Machine Learning

Lecture 7: Deep Learning I

Prof. Dr. Stephan Günnemann

Data Analytics and Machine Learning Group
Technische Universität München

Winter term 2020/2021

Pushing through pain to achieve what seems impossible is a more gratifying form of magic. "David Blaine"

Section 1

Introduction

Another look at Logistic Regression

linear decision boundaries, can only handle linearly seperable data

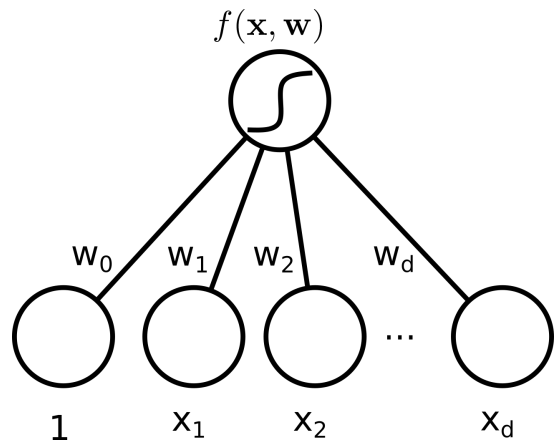
We had before:

$$y \mid \boldsymbol{x} \sim \text{Bernoulli}\Big(\sigma\big(\boldsymbol{w}^T\boldsymbol{x}\big)\Big)$$

$$\boldsymbol{w}^T\boldsymbol{x} := w_0 + w_1 x_1 + \ldots + w_D x_D$$

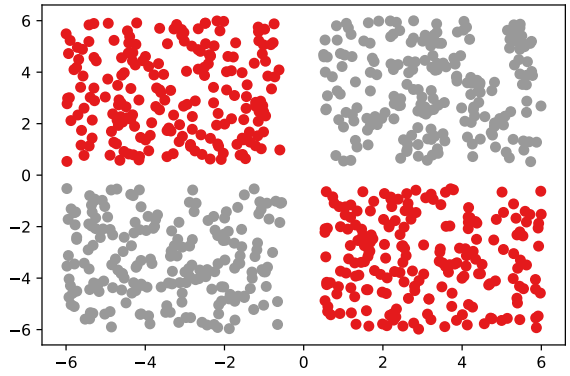
We can represent this graphically
(the first node $1 = x_0$ is for the bias term):

- each node is a (scalar) input
- multiply the input with the weight on the edge: $x_j w_j$
- compute weighted sum of incoming edges: $a_0 = \sum_{j=0}^{D} x_j w_j$ before applying act. fn.
- apply (activation) function: $p(y = 1 \mid \boldsymbol{x}) = \sigma(a_0) = f(\boldsymbol{x}, \boldsymbol{w})$



The XOR dataset

The XOR dataset is not linearly separable → Logistic Regression will fail since it learns a linear decision boundary



No hyperplane can seperate such data

How to handle non-linearity? Basis functions

We have input vectors $\boldsymbol{x}$ and associated targets y. We want to describe the underlying functional relation.

We can use the following simple model:

$$f(\boldsymbol{x}, \boldsymbol{w}) = \sigma\left(w_0 + \sum_{j=1}^{M-1} w_j \phi_j(\boldsymbol{x})\right) = \sigma(\boldsymbol{w}^\mathsf{T} \boldsymbol{\phi}(\boldsymbol{x})) \tag{1}$$

where

$\boldsymbol{\phi}$	basis function	—	many choices, can be nonlinear
w_0	bias	—	equivalent to defining $\phi_0 \equiv 1$; or adding constant 1 to every sample

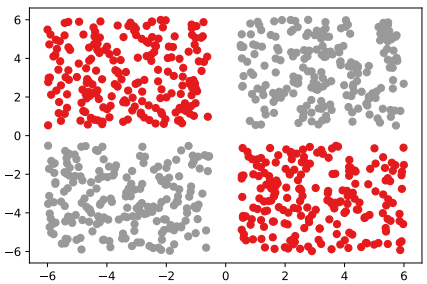
Remember we are linear in $\boldsymbol{w}$!

non linear in x

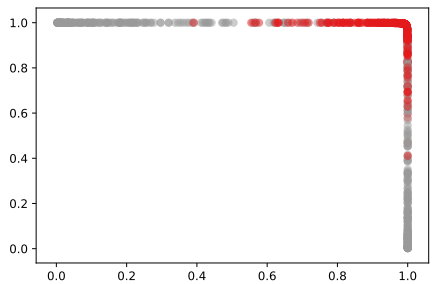
Example: Handling XOR dataset by custom basis functions

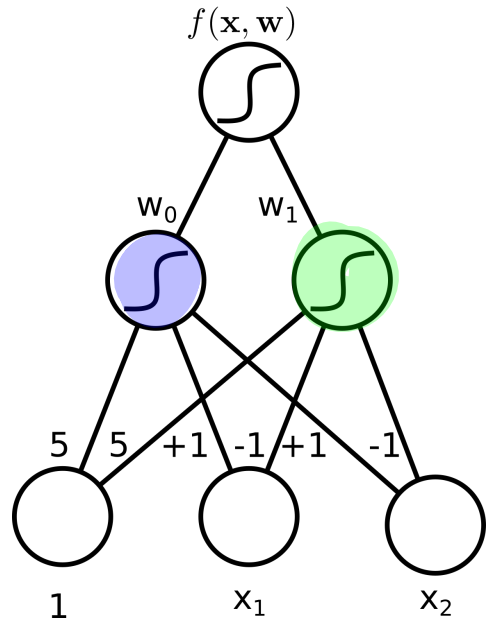
Apply a (nonlinear) transformation ϕ that maps samples to a space where they are linearly separable. For example:

$\mathbb{R}^2$ space



Transformed $\mathbb{R}^2$ space





Here we defined a custom basis function $\boldsymbol{\phi} : \mathbb{R}^3 \to \mathbb{R}^2$

$$\boldsymbol{\phi}(\boldsymbol{x}) = \boldsymbol{\phi}(1, x_1, x_2) = (\ \sigma(5 + x_1 + x_2)\ ,\ \sigma(5 - x_1 - x_2)\ )$$

basis transformation + classification step

weights correspond to basis fn

input

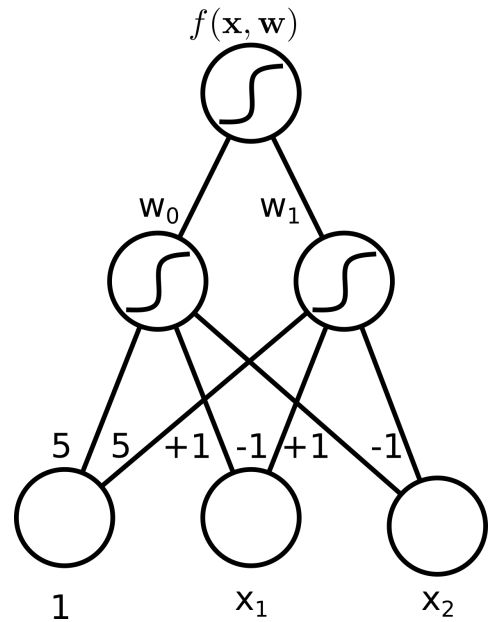
Overall function that is modeled:

$$f(\boldsymbol{x}, \boldsymbol{w}) = \sigma(\boldsymbol{w}^\mathsf{T} \boldsymbol{\phi}(\boldsymbol{x})) = \sigma_1 \left(\begin{bmatrix} w_0 & w_1 \end{bmatrix} \cdot \sigma_0 \left(\begin{bmatrix} 5 & 1 & 1 \\ 5 & -1 & -1 \end{bmatrix} \begin{bmatrix} 1 \\ x_1 \\ x_2 \end{bmatrix} \right) \right)$$

How to find the parameters $\boldsymbol{w}$?

Just train the model by minimizing a corresponding loss function.

Here: binary cross-entropy since binary classification problem.



$$\boldsymbol{w}^* = \arg\min_{\boldsymbol{w}} \sum_{n=1}^{N} - \left(y_n \log f(\boldsymbol{x}_n, \boldsymbol{w}) + (1 - y_n) \log \left(1 - f(\boldsymbol{x}_n, \boldsymbol{w}) \right) \right)$$

How to find the basis functions?

Different datasets require different transformations to become (almost) linearly separable.

"representation" "classification"

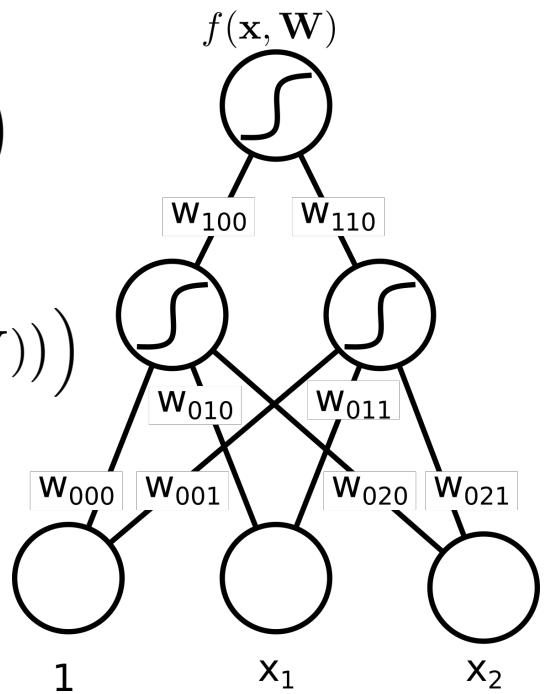
Idea: learn the basis functions and the weights of the logistic regression **jointly** from the data (end-to-end learning)

- Previously: only learning w_{100} and w_{110}
- Now: Learn all w_{ijk} where i=layer, j = input node, k = output node

neuron in layer; weight input

$$f(\boldsymbol{x}, \boldsymbol{W}) = \sigma_1\Big(\begin{bmatrix} w_{100} & w_{110} \end{bmatrix} \cdot \sigma_0\Big(\begin{bmatrix} w_{000} & w_{010} & w_{020} \\ w_{001} & w_{011} & w_{021} \end{bmatrix} \begin{bmatrix} 1 \\ x_1 \\ x_2 \end{bmatrix}\Big)\Big)$$

$$\boldsymbol{W}^* = \arg\min_{\boldsymbol{W}} \sum_{n=1}^{N} -\Big(y_n \log f(\boldsymbol{x}_n, \boldsymbol{W}) + (1-y_n) \log\big(1-f(\boldsymbol{x}_n, \boldsymbol{W})\big)\Big)$$

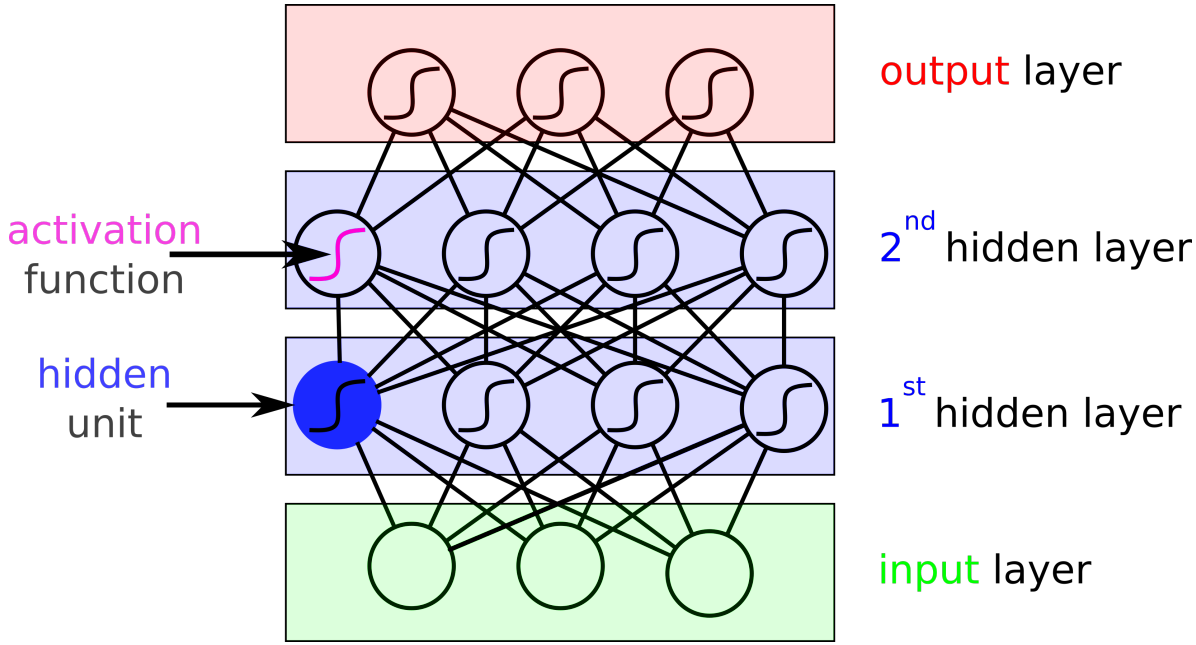


We get a simple Feed-Forward Neural Network (FFNN) with 1 hidden layer.

Note: σ_0 and σ_1 can be arbitrary activation functions.

Making the model more complicated

Each basis function can be a more complicated function of the feature vector $\boldsymbol{x}$ (a function of other basis functions rather than a function of $\boldsymbol{x}$).



By adding more hidden layers we get a "deep" neural network:

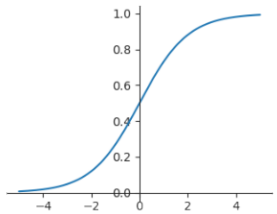
$$f(\boldsymbol{x}, \boldsymbol{W}) = \sigma_2\Big(\boldsymbol{W}_2^T \sigma_1\big(\boldsymbol{W}_1^\mathsf{T} \sigma_0(\boldsymbol{W}_0^T\, \boldsymbol{x})\big)\Big)$$

where $\boldsymbol{W} = \{\boldsymbol{W}_0, \boldsymbol{W}_1, \boldsymbol{W}_2\}$ are the weights to be learned.
Above architecture is called a Multi-layered Perceptron (MLP) = fully-connected (feed-forward) Neural Network

Activation functions

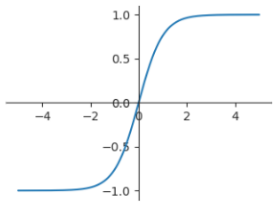
Sigmoid

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



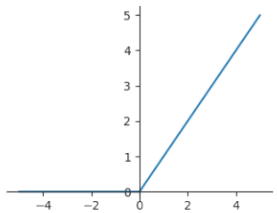
tanh

$\tanh(x)$



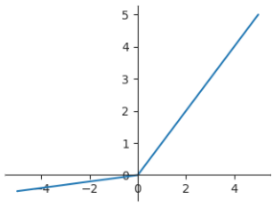
ReLU

$\max(0, x)$



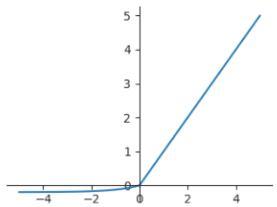
Leaky ReLU

$\max(0.1x, x)$



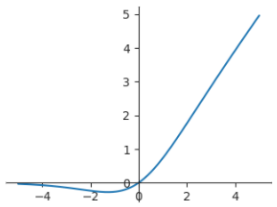
ELU

$$\begin{cases} x & x \geq 0 \\ \alpha(e^x - 1) & x < 0 \end{cases}$$



Swish

$x \cdot \sigma(x)$



Most activiation functions are applied *element-wise* when given a *multi-dimensional* input.

Softmax activation is a notable exception!

In principle, you can choose seperate act fn for each neuron!

Why use nonlinear activation functions?

they are the core of NN

Multiple linear layers: i.e no activation

$$\begin{aligned} f(\boldsymbol{x}, \boldsymbol{W}) &= \boldsymbol{W}_L \quad (\boldsymbol{W}_{L-1} \quad (\ldots \quad (\boldsymbol{W}_0 \boldsymbol{x}) \ldots)) \\ &= (\boldsymbol{W}_L \boldsymbol{W}_{L-1} \ldots \boldsymbol{W}_1 \boldsymbol{W}_0) \boldsymbol{x} \\ &= \boldsymbol{W}' \boldsymbol{x} \end{aligned}$$

results in a linear transformation!

Multiple nonlinear layers:

$$f(\boldsymbol{x}, \boldsymbol{W}) = \boldsymbol{W}_L \, \sigma_k(\boldsymbol{W}_{L-1} \, \sigma_{L-1}(\ldots \, \sigma_0(\boldsymbol{W}_0 \boldsymbol{x}) \ldots))$$

For non-linear functions we can not (in general) simplify it.

Neural networks are universal approximators

Universal approximation theorem theoretically

An MLP with a linear output layer and one hidden layer can approximate any continuous function defined over a closed and bounded subset of $\mathbb{R}^D$, under mild assumptions on the activation function ('squashing' activation functions; e.g. sigmoid) and given the number of hidden units is large enough.

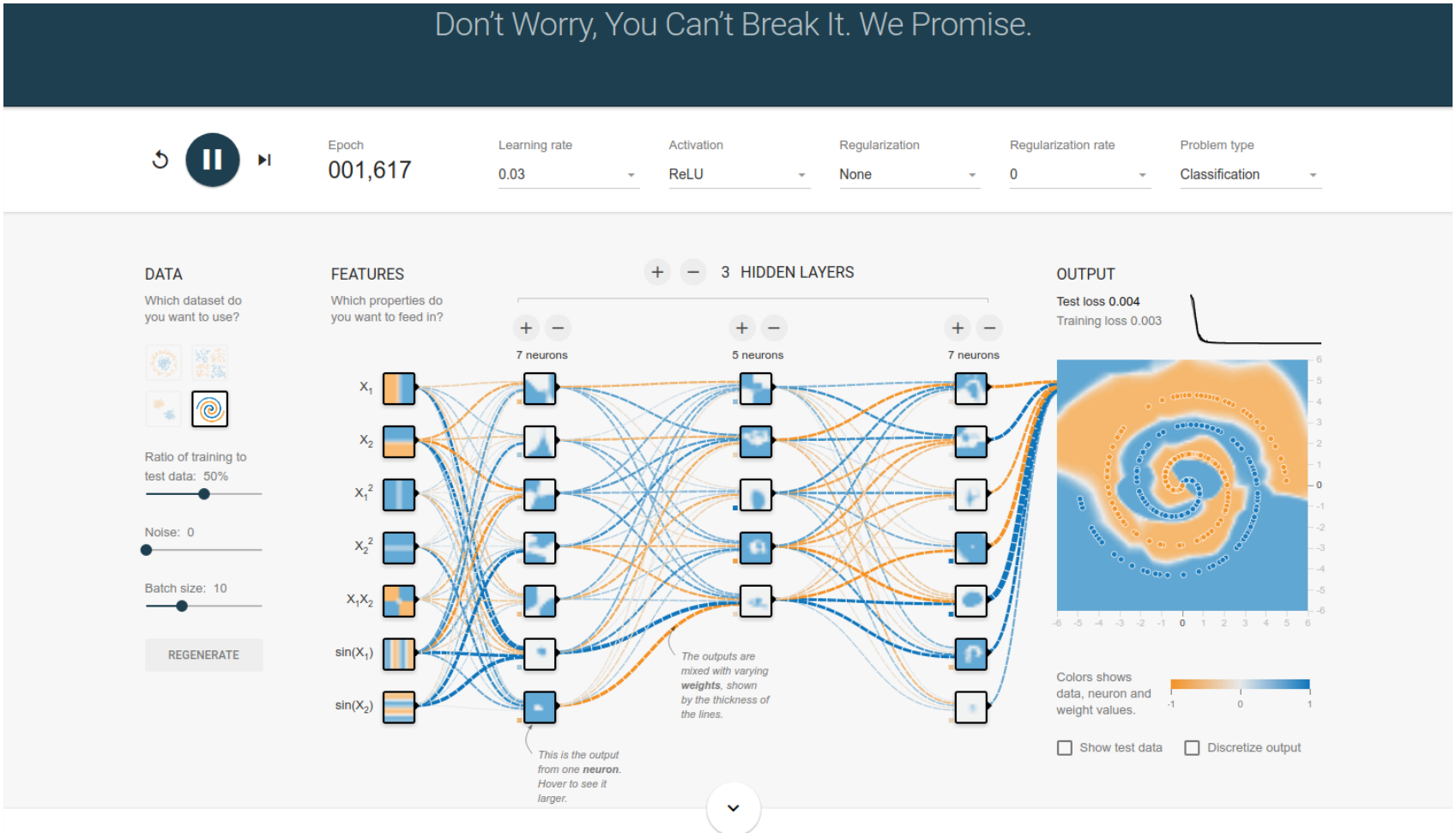
[Cybenko 1989; Funahashi 1989; Hornik et al 1989, 1991; Hartman et al 1990].

Also in the discrete case: a neural network can approximate any function from a discrete space to another.

Good news: Regardless of the function we want to learn, there exists an MLP that can approximate that function arbitrarily well.

Bad news: We have no idea how to find this MLP that provides the best approximation.

Demo: http://playground.tensorflow.org



Multiple hidden layers

According to the universal approximation theorem, a feed-forward network with 1 hidden layer can represent any function.

Why do we add more layers?

Theoretical reason:

- For some families of functions, if we use *a few* layers we would need a large number of hidden units (and therefore parameters). But we can get the same representation power by adding *more layers*, fewer hidden units, and fewer parameters.

Practical reason:

- Deeper networks (with some additional tricks) often train faster and generalize better.

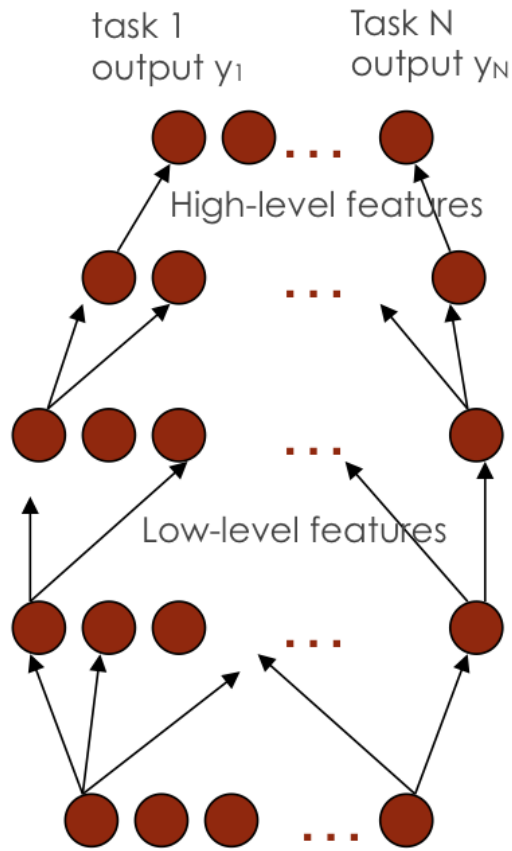
In practice, networks are not very wide but rather deep

i.e removing one layer may lead to explosion in # parameters

Functions that can be compactly represented with k layers may require exponentially many hidden units when using only $k-1$ layers.

Deep network can learn a hierarchy of representations.

Different high-level features share lower-level features.



from: *Understanding and Improving Deep Learning Algorithms*, Yoshua Bengio, ML Google Distinguished Lecture, 2010

NN are very complicated fn : several layers having linear transformation followed by non-linear activation fn ... stacking all together.

- We train them through loss fn in end-to-end fashion.

Section 2

Beyond binary classification

Loss function

Neural networks can be used for various prediction tasks.

Different tasks require changing

1. the activation function in the *final* layer
2. the loss function

For supervised learning common choices are

Prediction target	$p(\boldsymbol{y} \mid \boldsymbol{x})$	**Final layer**	**Loss function**
Binary	Bernoulli	Sigmoid	Binary cross entropy
Discrete	Categorical	Softmax	Cross entropy
Continuous	Gaussian	Identity	Squared error

e.g. regression

no activation

Example 1: Binary classification

- Data: $\{\boldsymbol{x}_n, y_n\}_{n=1}^N$, where $y_n \in \{0, 1\}$.
- Activation in the final layer: sigmoid

$$f(\boldsymbol{x}, \boldsymbol{W}) = \frac{1}{1 + \exp(-a)}$$

└ logits : output of last layer before act.

- Conditional distribution

$$p(y \mid \boldsymbol{x}) = \text{Bernoulli}(y \mid f(\boldsymbol{x}, \boldsymbol{W}))$$

- Loss function: binary cross entropy

$$\begin{aligned} E(\boldsymbol{W}) &= -\sum_{n=1}^{N} \log p(y_n \mid \boldsymbol{x}_n) \\ &= -\sum_{n=1}^{N} \Big(y_n \log f(\boldsymbol{x}_n, \boldsymbol{W}) + (1 - y_n) \log \left(1 - f(\boldsymbol{x}_n, \boldsymbol{W})\right) \Big) \end{aligned}$$

Example 2: Multi-class classification

- Data: $\{\boldsymbol{x}_n, \boldsymbol{y}_n\}_{n=1}^N$, where $\boldsymbol{y}_n \in \{0,1\}^K$ (one-hot notation).
- Activation in the final layer: softmax

$$f_k(\boldsymbol{x}, \boldsymbol{W}) = \frac{\exp(a_k)}{\sum_{j=1}^K \exp(a_j)}$$

↳ is not scalar anymore but a vector of logits

- Conditional distribution

$$p(\boldsymbol{y} \mid \boldsymbol{x}) = \text{Categorical}(\boldsymbol{y} \mid f(\boldsymbol{x}, \boldsymbol{W}))$$

- Loss function: categorical cross entropy

$$\begin{aligned} E(\boldsymbol{W}) &= -\sum_{n=1}^N \log p(\boldsymbol{y}_n \mid \boldsymbol{x}_n) \\ &= -\sum_{n=1}^N \sum_{k=1}^K y_{nk} \log f_k(\boldsymbol{x}_n, \boldsymbol{W}) \end{aligned}$$

Example 3: Single-output regression

- Data: $\{\boldsymbol{x}_n, y_n\}_{n=1}^N$, where $y_n \in \mathbb{R}$.
- Activation in the final layer: identity (no activation)

$$f(\boldsymbol{x}, \boldsymbol{W}) = a$$

- Conditional distribution: Gaussian

$$p(y \mid \boldsymbol{x}) = \mathcal{N}(y \mid f(\boldsymbol{x}, \boldsymbol{W}), 1)$$

- Loss function: squared error (a.k.a. Gaussian cross-entropy)

$$\begin{aligned} E(\boldsymbol{W}) &= -\sum_{n=1}^{N} \log p(y_n \mid \boldsymbol{x}_n) \\ &= \sum_{n=1}^{N} (y_n - f(\boldsymbol{x}_n, \boldsymbol{W}))^2 + \text{const.} \end{aligned}$$

Unsupervised deep learning

So far we have only considered neural networks for supervised learning. Unsupervised deep learning is also a very active field of research, with models such as

- Autoencoder (covered later in this course)
- Variational autoencoder (covered in our MLGS lecture)
- Generative adversarial networks (GAN; covered in our MLGS lecture)
- Unsupervised representation learning (e.g. word or node embeddings; covered in our MLGS lecture)

Choosing the loss

e.g 0,1 are not very suitable though

We are free to choose any loss that provides a useful gradient for training.

This choice includes both the function and which values to use and compare to. Many model types mainly differ in their choice of loss.

Example loss functions:

- Cross entropy: For supervised classification
- Mean squared error (MSE)
- Mean absolute error (MAE): Useful if we have outliers, but gradient is independent of the distance from the optimum (advanced optimizers handle this surprisingly well, though).
- Huber loss, LogCosh: MSE at close distances, MAE far away. Combine benefits of MSE and MAE.
- Wasserstein (earth mover's) distance, KL-divergence: For continuous distributions

Section 3

Parameter learning

Minimizing the loss function

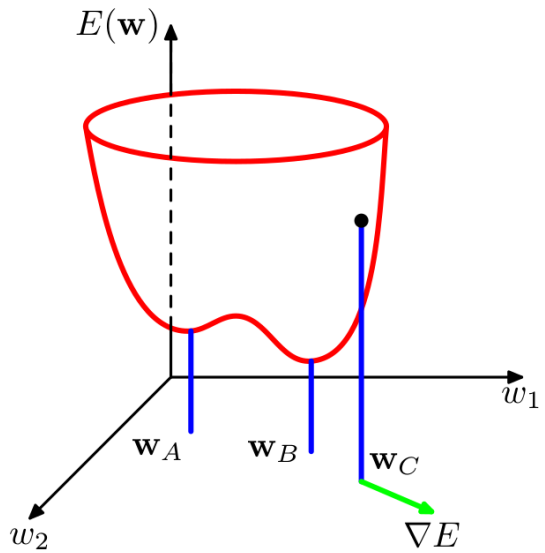
In practice $E(\boldsymbol{W})$ is often non-convex $\Longrightarrow$ optimization is tricky

- a local minimum is not necessarily a global minimum.
- potentially there exist several local minima, many of which can be equivalent (see next tutorial session)
- often it is not possible to find a global minimum nor is it useful.

We may find a few local minima, and pick the one with higher performance on a validation set.

Default approach:
find a local minimum by
using gradient descent

$$\boldsymbol{W}^{(new)} = \boldsymbol{W}^{(old)} - \tau \nabla_{\boldsymbol{W}} E(\boldsymbol{W}^{(old)})$$



How can we compute the gradient?

1) By hand: manually working out $\nabla_{\boldsymbol{W}} E$ and coding it is tricky and cumbersome (furthermore: see point 3).

2) Numeric: Can be done as

$$\frac{\partial E_n}{\partial w_{ij}} = \frac{E_n(w_{ij} + \epsilon) - E_n(w_{ij})}{\epsilon} + \mathcal{O}(\epsilon)$$

away from w_{ij}

- Each evaluation of the above equation roughly requires $\mathcal{O}(|\boldsymbol{W}|)$ operations, where $|\boldsymbol{W}|$ is the dimensionality of weight space.
- The evaluation has to be done for each parameter independently. Therefore computing $\nabla_{\boldsymbol{W}} E$ requires $\mathcal{O}(|\boldsymbol{W}|^2)$ operations!

evaluating the whole NN for just one W

this extremely expensive & might not be very stable

3) Symbolic differentiation: Automates essentially how you would compute the gradient function by hand.
But: "Writing down" the general expression for the gradient of every parameter is very expensive

- potentially exponentially many different cases (e.g. when having multiple layers with ReLUs)
- many terms reappear in the gradient computation for *different* parameters (since the function f is hierarchically constructed); these terms could be re-used to make computation faster; however symbolic differentiation does not exploit this insight

4) Automatic differentiation: e.g. **backpropagation** for neural networks

- Evaluate $\nabla_{\boldsymbol{W}} E(\boldsymbol{W})$ at the current point $\boldsymbol{W}$
- Evaluation in $\mathcal{O}(|\boldsymbol{W}|)$ (every "neuron" is visited only twice)

There is no need to get full grad of the whole network

forward backward

Backpropagation: Toy example

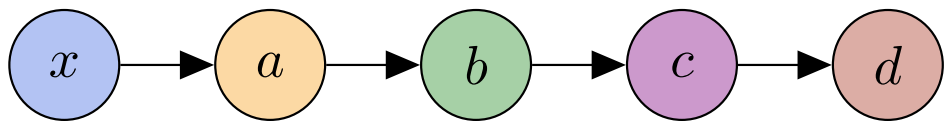
$$f(x) = \frac{2}{\sin(\exp(-x))} \qquad f(x) = d(c(b(a(x))))$$

Modules:

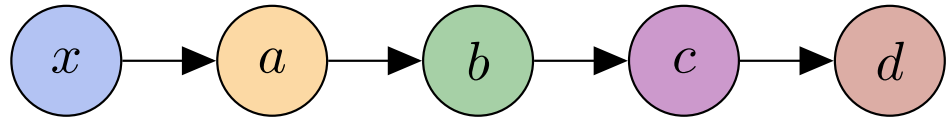
functions

$$a(x) = -x$$
$$b(a) = \exp(a)$$
$$c(b) = \sin b$$
$$d(c) = \frac{2}{c}$$

Computational graph:



$$f(x) = \frac{2}{\sin(\exp(-x))}$$



Modules:

$$a(x) = -x$$
$$b(a) = \exp(a)$$
$$c(b) = \sin b$$
$$d(c) = \frac{2}{c}$$

Chain rule:

$$\frac{\partial f}{\partial x} = \frac{\partial d}{\partial c}\frac{\partial c}{\partial b}\frac{\partial b}{\partial a}\frac{\partial a}{\partial x}$$

$\frac{\partial f}{\partial x}$ is the global derivative

$\frac{\partial d}{\partial c}, \frac{\partial c}{\partial b}, \frac{\partial b}{\partial a}, \frac{\partial a}{\partial x}$ are the local derivatives

i.e Jacobians

Backpropagation in a nutshell

↳ efficient way to calculate the gradient

- Write your function as a composition of modules

  What the modules are is up to you

- Work out the local derivative of each module symbolically

- Do a forward pass for a given input x

  i.e. compute the function $f(x)$ and remember the intermediate values (caching)

- ~~*~~ Compute the local derivatives for x

- Obtain the global derivative by multiplying the local derivatives

Work out the local derivatives

Compute the local derivative of each module symbolically (independently)

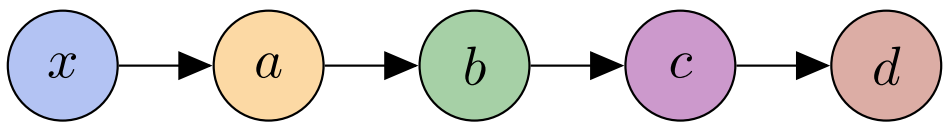
$$a(x) = -x \qquad \frac{\partial a}{\partial x} = -1$$

$$b(a) = \exp(a) \qquad \frac{\partial b}{\partial a} = \exp(a)$$

$$c(b) = \sin b \qquad \frac{\partial c}{\partial b} = \cos b$$

$$d(c) = \frac{2}{c} \qquad \frac{\partial d}{\partial c} = -\frac{2}{c^2}$$

Computational graph:



Forward pass

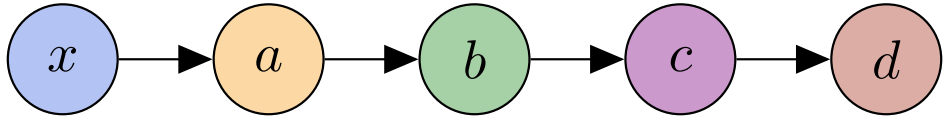
weights

We want to compute the derivative $\frac{\partial f}{\partial x}$ at $x = -4.499$

In the forward pass, we compute f by following the computational graph and cache (memorize) the intermediate values of a, b, c, d

$$
\begin{aligned}
a &= -x = 4.499 \\
b &= \exp(a) = 90 \\
c &= \sin b = 1 \\
d &= \frac{2}{c} = 2
\end{aligned}
$$

Computational graph:



Backward pass

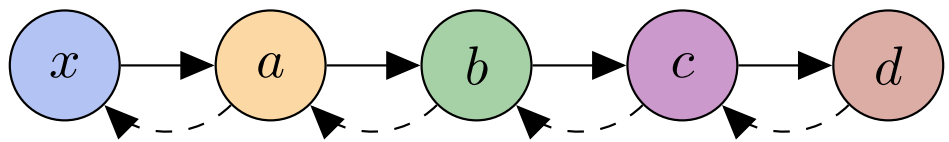
In the backward pass, we use the cached values of a, b, c, d to compute the local derivatives $\frac{\partial d}{\partial c}, \frac{\partial c}{\partial b}, \frac{\partial b}{\partial a}, \frac{\partial a}{\partial x}$

$$\frac{\partial a}{\partial x} = -1 \qquad \frac{\partial b}{\partial a} = \exp(a) = \exp(4.499) = 90$$

$$\frac{\partial c}{\partial b} = \cos b = \cos 90 = 0 \qquad \frac{\partial d}{\partial c} = -\frac{2}{c^2} = -\frac{2}{2^2} = -\frac{1}{2}$$

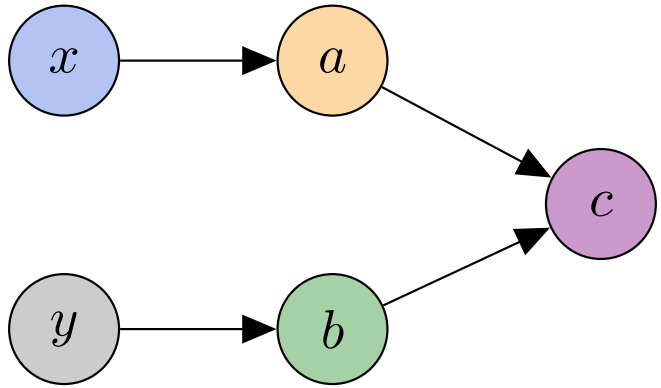
We obtain the global derivative by multiplying the local derivatives

$$\frac{\partial f}{\partial x} = \frac{\partial d}{\partial c}\frac{\partial c}{\partial b}\frac{\partial b}{\partial a}\frac{\partial a}{\partial x} = -\frac{1}{2} \cdot 0 \cdot 90 \cdot -1 = 0$$



Multiple inputs

What if a function has multiple inputs?

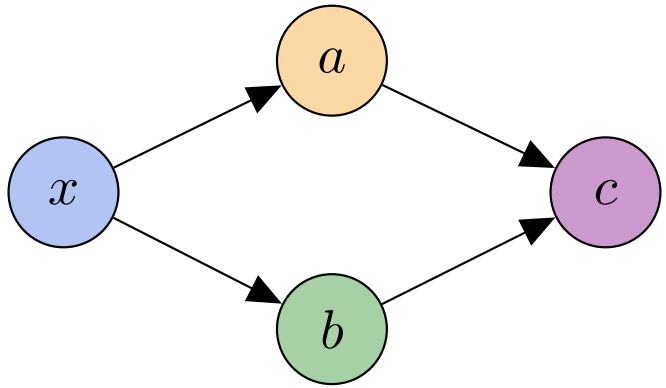


We compute the derivative along each of the paths

$$\frac{\partial c}{\partial x} = \frac{\partial c}{\partial a}\frac{\partial a}{\partial x} \qquad \frac{\partial c}{\partial y} = \frac{\partial c}{\partial b}\frac{\partial b}{\partial y}$$

Multiple paths

What if a computational graph contains multiple paths from x to c?

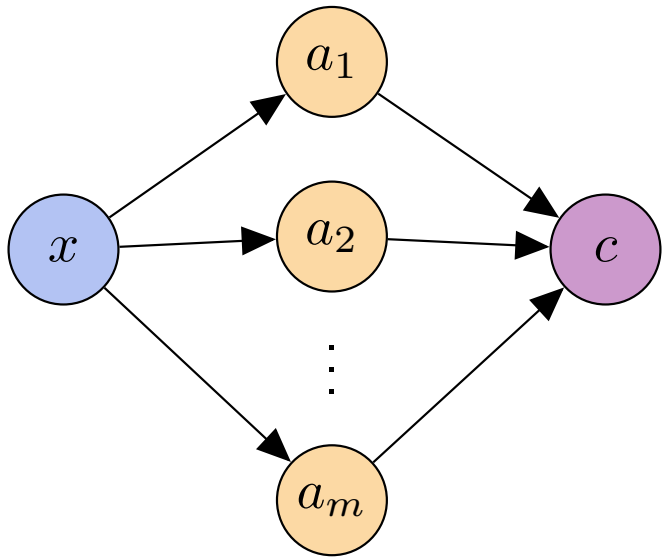


We need to sum the derivatives along each of the paths

$$\frac{\partial c}{\partial x} = \frac{\partial c}{\partial a}\frac{\partial a}{\partial x} + \frac{\partial c}{\partial b}\frac{\partial b}{\partial x}$$

Multivariate chain rule

The generalization to an arbitrary number of paths is given by the multivariate chain rule



We need to sum the derivatives along all the paths

$$\frac{\partial c}{\partial x} = \sum_{i=1}^{m} \frac{\partial c}{\partial a_i} \frac{\partial a_i}{\partial x}$$

Jacobian and the gradient

Consider a function $f : \mathbb{R}^n \to \mathbb{R}^m$ and let $\boldsymbol{a} = f(\boldsymbol{x})$.

The Jacobian is an $m \times n$ matrix of partial derivatives

$$\frac{\partial \boldsymbol{a}}{\partial \boldsymbol{x}} = \begin{bmatrix} \frac{\partial a_1}{\partial x_1} & \cdots & \frac{\partial a_1}{\partial x_n} \\ \vdots & \ddots & \vdots \\ \frac{\partial a_m}{\partial x_1} & \cdots & \frac{\partial a_m}{\partial x_n} \end{bmatrix} \in \mathbb{R}^{m \times n}$$

loss

Let $g : \mathbb{R}^m \to \mathbb{R}$ and let $c = g(\boldsymbol{a})$.

The gradient $\nabla_{\boldsymbol{a}} c \in \mathbb{R}^m$ is the transpose of the Jacobian $\frac{\partial c}{\partial \boldsymbol{a}} \in \mathbb{R}^{1 \times m}$

loss

$$\nabla_{\boldsymbol{a}} c = \left(\frac{\partial c}{\partial \boldsymbol{a}} \right)^T = \begin{bmatrix} \frac{\partial c}{\partial a_1} & \cdots & \frac{\partial c}{\partial a_m} \end{bmatrix}^T$$ (vector)

* Note that the gradient $\nabla_{\boldsymbol{a}} c$ has the same shape as $\boldsymbol{a}$. — input

---

We are using the so-called numerator notation. Some other resources use a different (denominator) notation — be careful when using other books or slides.

Chain rule in matrix form

Multivariate

We can compactly represent the chain rule using Jacobian matrices

$$\frac{\partial c}{\partial x_j} = \sum_{i=1}^{m} \frac{\partial c}{\partial a_i} \frac{\partial a_i}{\partial x_j}$$

operating on such scalar forms is more tedious

We can write this in matrix form

$$\frac{\partial c}{\partial \boldsymbol{x}} = \frac{\partial c}{\partial \boldsymbol{a}} \frac{\partial \boldsymbol{a}}{\partial \boldsymbol{x}}$$

where $\left[\frac{\partial \boldsymbol{a}}{\partial \boldsymbol{x}}\right]_{ij} = \frac{\partial a_i}{\partial x_j}$

Jacobian

or equivalently

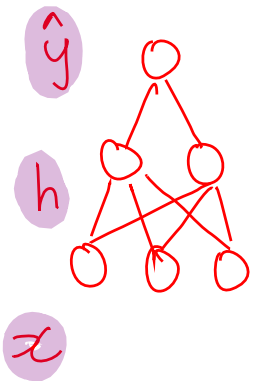
$$\nabla_{\boldsymbol{x}} c = \left(\frac{\partial \boldsymbol{a}}{\partial \boldsymbol{x}}\right)^T \nabla_{\boldsymbol{a}} c$$

matrix vector

Computational graph of a neural network

Consider a simple regression problem: $\boldsymbol{x} \in \mathbb{R}^D, y \in \mathbb{R}$, square error loss.

Generate predictions with a feed-forward NN



$$\boldsymbol{a} = \boldsymbol{W}\boldsymbol{x} + \boldsymbol{b}$$
$$\boldsymbol{h} = \sigma(\boldsymbol{a})$$
$$\hat{y} = \boldsymbol{V}\boldsymbol{h} + c$$
$$E = (\hat{y} - y)^2$$

linear transformation

applying non linearity

hidden layer

affine layer

loss

In order to optimize $\boldsymbol{W}$ with gradient descent, we need to compute $\frac{\partial E}{\partial \boldsymbol{W}}$.

thinking of them as

staking several modules

Computational graph of a neural network

Consider a simple regression problem: $\boldsymbol{x} \in \mathbb{R}^D, y \in \mathbb{R}$, square error loss.

Generate predictions with a feed-forward NN

$$\begin{aligned}
\boldsymbol{a} &= \boldsymbol{W}\boldsymbol{x} + \boldsymbol{b} \\
\boldsymbol{h} &= \sigma(\boldsymbol{a}) \\
\hat{y} &= \boldsymbol{V}\boldsymbol{h} + c \\
E &= (\hat{y} - y)^2
\end{aligned}$$

In order to optimize $\boldsymbol{W}$ with gradient descent, we need to compute $\frac{\partial E}{\partial \boldsymbol{W}}$.

Does chain rule even make sense in this scenario?

local derivatives

global derivative

$$\frac{\partial E}{\partial \boldsymbol{W}} = \frac{\partial E}{\partial \hat{y}} \frac{\partial \hat{y}}{\partial \boldsymbol{h}} \frac{\partial \boldsymbol{h}}{\partial \boldsymbol{a}} \frac{\partial \boldsymbol{a}}{\partial \boldsymbol{W}}$$

What does it mean to take a derivative of a vector $\boldsymbol{a}$ w.r.t. a matrix $\boldsymbol{W}$?

Matrix calculus

What is the derivative of a vector $\boldsymbol{a} \in \mathbb{R}^H$ w.r.t. a matrix $\boldsymbol{W} \in \mathbb{R}^{H \times D}$?

output

input

how can input affect output

		Input is a ...		
		scalar	vector	matrix
Output is a...	scalar	scalar	vector	matrix
	vector	vector	matrix	3-way tensor
	matrix	matrix	3-way tensor	4-way tensor

That means, $\frac{\partial \boldsymbol{a}}{\partial \boldsymbol{W}}$ is a 3-way tensor of shape $(H \times H \times D)$ where

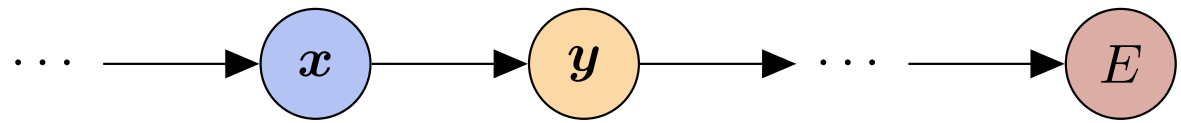
$$\left(\frac{\partial \boldsymbol{a}}{\partial \boldsymbol{W}}\right)_{ijk} = \frac{\partial a_i}{W_{jk}}$$

Accumulating the gradient

The local derivatives (i.e. Jacobians) such as $\frac{\partial \boldsymbol{a}}{\partial \boldsymbol{W}}$ are large multidimensional tensors.

Materializing (computing) them is expensive and unnecessary.

In reality, we only care about the global derivative!



Assume that we know $\frac{\partial E}{\partial \boldsymbol{y}}$ and want to compute $\frac{\partial E}{\partial \boldsymbol{x}}$.

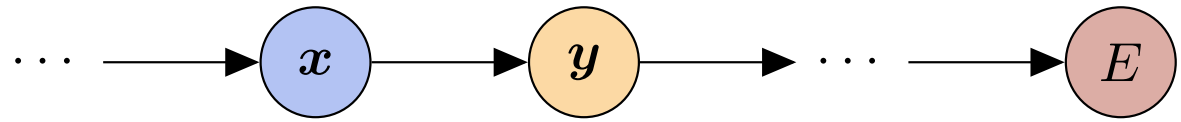
Most of the time, the Jacobian-vector product

$$\frac{\partial E}{\partial \boldsymbol{x}} = \frac{\partial E}{\partial \boldsymbol{y}} \frac{\partial \boldsymbol{y}}{\partial \boldsymbol{x}}$$

can be computed efficiently without materializing the Jacobian $\frac{\partial \boldsymbol{y}}{\partial \boldsymbol{x}}$!

Accumulating the gradient: Implementation

Backpropagation is typically implemented using the following abstraction.



Each module in the computational graph defines two functions

`forward`$(\boldsymbol{x})$: given input $\boldsymbol{x}$, compute output $\boldsymbol{y}$

`backward`$\left(\frac{\partial E}{\partial \boldsymbol{y}}\right)$: given the incoming global derivative $\frac{\partial E}{\partial \boldsymbol{y}}$

compute the product $\frac{\partial E}{\partial \boldsymbol{x}} = \frac{\partial E}{\partial \boldsymbol{y}} \frac{\partial \boldsymbol{y}}{\partial \boldsymbol{x}}$

Then, $\frac{\partial E}{\partial \boldsymbol{x}}$ is passed as input to the parent node in the computational graph, etc.

Example: Affine layer

Back to our problem of finding

$$\frac{\partial E}{\partial \boldsymbol{W}} = \frac{\partial E}{\partial \hat{y}} \frac{\partial \hat{y}}{\partial \boldsymbol{h}} \frac{\partial \boldsymbol{h}}{\partial \boldsymbol{a}} \frac{\partial \boldsymbol{a}}{\partial \boldsymbol{W}}$$

An affine layer is defined as

$$\boldsymbol{a} = \boldsymbol{W}\boldsymbol{x} + \boldsymbol{b}$$

where $\boldsymbol{x} \in \mathbb{R}^D, \boldsymbol{W} \in \mathbb{R}^{H \times D}, \boldsymbol{b} \in \mathbb{R}^H, \boldsymbol{a} \in \mathbb{R}^H$

`forward`$(\boldsymbol{W}, \boldsymbol{x}, \boldsymbol{b})$: compute $\boldsymbol{W}\boldsymbol{x} + \boldsymbol{b}$

`backward`$\left(\frac{\partial E}{\partial \boldsymbol{a}}\right)$: compute

global grad wrt point where we are in

$$\frac{\partial E}{\partial \boldsymbol{a}} \frac{\partial \boldsymbol{a}}{\partial \boldsymbol{W}}, \qquad \frac{\partial E}{\partial \boldsymbol{a}} \frac{\partial \boldsymbol{a}}{\partial \boldsymbol{x}}, \qquad \frac{\partial E}{\partial \boldsymbol{a}} \frac{\partial \boldsymbol{a}}{\partial \boldsymbol{b}}$$

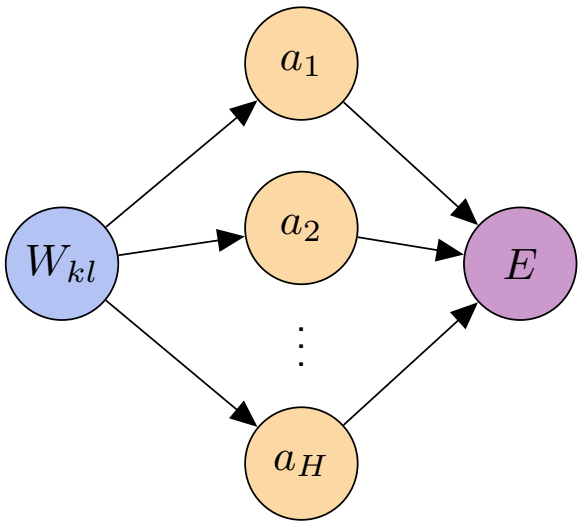
How can we implement this efficiently?

Remember in backprop, we are not actually computing all these local gradients. We calc the accumulated global derivative up to a specific point.

Step 1: work out the scalar derivative

Understand the flow!

First, let's find $\frac{\partial E}{\partial W_{kl}}$ for some k, l



$$\frac{\partial E}{\partial \boldsymbol{a}} \frac{\partial \boldsymbol{a}}{\partial W_{kl}} = \sum_i \frac{\partial E}{\partial a_i} \frac{\partial a_i}{\partial W_{kl}}$$

$$= \sum_i \frac{\partial E}{\partial a_i} \frac{\partial (\boldsymbol{W}\boldsymbol{x} + \boldsymbol{b})_i}{\partial W_{kl}}$$

$$= \sum_i \frac{\partial E}{\partial a_i} \frac{\partial (\boldsymbol{W}_{i:}\boldsymbol{x})}{\partial W_{kl}}$$

$$= \sum_i \frac{\partial E}{\partial a_i} \frac{\partial (\sum_j W_{ij} x_j)}{\partial W_{kl}}$$

$$= \sum_i \frac{\partial E}{\partial a_i} \frac{\partial (\sum_j W_{ij} x_j)}{\partial W_{kl}}$$

$$= \sum_i \sum_j \frac{\partial E}{\partial a_i} \frac{\partial W_{ij} x_j}{\partial W_{kl}} = \frac{\partial E}{\partial a_k} x_l$$

Step 2: backward pass in matrix form

For each element W_{kl} we have

$$\frac{\partial E}{\partial W_{kl}} = \frac{\partial E}{\partial a_k} x_l$$

We can compute $\frac{\partial E}{\partial W_{kl}}$ for all elements k, l in matrix form

$$\begin{bmatrix} x_1 & \cdots & x_D \end{bmatrix}^\top \times \begin{bmatrix} \frac{\partial E}{\partial a_1} & \cdots & \frac{\partial E}{\partial a_H} \end{bmatrix}$$

Jacobian

$$= \begin{bmatrix} \frac{\partial E}{\partial W_{11}} & \cdots & \frac{\partial E}{\partial W_{1H}} \\ \vdots & \ddots & \vdots \\ \frac{\partial E}{\partial W_{D1}} & \cdots & \frac{\partial E}{\partial W_{DH}} \end{bmatrix}$$

Or more compactly

incoming global derivative from next layer / module

$$\frac{\partial E}{\partial \boldsymbol{W}} = \boldsymbol{x} \frac{\partial E}{\partial \boldsymbol{a}}$$

* Matrix operations are much more efficient than `for`-loops thanks to vectorization (especially on GPUs)

No materialization of Jacobian

both x & $\frac{\partial}{\partial a} E$ are vectors

Example: Affine layer

An affine layer is defined as

$$\boldsymbol{a} = \boldsymbol{W}\boldsymbol{x} + \boldsymbol{b}$$

`forward`$(\boldsymbol{W}, \boldsymbol{x}, \boldsymbol{b})$: compute $\boldsymbol{W}\boldsymbol{x} + \boldsymbol{b}$

`backward`$\left(\frac{\partial E}{\partial \boldsymbol{a}}\right)$: compute

Backprop step is always simple & efficient

$$\frac{\partial E}{\partial \boldsymbol{W}} = \frac{\partial E}{\partial \boldsymbol{a}} \frac{\partial \boldsymbol{a}}{\partial \boldsymbol{W}} = \boldsymbol{x} \frac{\partial E}{\partial \boldsymbol{a}}$$
$$\frac{\partial E}{\partial \boldsymbol{x}} = \frac{\partial E}{\partial \boldsymbol{a}} \frac{\partial \boldsymbol{a}}{\partial \boldsymbol{x}} = \frac{\partial E}{\partial \boldsymbol{a}} \boldsymbol{W}$$
$$\frac{\partial E}{\partial \boldsymbol{b}} = \frac{\partial E}{\partial \boldsymbol{a}} \frac{\partial \boldsymbol{a}}{\partial \boldsymbol{b}} = \frac{\partial E}{\partial \boldsymbol{a}}$$

The derivatives for $\boldsymbol{x}$ and $\boldsymbol{b}$ are obtained similarly to $\boldsymbol{W}$.

Backpropagation: Summary

- Define the computation as a composition of modules
- The `forward` function computes the output of the module
- The `backward` function accumulates the total gradient
- We can implement `backward` without materializing the Jacobian
- Backpropagation walks backward through the computational graph, accumulating the product of gradients

walks

twice

→ no full grad is needed, just grad at current point

→ no materialization of Jacobian i.e faster implementation

- Basis transformation shall be learnable
- Last layer reflects our task

Reading material

- Goodfellow, Deep Learning: Chapter 6

Acknowledgements

- Some slides based on an earlier version by Patrick van der Smagt
- Backpropagation slides are based on the materials from `mlvu.github.io` by Peter Bloem