proposed to solve the Dying Relu problem

Leaky ReLU

$$h(x) = \begin{cases} x, \text{when } x > 0 \\ ax, \text{when x} \leq 0 \end{cases}$$

$$\frac{\partial h}{\partial x} = \begin{cases} 1, \text{when } x > 0 \\ a, \text{when x} \leq 0 \end{cases}$$



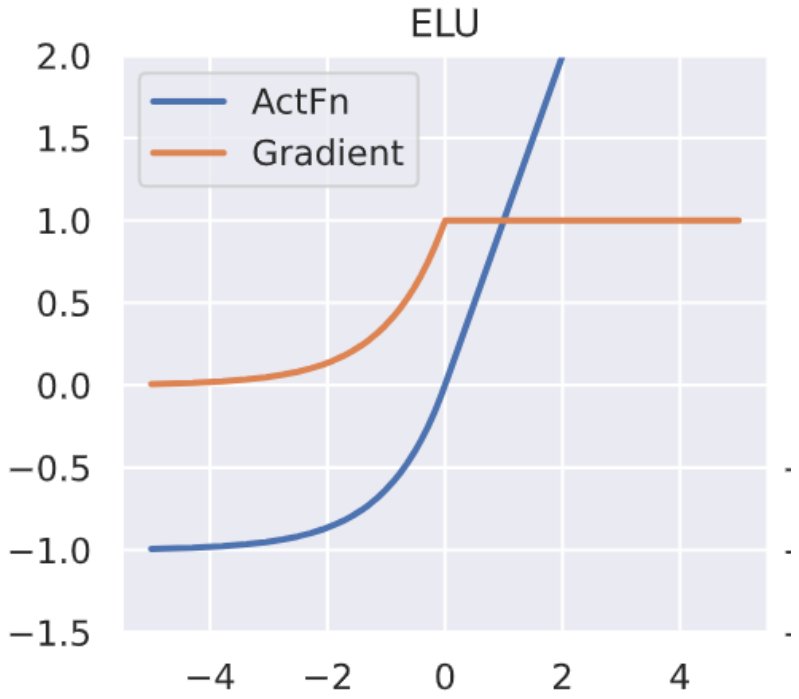
Leaky ReLUs allow a small, positive gradient when the unit is not active.

Parametric ReLUs, or PReLU, treat a as learnable parameter.

Exponential Linear Unit (ELU)

$$h(x) = \begin{cases} x, \text{when } x > 0 \\ \exp(x) - 1, x \leq 0 \end{cases}$$

$$\frac{\partial h}{\partial x} = \begin{cases} 1, \text{when } x > 0 \\ \exp(x), x \leq 0 \end{cases}$$

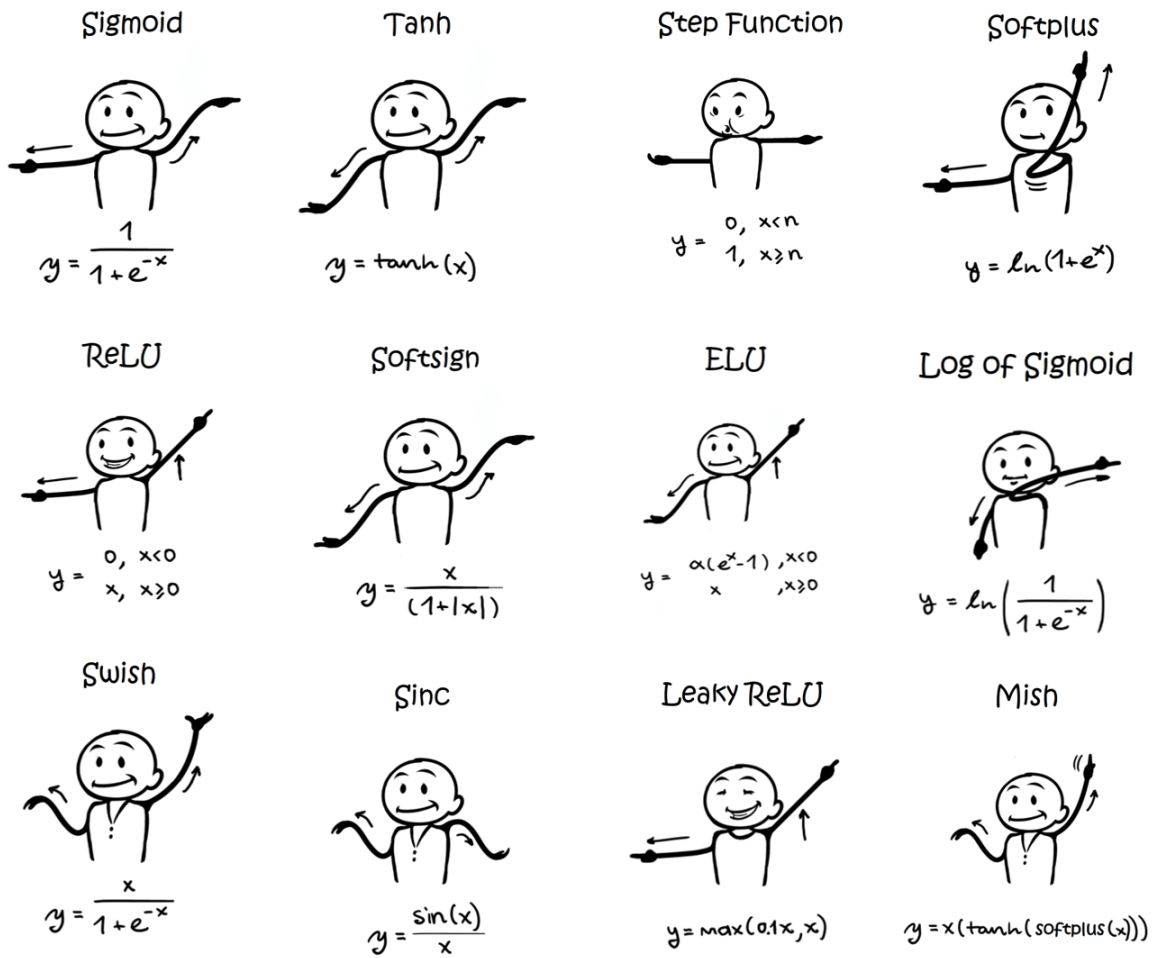


ELU is a smooth approximation to the rectifier.

It has a non-monotonic "bump" when x < 0.

It serves as the default activation for models such as BERT.

The many flavors of activation functions



How to choose an activation function

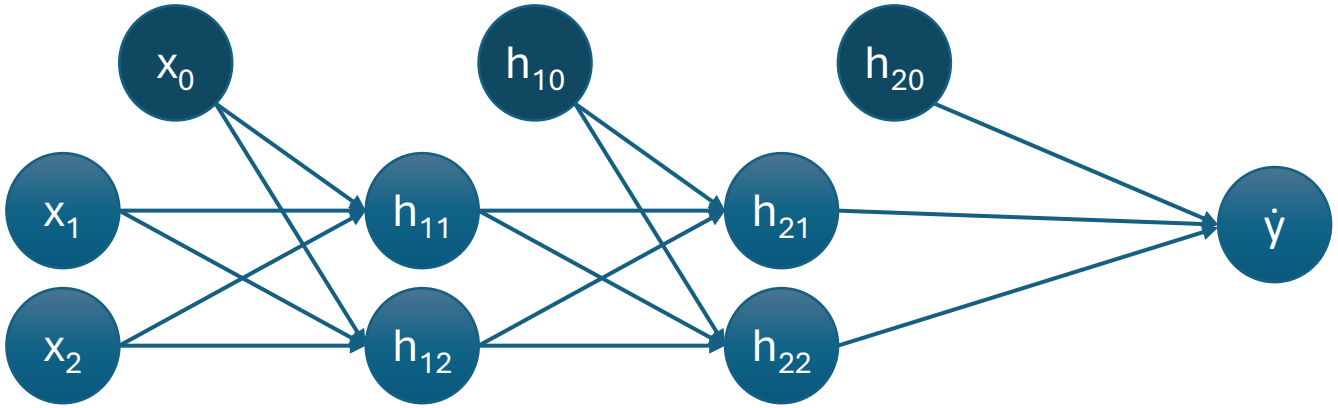
Hidden layers

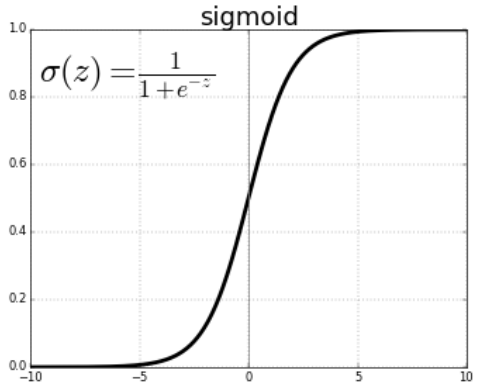
- In modern neural networks, the default recommendation is to use the rectified linear unit (ReLU)
- *(Recurrent Neural Networks:* Tanh and/or Sigmoid activation function.)

Output layer

- Regression: One node, linear activation.
- Binary Classification: One node, sigmoid activation.
- Multiclass Classification: One node per class, softmax activation.

Cost function: binary classification





Let $y_i \in \{0,1\}$ denote the binary label of example i and $p_i \in [0,1]$ denote the output of example i

Our goal: minimize p_i if y_i=0, maximize if y_i = 1

Maximize: $p_i^{y_i} \cdot (1 - p_i)^{(1-y_i)}$

I.e. minimize: $-(y_i \cdot \log(p_i) + (1 - y_i) \cdot \log(1 - p_i))$ **Negative log likelihood aka cross entropy**

distillation: train a network in the labels only to get a prob distribution and then use it as target of my nn on the actual samples

Multi-class classification: softmax

there is still a binary view, we consider it only if it is correct,

Outputs probability distribution.

$$h(x_i) = \frac{e^{x_i}}{\sum_j e^{x_j}}$$

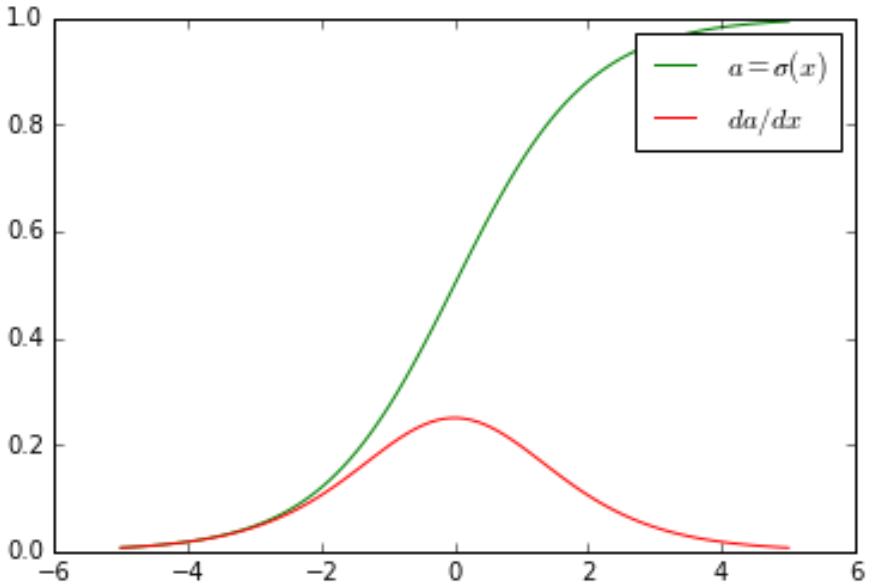
$\sum_{i=1}^{K} h(x_i) = 1$ for K classes or simply normalizes in a non-linear manner.

Avoid exponentianting too large/small numbers for better stability.

$$h(x_i) = \frac{e^{x_i}}{\sum_j e^{x_j}} = \frac{e^{x_i - \mu}}{\sum_j e^{x_j - \mu}} \quad , \mu = \max_i x_i$$

the mu is the temperature, it help to normalizes because often there is one term (which is a very high peak) which dominates the distribution

Loss becomes: $-\sum_{j=1}^{K} y_j \log(\mathrm{p}(\mathrm{x})_j)$



What about settings with multiple classes, where each sample can have multiple classes as prediction?

Use logistic sigmoid for each output node, where the target vector will be no more one-hot encoded

What ordinal classification?

Ordinal classification (also known as ordinal regression) is a type of supervised learning problem where the target categories have a natural order or ranking, but the differences between categories are not necessarily equal or measurable numerically. This distinguishes it from standard multi-class classification, which treats all classes as independent and unrelated.

With ordinal classification we could treat the problem as a regression one

Architecture design

The overall structure of the network:

how many units should it should have

how those units should be connected to each other

Neural networks are organized into groups of units, called layers in a chain structure.

The first layer is given by

$$\boldsymbol{h}^{(1)} = g^{(1)}\left(\boldsymbol{W}^{(1)\top}\boldsymbol{x} + \boldsymbol{b}^{(1)}\right)$$

And the the second layer is

$$\boldsymbol{h}^{(2)} = g^{(2)}\left(\boldsymbol{W}^{(2)\top}\boldsymbol{h}^{(1)} + \boldsymbol{b}^{(2)}\right)$$

Universal approximation theorem

Universal approximation theorem (recap: ML1)

Feedforward networks with hidden layers provide a universal approximation framework.

A large MLP with even a single hidden layer is able to represent any function provided that the network is given enough hidden units.

However, no guarantee that the training algorithm will be able to learn that function

May not be able to find the parameter values that corresponds to the desired function.

Might choose the wrong function due to overfitting.

How many hidden units?

Width and depth

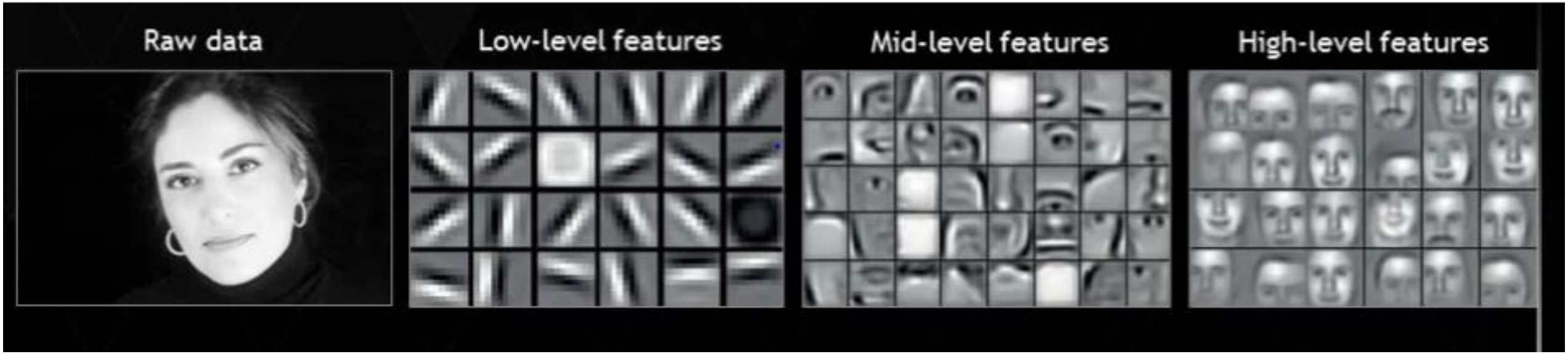
In the worse case, an exponential number of hidden units

> *a deep rectifier net can require an exponential number of hidden units with a shallow (one hidden layer) network*.

We like deep models in deep learning:

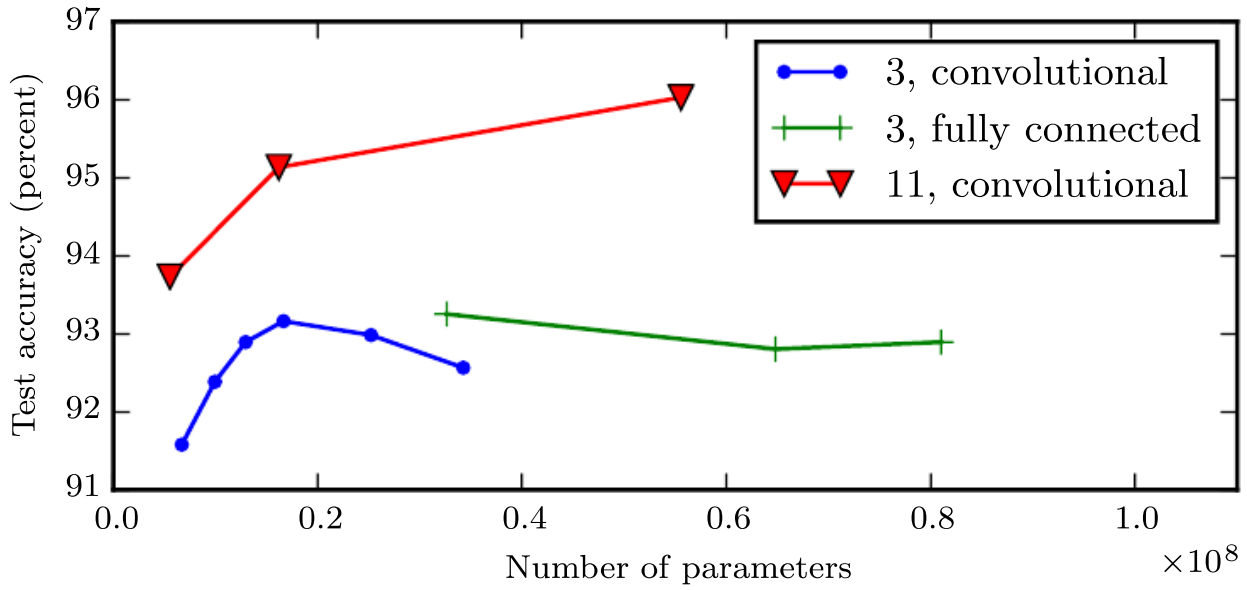
1. can reduce the number of units required to represent the desired function.
2. can reduce the amount of generalization error.
3. deeper networks often generalize better.

Deep networks and pattern hierarchies



An empirical result on width and depth

Increasing the number of parameters in layers of ConvNets without increasing depth is not nearly as effective at increasing test set performance.



How units are connected between layers also matters

FFN: A jungle of architectures

Perceptrons, MLPs

RNNs, LSTMs, GRUs

Vanilla, Variational, Denoising Autoencoders

Hopfield Nets, Restricted Boltzmann Machines

Convolutional Nets, Deconvolutional Nets

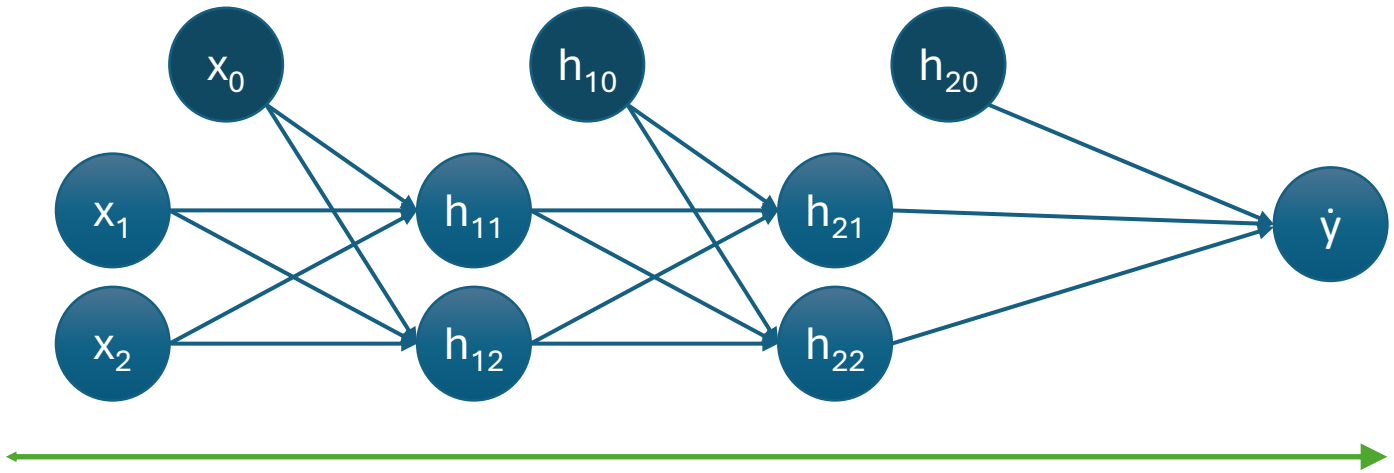
Generative Adversarial Nets

Deep Residual Nets, Neural Turing Machines

Transformers

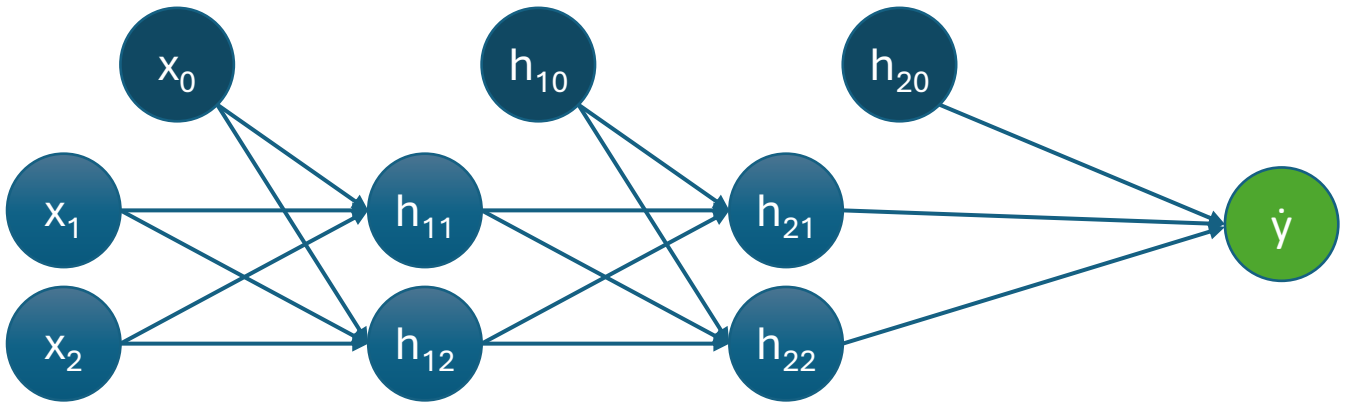
They all rely on modules

Training deep networks: summary



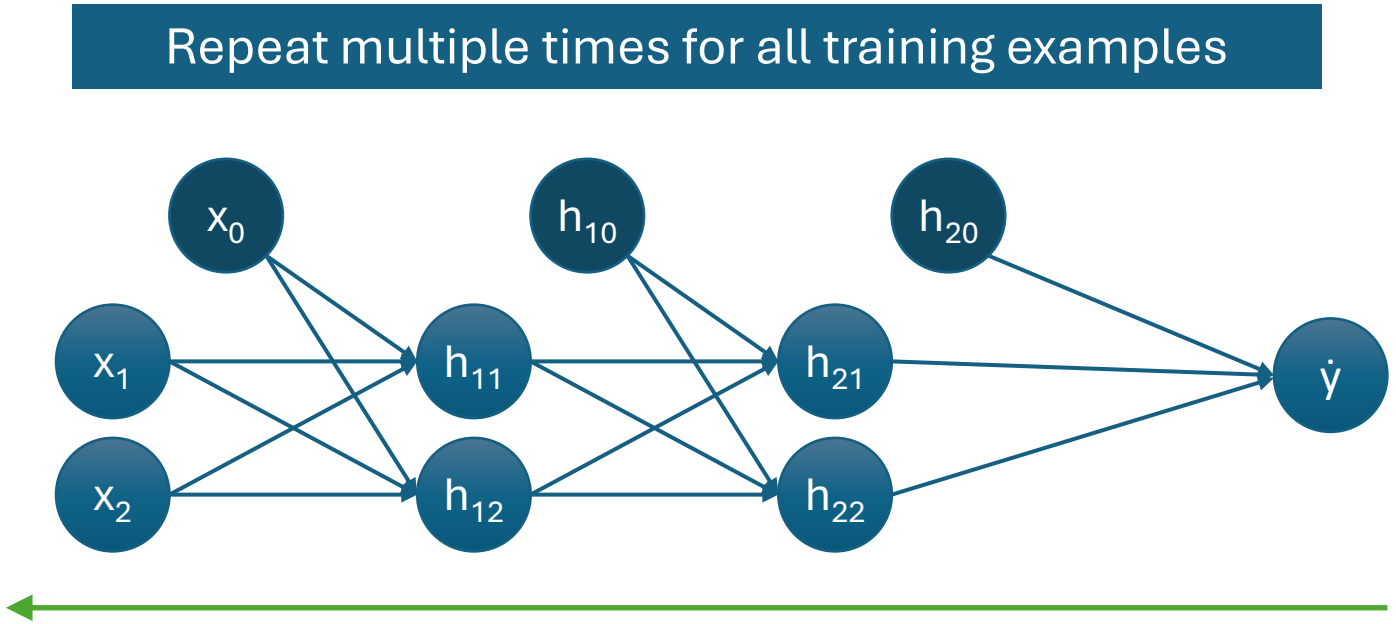
1. Move input through network to yield prediction

Training deep networks: summary



1. Move input through network to yield prediction
2. Compare prediction to ground truth label

Training deep networks: summary



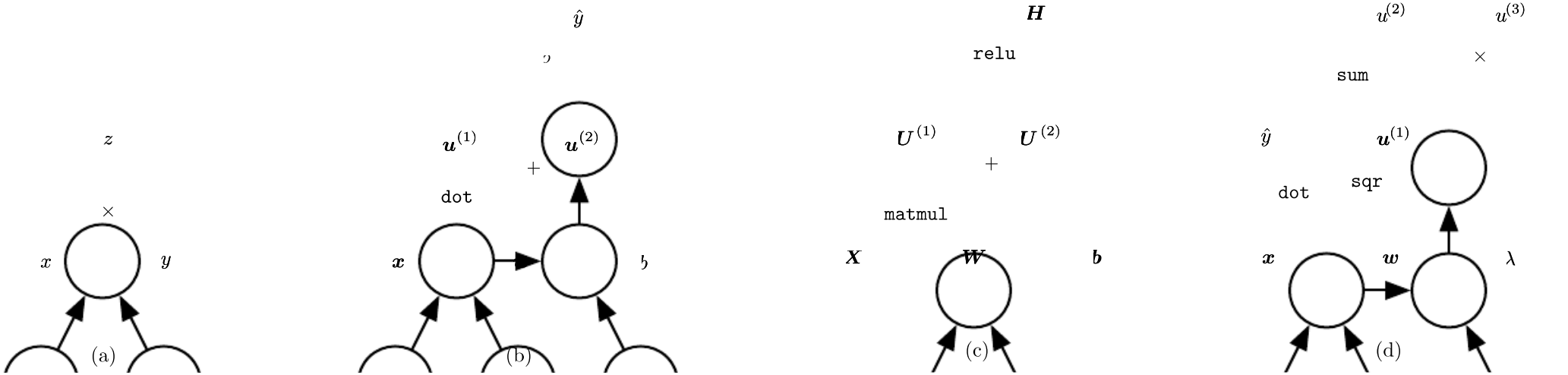
1. Move input through network to yield prediction
2. Compare prediction to ground truth label
3. Backpropagate errors to all weights

Break

Functions as computation graphs

For backprop, it is helpful to view function compositions as graphs.

Each node in the graph indicates a variable, an operation is a simple function of one or more variables.



Backprop: chain rule as an algorithm

The neural network loss is a composite function of modules.

We want the gradient w.r.t. to the parameters of the l layer.

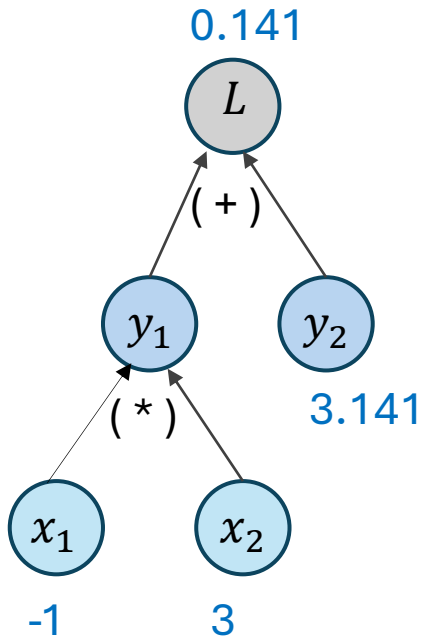
$$\frac{d\mathcal{L}}{dw_l} = \frac{d\mathcal{L}}{dh_L} \cdot \frac{dh_L}{dh_{L-1}} \cdot \ldots \cdot \frac{dh_l}{dw_l} \quad \Rightarrow \quad \frac{d\mathcal{L}}{dw_l} = \frac{d\mathcal{L}}{dh_l} \cdot \frac{dh_l}{dw_l}$$

Gradient of loss w.r.t. the module output

Gradient of a module w.r.t. its parameters

Back-propagation is an algorithm that computes the chain rule, with a specific **order of operations that is highly efficient**.

Chain rule with computation graphs



dL / dx1 = ?

= dL/dy1 * dy1/dx1

now L = y1 + y2

so dL/dy1 = 1

= 1* dy1/dx1

now y1 = x1 * x2

so dy1/dx1 = x2

= 1*x2

Visual example of compositionality

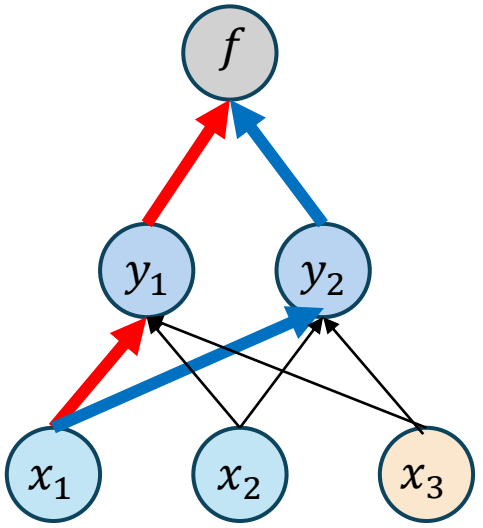
For $\mathrm{h} = f \circ y(x)$, here f, and y's denote functions

$$\frac{dh}{dx} = \frac{df}{dy}\frac{dy}{dx} = \begin{bmatrix} \frac{\partial f}{\partial y_1} & \frac{\partial f}{\partial y_2} \end{bmatrix} \begin{bmatrix} \frac{\partial y_1}{\partial x_1} & \frac{\partial y_1}{\partial x_2} & \frac{\partial y_1}{\partial x_3} \\ \frac{\partial y_2}{\partial x_1} & \frac{\partial y_2}{\partial x_2} & \frac{\partial y_2}{\partial x_3} \end{bmatrix}$$

Focusing on one of the partial derivatives: $\frac{dh}{dx_1}$

$$\frac{dh}{dx_1} = \frac{df}{dy_1}\frac{dy_1}{dx_1} + \frac{df}{dy_2}\frac{dy_2}{dx_1}$$

The partial derivative depends on all paths from f to x_i.



Why you should know about backprop

Backprop is the foundation of modern deep learning.

However, we want to have backward propagation be automated.

If we know how deep learning is broken down, we can handle any new algorithm advance, i.e., we will remain relevant.

Backprop derivation and implementation also a popular interview question!

Four ways to differentiate

Manual

Numerical

Symbolic

Automatic

Numerical differentiation

Idea: we don't care about the actual derivative, we just need to know the slope, so let's compute it directly.

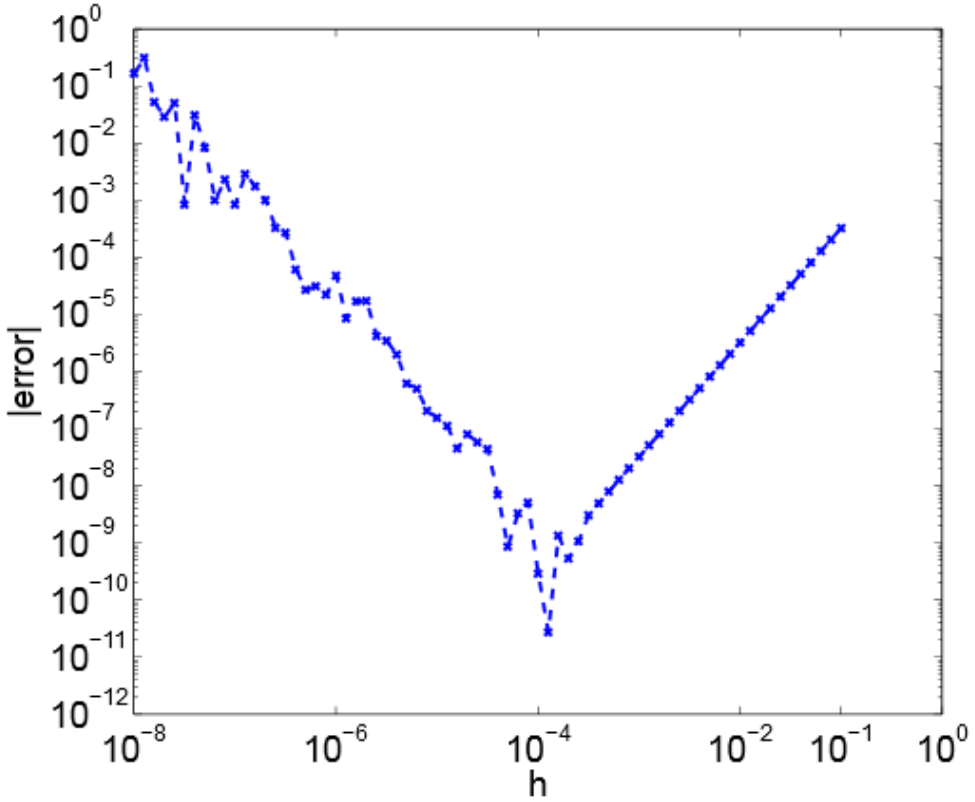
$$\frac{df}{dx} = \lim_{h \to 0} \frac{f(x+h) - f(x)}{h}$$

Pro: works even when function itself is unknown.

Con: unstable, sensitive to approximation errors, computationally heavy.

Truncation and round-off errors

Round-off error: h too small



Trunction error: h too big

Symbolic Differentiation

Let the compute figure out the closed-form expression through calculus.

Pro: no more approximation errors.

I'm doing too much expanding all the terms

Con: expression swell.

$$f(x) = \frac{e^{wx+b} + e^{-(wx+b)}}{e^{wx+b} - e^{-(wx+b)}}$$

$$\frac{\partial f}{\partial w} = \frac{(-xe^{-b-wx} - xe^{b+wx})(e^{-b-wx} + e^{b+wx})}{(-e^{-b-wx} + e^{b+wx})^2} + \frac{-xe^{-b-wx} + xe^{b+wx}}{-e^{-b-wx} + e^{b+wx}}$$

Automatic differentiation

Main idea:

Express functions into its variables and elementary operations.

Pre-define the derivatives of the elementary operations.

Chain the elementary derivatives without repeating operations over and over.

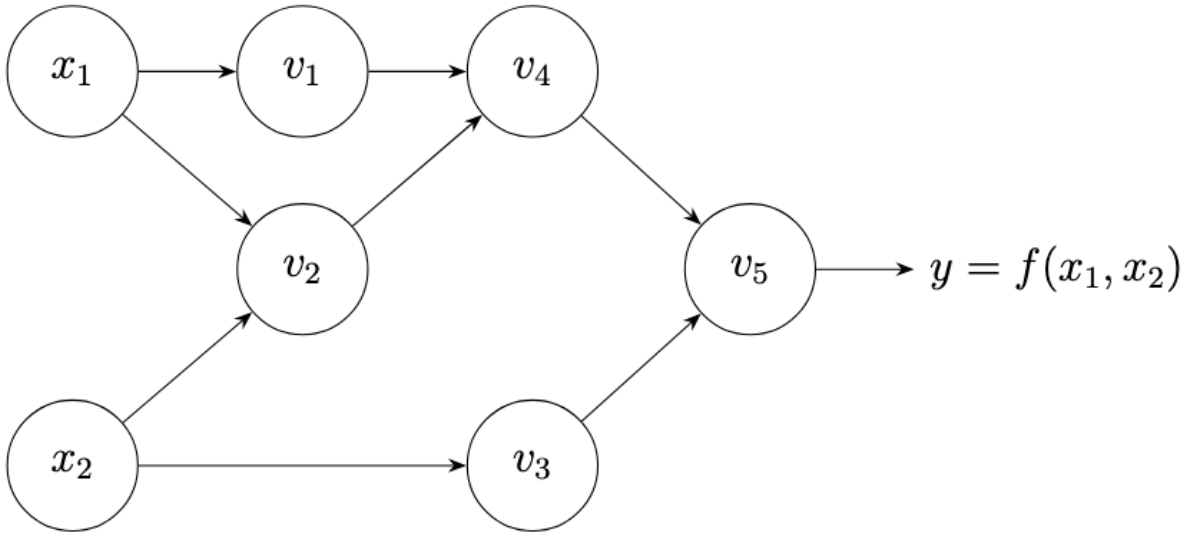
Forward and backward propagation by computers.

Based on the excellent survey “Automatic differentiation in machine learning: a survey”, Baydin et al. JMLR 2018.

Forward mode AutoDiff by example

Consider the following function and computation graph:

$$y = f(x_1, x_2) = \log x_1 + x_1 x_2 - \sin x_2$$

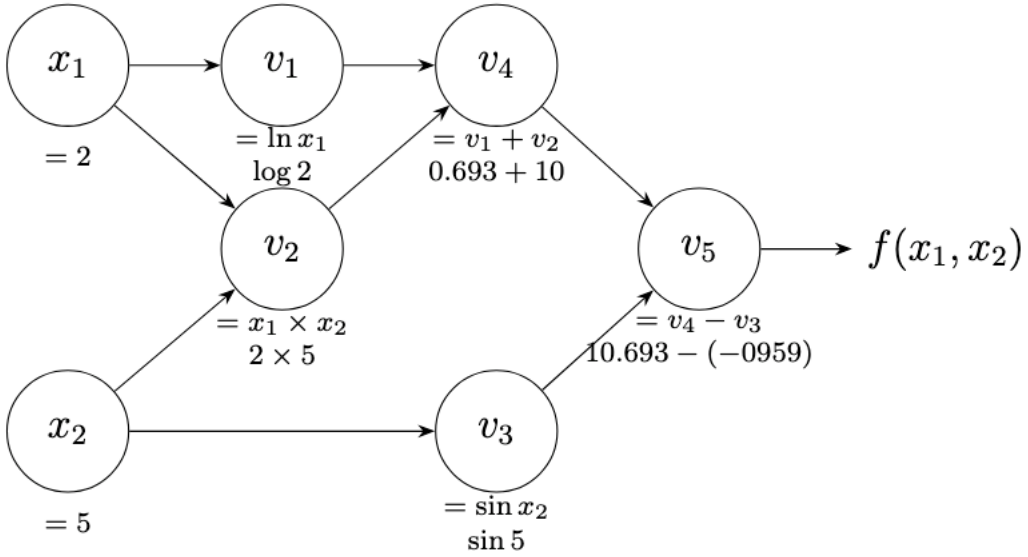


Forward mode AutoDiff by example

We first do the forward pass of this “network” for input (2,5).

$$y = f(x_1, x_2) = \log x_1 + x_1 x_2 - \sin x_2$$

x_1		$= 2$
x_2		$= 5$
v_1	$= \log x_1$	$= \log 2$
v_2	$= x_1 \times x_2$	$= 2 \times 5$
v_3	$= \sin x_2$	$= \sin 5$
v_4	$= v_1 + v_2$	$= 0.693 + 10$
v_5	$= v_4 - v_3$	$= 10.693 + 0.959$
y	$= v_5$	$= 11.652$



Forward mode AutoDiff by example

Forward mode AutoDiff computes the derivative of the output w.r.t. one input.

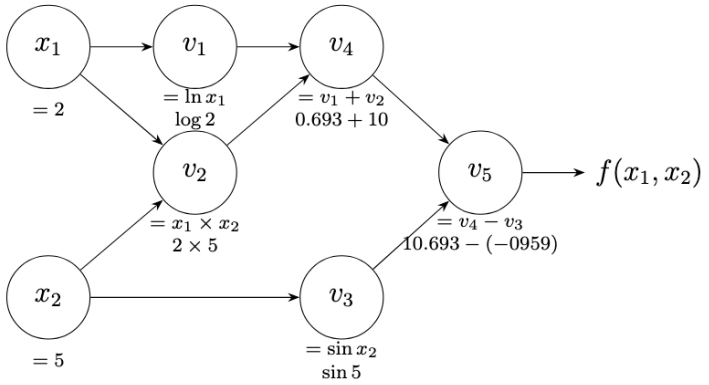
Let's compute the partial derivative $\partial y / \partial x_1$

Throughout forward mode AutoDiff, we use the following notation: $\dot{v} = \frac{\partial v}{\partial x_1}$

Forward mode AutoDiff by example

Forward trace of our function:

$$y = f(x_1, x_2) = \log x_1 + x_1 x_2 - \sin x_2$$



$$
\begin{array}{lll}
\dot{x}_1 & = \partial x_1 / \partial x_1 & = 1 \\
\dot{x}_2 & = \partial x_2 / \partial x_1 & = 0 \\
\hline
\dot{v}_1 & = \dot{x}_1 / x_1 & = 1/2 \\
\dot{v}_2 & = \dot{x}_1 \times x_2 + \dot{x}_2 \times x_1 & = 1 \times 5 + 0 \times 2 \\
\dot{v}_3 & = \dot{x}_2 \times \cos x_2 & = 0 \times \cos 5 \\
\dot{v}_4 & = \dot{v}_1 + \dot{v}_2 & = 0.5 + 5 \\
\dot{v}_5 & = \dot{v}_4 - \dot{v}_3 & = 5.5 - 0 \\
\hline
\dot{y} & = \dot{v}_5 & = 5.5
\end{array}
$$

$v_1 = \log x_1$

$$\dot{v}_1 = \frac{\partial v_1}{\partial x_1} \times \frac{\partial x_1}{\partial x_1} = \frac{1}{x_1} \times \frac{\partial x_1}{\partial x_1} = \frac{\dot{x}_1}{x_1}$$

= 1 = 1/x1

Reverse mode AutoDiff by example

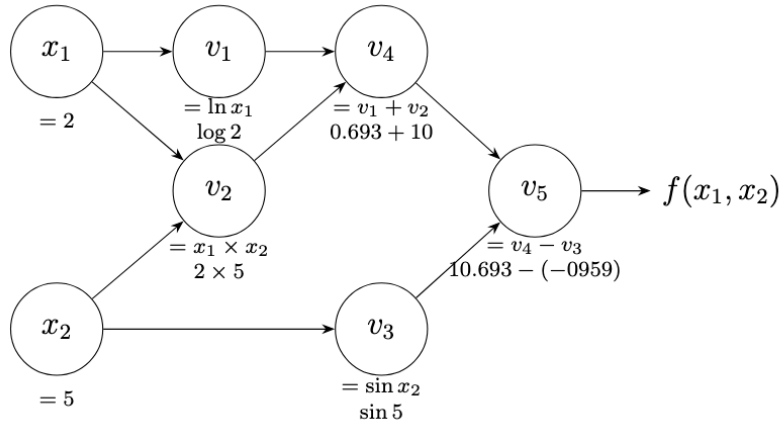
Reverse mode AutoDiff computes the derivative of the output w.r.t. all inputs.

New notation (like we normally use): $\bar{v} = \frac{\partial y}{\partial v}$

Let's do the reverse mode to calculate: $\frac{\partial y}{\partial x_2}$

Reverse mode AutoDiff by example

$$y = f(x_1, x_2) = \log x_1 + x_1 x_2 - \sin x_2$$



If I store the already computed derivatives when going backward in the network thanks to the chain rule I will have to compute only one new derivative per node

x_1		$= 2$
x_2		$= 5$
v_1	$= \log x_1$	$= \log 2$
v_2	$= x_1 \times x_2$	$= 2 \times 5$
v_3	$= \sin x_2$	$= \sin 5$
v_4	$= v_1 + v_2$	$= 0.693 + 10$
v_5	$= v_4 - v_3$	$= 10.693 + 0.959$
y	$= v_5$	$= 11.652$

$\bar{x}_1$			$= 5.5$
$\bar{x}_2$			$= 1.716$
$\bar{x}_1$	$= \bar{x}_1 + \bar{v}_1 \frac{\partial v_1}{\partial x_1}$	$= \bar{x}_1 + \bar{v}_1 / x_1$	$= 5.5$
$\bar{x}_2$	$= \bar{x}_2 + \bar{v}_2 \frac{\partial v_2}{\partial x_2}$	$= \bar{x}_2 + \bar{v}_2 \times x_1$	$= 1.716$
$\bar{x}_1$	$= \bar{v}_2 \frac{\partial v_2}{\partial x_1}$	$= \bar{v}_2 \times x_2$	$= 5$
$\bar{x}_2$	$= \bar{v}_3 \frac{\partial v_3}{\partial x_2}$	$= \bar{v}_3 \times \cos x_2$	$= -0.284$
$\bar{v}_2$	$= \bar{v}_4 \frac{\partial v_4}{\partial v_2}$	$= \bar{v}_4 \times 1$	$= 1$
$\bar{v}_1$	$= \bar{v}_4 \frac{\partial v_4}{\partial v_1}$	$= \bar{v}_4 \times 1$	$= 1$
$\bar{v}_3$	$= \bar{v}_5 \frac{\partial v_5}{\partial v_3}$	$= \bar{v}_5 \times (-1)$	$= -1$
$\bar{v}_4$	$= \bar{v}_5 \frac{\partial v_5}{\partial v_4}$	$= \bar{v}_5 \times 1$	$= 1$
$\bar{v}_5$	$= \bar{y}$	$= 1$	

$$\bar{v}_4 = \frac{\partial y}{\partial v_4} = \frac{\partial y}{\partial v_5}\frac{\partial v_5}{\partial v_4} = \bar{v}_5 \frac{\partial v_5}{\partial v_4} = \bar{v}_5 \times 1$$

$$\bar{v}_5 = \frac{\partial y}{\partial v_5} = \frac{\partial y}{\partial y} = 1$$

AutoDiff summarized

For general function h: $\mathbb{R}^m \rightarrow \mathbb{R}^n$

m forward mode differentiations and *n* reverse mode differentiations.

One of the two is known as backprop, which one?

Reverse mode.

Why is reverse mode differentiation, the norm in deep learning?

We do deep learning on high-dimensional data.

Autodiff visually

Forward propagation

$h_0 = x$

$h_1 = \sigma(w_1 h_0)$ → Store h_1 . Remember that $\partial_x \sigma = \sigma \cdot (1 - \sigma)$

$h_2 = \sigma(w_2 h_1)$ → Store h_2

$\mathcal{L} = 0.5 \cdot \|l - h_2\|^2$

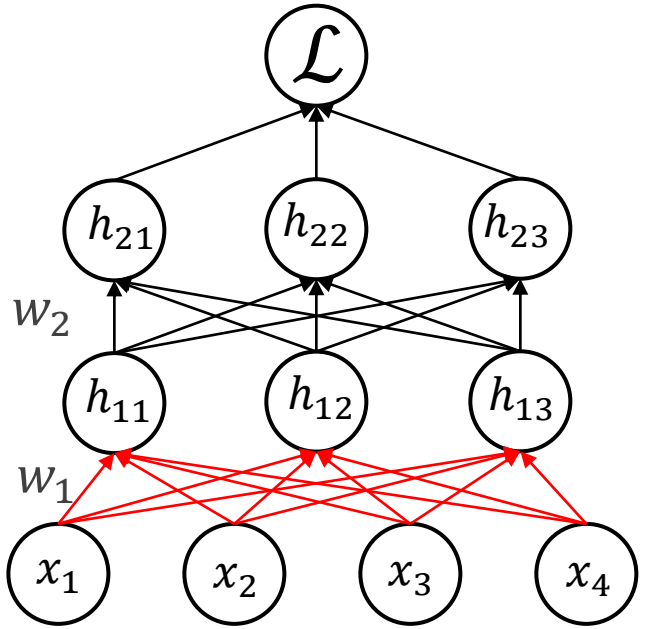
Backward propagation

$$\frac{d\mathcal{L}}{dh_2} = -(y^* - h_2)$$

$$\frac{d\mathcal{L}}{dw_2} = \frac{d\mathcal{L}}{dh_2}\frac{dh_2}{dw_2} = \frac{d\mathcal{L}}{dh_2} h_1 \sigma(w_2 h_1)\big(1 - \sigma(w_2 h_1)\big) = \frac{d\mathcal{L}}{dh_2} h_1 h_2 (1 - h_2)$$

$$\frac{d\mathcal{L}}{dh_1} = \frac{d\mathcal{L}}{dh_2}\frac{dh_2}{dh_1} = \frac{d\mathcal{L}}{dh_2} w_2 \sigma(w_2 h_1)\big(1 - \sigma(w_2 h_1)\big) = \frac{d\mathcal{L}}{dh_2} w_2 h_2 (1 - h_2)$$

$$\frac{d\mathcal{L}}{dw_1} = \frac{d\mathcal{L}}{dh_1}\frac{dh_1}{dw_1} = \frac{d\mathcal{L}}{dh_1} h_0 \sigma(w_1 h_0)\big(1 - \sigma(w_1 h_0)\big) = \frac{d\mathcal{L}}{dh_1} h_0 h_1 (1 - h_1)$$



Autodiff visually

Forward propagation

$h_0 = x$

$h_1 = \sigma(w_1 h_0)$ → Store h_1 . Remember that $\partial_x \sigma = \sigma \cdot (1 - \sigma)$

$h_2 = \sigma(w_2 h_1)$ → Store h_2

$\mathcal{L} = 0.5 \cdot \|l - h_2\|^2$

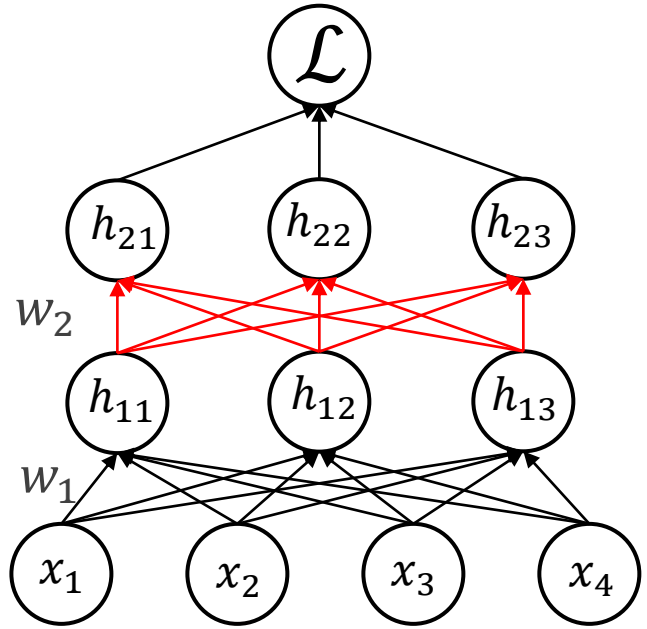
Backward propagation

$$\frac{d\mathcal{L}}{dh_2} = -(y^* - h_2)$$

$$\frac{d\mathcal{L}}{dw_2} = \frac{d\mathcal{L}}{dh_2}\frac{dh_2}{dw_2} = \frac{d\mathcal{L}}{dh_2} h_1 \sigma(w_2 h_1)\big(1 - \sigma(w_2 h_1)\big) = \frac{d\mathcal{L}}{dh_2} h_1 h_2 (1 - h_2)$$

$$\frac{d\mathcal{L}}{dh_1} = \frac{d\mathcal{L}}{dh_2}\frac{dh_2}{dh_1} = \frac{d\mathcal{L}}{dh_2} w_2 \sigma(w_2 h_1)\big(1 - \sigma(w_2 h_1)\big) = \frac{d\mathcal{L}}{dh_2} w_2 h_2 (1 - h_2)$$

$$\frac{d\mathcal{L}}{dw_1} = \frac{d\mathcal{L}}{dh_1}\frac{dh_1}{dw_1} = \frac{d\mathcal{L}}{dh_1} h_0 \sigma(w_1 h_0)\big(1 - \sigma(w_1 h_0)\big) = \frac{d\mathcal{L}}{dh_1} h_0 h_1 (1 - h_1)$$



Autodiff visually

Forward propagation

$h_0 = x$

$h_1 = \sigma(w_1 h_0)$ → Store h_1 . Remember that $\partial_x \sigma = \sigma \cdot (1 - \sigma)$

$h_2 = \sigma(w_2 h_1)$ → Store h_2

$\mathcal{L} = 0.5 \cdot \|l - h_2\|^2$

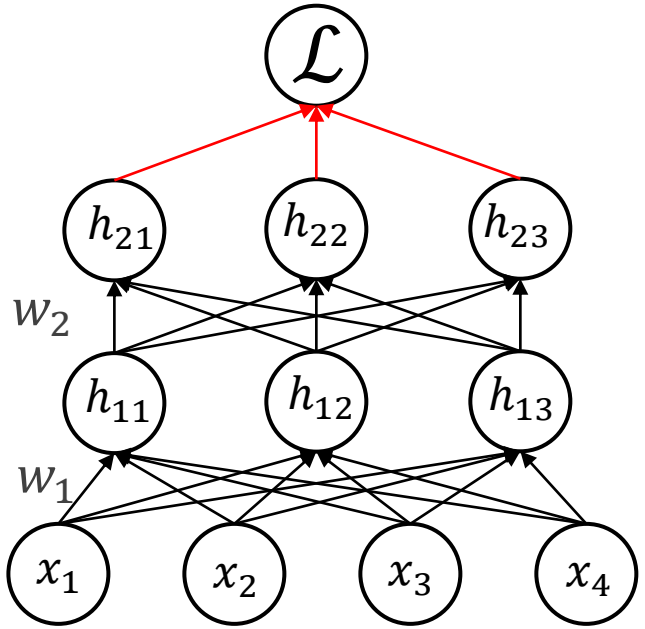
Backward propagation

$$\frac{d\mathcal{L}}{dh_2} = -(y^* - h_2)$$

$$\frac{d\mathcal{L}}{dw_2} = \frac{d\mathcal{L}}{dh_2}\frac{dh_2}{dw_2} = \frac{d\mathcal{L}}{dh_2} h_1 \sigma(w_2 h_1)\big(1 - \sigma(w_2 h_1)\big) = \frac{d\mathcal{L}}{dh_2} h_1 h_2 (1 - h_2)$$

$$\frac{d\mathcal{L}}{dh_1} = \frac{d\mathcal{L}}{dh_2}\frac{dh_2}{dh_1} = \frac{d\mathcal{L}}{dh_2} w_2 \sigma(w_2 h_1)\big(1 - \sigma(w_2 h_1)\big) = \frac{d\mathcal{L}}{dh_2} w_2 h_2 (1 - h_2)$$

$$\frac{d\mathcal{L}}{dw_1} = \frac{d\mathcal{L}}{dh_1}\frac{dh_1}{dw_1} = \frac{d\mathcal{L}}{dh_1} h_0 \sigma(w_1 h_0)\big(1 - \sigma(w_1 h_0)\big) = \frac{d\mathcal{L}}{dh_1} h_0 h_1 (1 - h_1)$$



Autodiff visually

Forward propagation

$h_0 = x$

$h_1 = \sigma(w_1 h_0)$ → Store h_1 . Remember that $\partial_x \sigma = \sigma \cdot (1 - \sigma)$

$h_2 = \sigma(w_2 h_1)$ → Store h_2

$\mathcal{L} = 0.5 \cdot \|l - h_2\|^2$

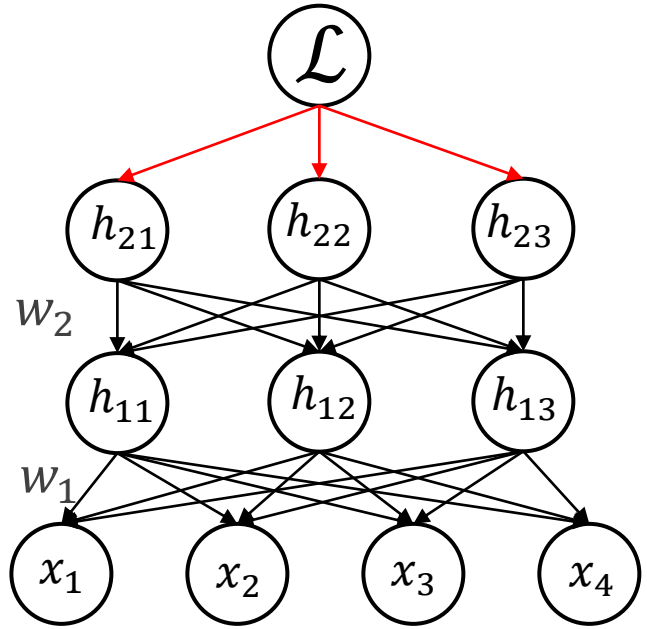
Backward propagation

$$\frac{d\mathcal{L}}{dh_2} = -(y^* - h_2)$$

$$\frac{d\mathcal{L}}{dw_2} = \frac{d\mathcal{L}}{dh_2}\frac{dh_2}{dw_2} = \frac{d\mathcal{L}}{dh_2} h_1 \sigma(w_2 h_1)\big(1 - \sigma(w_2 h_1)\big) = \frac{d\mathcal{L}}{dh_2} h_1 h_2 (1 - h_2)$$

$$\frac{d\mathcal{L}}{dh_1} = \frac{d\mathcal{L}}{dh_2}\frac{dh_2}{dh_1} = \frac{d\mathcal{L}}{dh_2} w_2 \sigma(w_2 h_1)\big(1 - \sigma(w_2 h_1)\big) = \frac{d\mathcal{L}}{dh_2} w_2 h_2 (1 - h_2)$$

$$\frac{d\mathcal{L}}{dw_1} = \frac{d\mathcal{L}}{dh_1}\frac{dh_1}{dw_1} = \frac{d\mathcal{L}}{dh_1} h_0 \sigma(w_1 h_0)\big(1 - \sigma(w_1 h_0)\big) = \frac{d\mathcal{L}}{dh_1} h_0 h_1 (1 - h_1)$$



Autodiff visually

Forward propagation

$h_0 = x$

$h_1 = \sigma(w_1 h_0)$ → Store h_1 . Remember that $\partial_x \sigma = \sigma \cdot (1 - \sigma)$

$h_2 = \sigma(w_2 h_1)$ → Store h_2

$\mathcal{L} = 0.5 \cdot \|l - h_2\|^2$

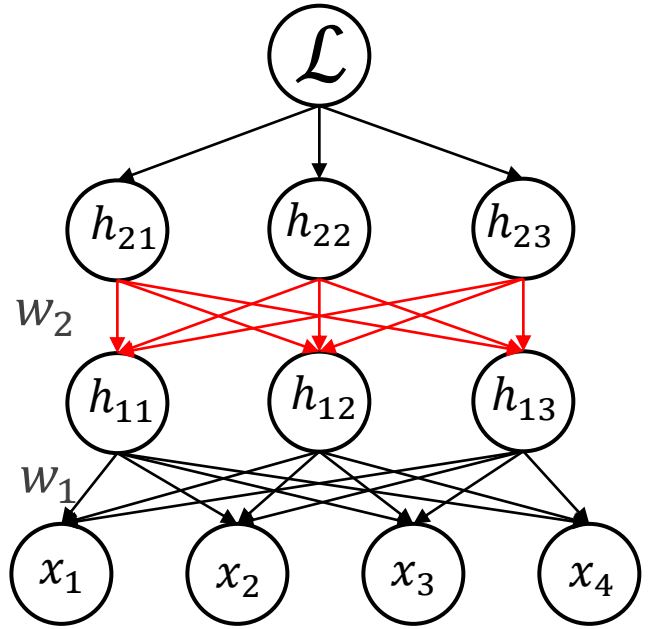
Backward propagation

$$\frac{d\mathcal{L}}{dh_2} = -(y^* - h_2)$$

$$\frac{d\mathcal{L}}{dw_2} = \frac{d\mathcal{L}}{dh_2}\frac{dh_2}{dw_2} = \frac{d\mathcal{L}}{dh_2} h_1 \sigma(w_2 h_1)\big(1 - \sigma(w_2 h_1)\big) = \frac{d\mathcal{L}}{dh_2} h_1 h_2 (1 - h_2)$$

$$\frac{d\mathcal{L}}{dh_1} = \frac{d\mathcal{L}}{dh_2}\frac{dh_2}{dh_1} = \frac{d\mathcal{L}}{dh_2} w_2 \sigma(w_2 h_1)\big(1 - \sigma(w_2 h_1)\big) = \frac{d\mathcal{L}}{dh_2} w_2 h_2 (1 - h_2)$$

$$\frac{d\mathcal{L}}{dw_1} = \frac{d\mathcal{L}}{dh_1}\frac{dh_1}{dw_1} = \frac{d\mathcal{L}}{dh_1} h_0 \sigma(w_1 h_0)\big(1 - \sigma(w_1 h_0)\big) = \frac{d\mathcal{L}}{dh_1} h_0 h_1 (1 - h_1)$$



Autodiff visually

Forward propagation

$h_0 = x$

$h_1 = \sigma(w_1 h_0)$ → Store h_1 . Remember that $\partial_x \sigma = \sigma \cdot (1 - \sigma)$

$h_2 = \sigma(w_2 h_1)$ → Store h_2

$\mathcal{L} = 0.5 \cdot \|l - h_2\|^2$

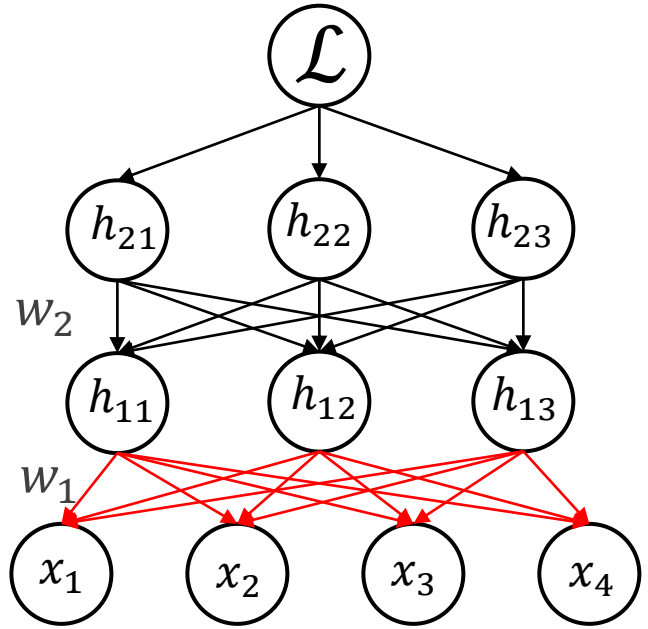
Backward propagation

$$\frac{d\mathcal{L}}{dh_2} = -(y^* - h_2)$$

$$\frac{d\mathcal{L}}{dw_2} = \frac{d\mathcal{L}}{dh_2}\frac{dh_2}{dw_2} = \frac{d\mathcal{L}}{dh_2} h_1 \sigma(w_2 h_1)\big(1 - \sigma(w_2 h_1)\big) = \frac{d\mathcal{L}}{dh_2} h_1 h_2 (1 - h_2)$$

$$\frac{d\mathcal{L}}{dh_1} = \frac{d\mathcal{L}}{dh_2}\frac{dh_2}{dh_1} = \frac{d\mathcal{L}}{dh_2} w_2 \sigma(w_2 h_1)\big(1 - \sigma(w_2 h_1)\big) = \frac{d\mathcal{L}}{dh_2} w_2 h_2 (1 - h_2)$$

→ $$\frac{d\mathcal{L}}{dw_1} = \frac{d\mathcal{L}}{dh_1}\frac{dh_1}{dw_1} = \frac{d\mathcal{L}}{dh_1} h_0 \sigma(w_1 h_0)\big(1 - \sigma(w_1 h_0)\big) = \frac{d\mathcal{L}}{dh_1} h_0 h_1 (1 - h_1)$$



Our lives without AutoDiff

```python
# -*- coding: utf-8 -*-
import numpy as np
import math

# Create random input and output data
x = np.linspace(-math.pi, math.pi, 2000)
y = np.sin(x)

# Randomly initialize weights
a = np.random.randn()
b = np.random.randn()
c = np.random.randn()
d = np.random.randn()

learning_rate = 1e-6
for t in range(2000):
    # Forward pass: compute predicted y
    # y = a + b x + c x^2 + d x^3
    y_pred = a + b * x + c * x ** 2 + d * x ** 3

    # Compute and print loss
    loss = np.square(y_pred - y).sum()
    if t % 100 == 99:
        print(t, loss)

    # Backprop to compute gradients of a, b, c, d with respect to loss
    grad_y_pred = 2.0 * (y_pred - y)
    grad_a = grad_y_pred.sum()
    grad_b = (grad_y_pred * x).sum()
    grad_c = (grad_y_pred * x ** 2).sum()
    grad_d = (grad_y_pred * x ** 3).sum()

    # Update weights
    a -= learning_rate * grad_a
    b -= learning_rate * grad_b
    c -= learning_rate * grad_c
    d -= learning_rate * grad_d

print(f'Result: y = {a} + {b} x + {c} x^2 + {d} x^3')
```

Our lives with AutoDiff (AutoGrad)

```python
class DynamicNet(torch.nn.Module):
    def __init__(self):
        """
        In the constructor we instantiate five parameters and assign them as members.
        """
        super().__init__()
        self.a = torch.nn.Parameter(torch.randn(()))
        self.b = torch.nn.Parameter(torch.randn(()))
        self.c = torch.nn.Parameter(torch.randn(()))
        self.d = torch.nn.Parameter(torch.randn(()))
        self.e = torch.nn.Parameter(torch.randn(()))

    def forward(self, x):
        """
        For the forward pass of the model, we randomly choose either 4, 5
        and reuse the e parameter to compute the contribution of these orders.

        Since each forward pass builds a dynamic computation graph, we can use normal
        Python control-flow operators like loops or conditional statements when
        defining the forward pass of the model.

        Here we also see that it is perfectly safe to reuse the same parameter many
        times when defining a computational graph.
        """
        y = self.a + self.b * x + self.c * x ** 2 + self.d * x ** 3
        for exp in range(4, random.randint(4, 6)):
            y = y + self.e * x ** exp
        return y
```

```python
# Create Tensors to hold input and outputs.
x = torch.linspace(-math.pi, math.pi, 2000)
y = torch.sin(x)

# Construct our model by instantiating the class defined above
model = DynamicNet()

# Construct our loss function and an Optimizer. Training this strange model with
# vanilla stochastic gradient descent is tough, so we use momentum
criterion = torch.nn.MSELoss(reduction='sum')
optimizer = torch.optim.SGD(model.parameters(), lr=1e-8, momentum=0.9)
for t in range(30000):
    # Forward pass: Compute predicted y by passing x to the model
    y_pred = model(x)

    # Compute and print loss
    loss = criterion(y_pred, y)
    if t % 2000 == 1999:
        print(t, loss.item())

    # Zero gradients, perform a backward pass, and update the weights.
    optimizer.zero_grad()
    loss.backward()
    optimizer.step()

print(f'Result: {model.string()}')
```

Summary

Forward and backward propagation.

Forward and reverse automatic differentiation.

Implementation of automatic differentiation.

Learning and reflection

Understanding Deep Learning: Chapter 4

Understanding Deep Learning: Chapter 5

Understanding Deep Learning: Chapter 6.1, 7.1-7.4

Next lecture

Lecture	Title
1	Intro and history of deep learning
3	Deep learning optimization I
5	Convolutional deep learning
7	Graph deep learning
9	Multi-modal deep learning
11	What doesn't work in deep learning
13	Q&A

Lecture	Title
2	AutoDiff
4	Deep learning optimization II
6	Attention-based deep learning
8	From supervised to unsupervised deep learning
10	Generative deep learning
12	Non-Euclidean deep learning
14	Deep learning for videos