Neural networks

Motivation:

→ we want a powerful model which can describe arbitrary functional dependencies

↳ usually translates into large number of tunable / trainable parameters

↳ but has to be computationally efficient (evaluation of prediction $f_w(x)$ and gradients $\nabla_w f_w(x)$)

↳ ~~[illegible]~~ for training (next section)

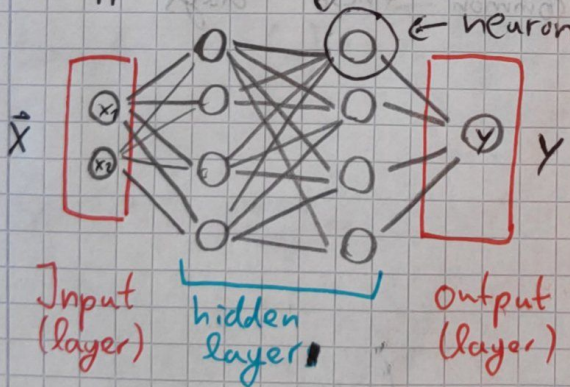
Idea:

→ stack many simple operations on top of each other!

Fully connected dense neural networks – dense layer

↳ prototype of neural networks

↳ appear in many architectures (almost all)



hidden layer: linear operation + non-linear operation

$$\vec{z} = W \cdot \vec{x} + \vec{b} \quad , \quad y = \sigma(z)$$

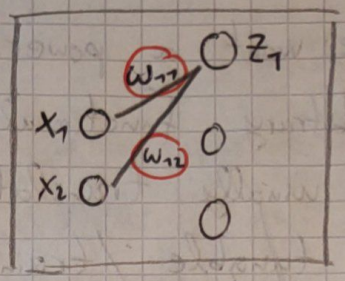
W: weight matrix of tunable parameters

σ: non-linear function

weight matrix $W = \begin{pmatrix} w_{11} & w_{12} & \cdots \\ w_{21} & w_{22} & \\ \vdots & & w_{MN} \end{pmatrix}$ (N rows, M columns)

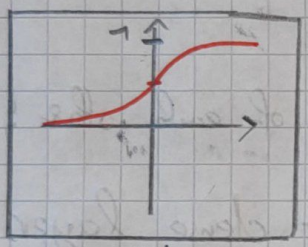
N: dimension of input $\vec{x}$

M: number of units / neurons



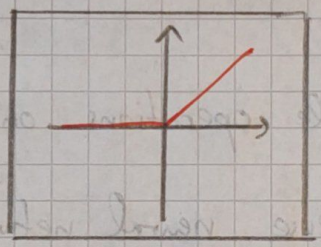
Layered structure: $\cdots \sigma(W_2 \sigma(W_1 \vec{x} + \vec{b}_1) + \vec{b}_2) \cdots$

Non linearity? ← activation function

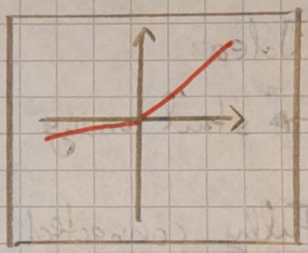


sigmoid $s(x) = \frac{1}{1+e^{-x}}$

→ historically important, only used for last layer in classification (or softmax: higher dim. version)



ReLU (rectified linear unit)



Leaky ReLU

→ most common these days

to wrap up:

→ trainable parameters: w_{ij}^{ℓ}, b_j^{ℓ} ← Layer

→ but what about:

↳ Number of inner layers
↳ number of units per layer
↳ what activation function

} hyperparameters!

→ have to be chosen by hand

how to chose hyperparameters?

↳ experiments + intuition

↳ very problem specific

Back to original motivation: (and theoretically)

↳ powerful? → NN have to be proven experimentally to be able to describe all kind of functional dependencies!

↳ efficient? → layered structure of simple operations

→ matrix multiplication can be parallelized

→ gradients can be computed efficiently (Backpropagation algorithm)

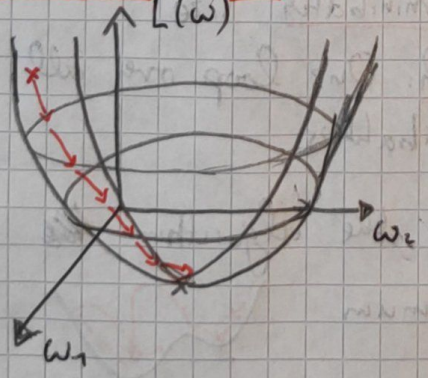
Training

training ≙ minimization of loss function with respect to network weights

Example: $L(w) = \sum_{i=1}^{N} |f_w(x_i) - y_i|^2$

(f_w: neural network)

Gradient descent

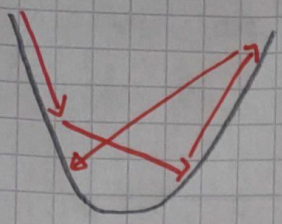


→ use d gradient to find minimum

→ $w_{t+1} = w_t - \eta_t \nabla_w L(w)$

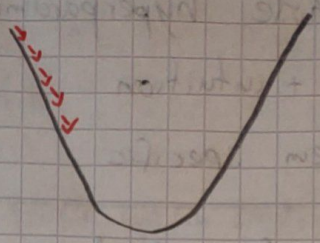
learning rate

- controlls step size
- hyperparameter



large learning rate

→ can't reach minimum



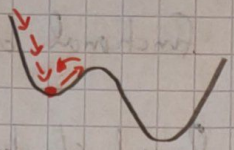
small learning rate

→ too many steps, expensive

problems:

→ stuck in local minimum

→ gradients are expensive for large datasets



⇒ <u>Stochastic gradient descent</u>

$$L(w) = \sum_{i=1}^{N} l_i(w) \quad | \text{ example } l_i(w) = |f(x_i, w) - y_i|^2$$

Idea: Approximate gradients with batches of data

$$\nabla_w L(w) = \nabla_w \sum_{i=1}^{N} l_i(w) \longrightarrow \sum_{i=1}^{M} \nabla_w l_i(w)$$

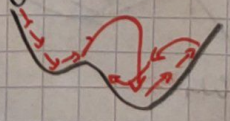
$$= \sum_{i=1}^{N} \nabla_w l_i(w)$$

with $M < N$

<u>M: minibatch</u> size

<u>Epoch</u>: One loop over all minibatches

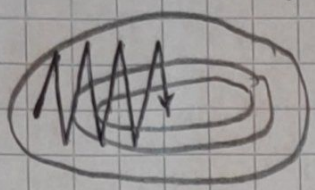
→ adds stochasticity which can help the algorithm to get out of a local minimum



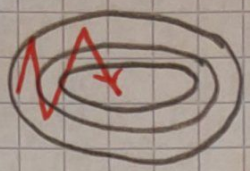
→ less expensive!

- there are more algorithms but they build up on the idea of SGD

↳ SGD + momentum: speeds up in direction with small but consistent gradients!



SGD



SGD + momentum

↳ Adam: very commonly used; uses in addition $\left(\nabla_w L(w)\right)^2$ (gradient squared)

~~Summary so far:~~

~~↳ we know how basic neural networks work~~

~~→ provide us with powerful models~~

~~↳ we know how to train them~~

~~↳~~

~~We have~~

---

to wrap up:

→ Training is just finding a minimum of the loss function

→ we have efficient algorithms to train neural networks.

SGD, Adam, ...
