NEURAL NETWORKS

Before specifying how neural nets are parametrised, we will talk about the optimisation algorithm that is used to train them, which is the same we use for the logistic regression model

Gradient Descent

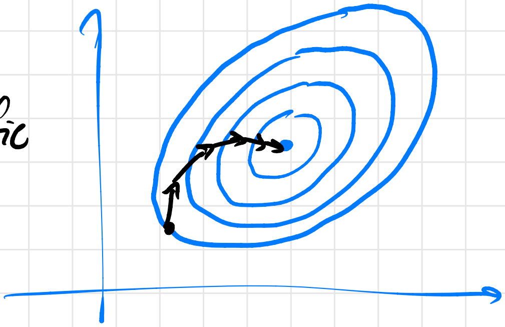
An iterative algorithm to find the minimum of a function $C(\vec{w})$ such that, at iteration $\tau+1$, the parameters $\vec{w}$ are updated as

$$\vec{w}_{\tau+1} = \vec{w}_{\tau} - \eta \frac{dC(\vec{w})}{d\vec{w}}$$

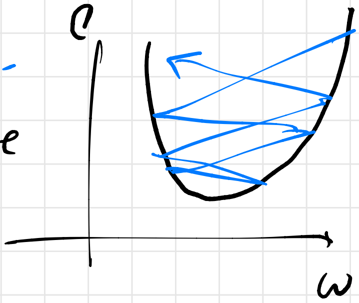
when arrived at the minimum, the derivative is zero and the parameters don't evolve any more

called the "learning rate" and it could depend on τ

The learning rate has to be tuned for each specific case :



- If too large, the algorithm may miss the minimum.
- If too small, there is a waste of computational time.



- The state-of-the-art in optimisation of neural nets is given by variations of what it's called "Stochastic Gradient Descent".

  The motivation is two-fold:

  - Standard G.D. requires evaluation of the full $C(\vec{w}) = \frac{1}{N}\sum_{i=1}^{N} C_i(\vec{w})$. So it may be costly to evaluate for large datasets
  - Standard G.D. gets easily stucked

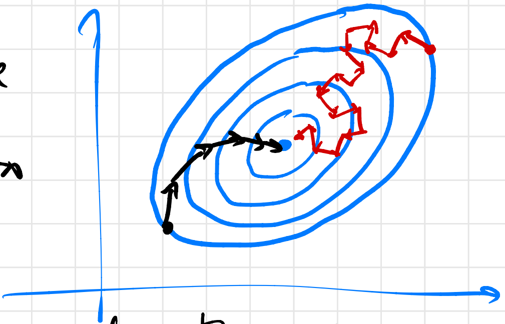
in local minima, and thus unable to find the global minimum

The idea of Stochastic G.D. is to approximate

$$\frac{dC(\vec{w})}{d\vec{w}} \approx \frac{dC_i(\vec{w})}{d\vec{w}}$$

where "i" is a data point (or a subset of data points, a.k.a. "batch") randomly chosen

- The trajectory will have thus a random direction at each iteration, <u>but</u>



on average it will converge to the minimum, because

$$\langle C_i(\vec{w}) \rangle = \frac{1}{N} \sum_i C_i(\vec{w}) = C(\vec{w})$$

Also, because of this stochastic behavior, it is easier for the algorithm to jump out of a local minimum.

Neural nets

Let's start with the shape of a linear model:

$$\underbrace{f(\vec{x}, \vec{v})}_{\text{parameters}} = f\left(\sum_{h=1}^{H} v_h \, \phi_h(\vec{x}) \right)$$

$$f(a) = \begin{cases} a & \text{in regression problems} \\ \text{sigm}(\sigma) & \text{in classification} \\ \text{softm}(\sigma) & \end{cases}$$

The idea of neural nets is to have more flexibility/expressivity by letting the $\phi_h(\vec{x})$ themselves depend on other parameters

$$\phi_h(\vec{x}, \vec{w}_h) = g\left(\vec{w}_h^T \cdot \vec{x} + w_{0h}\right)$$

↳ non-linear "activation" function

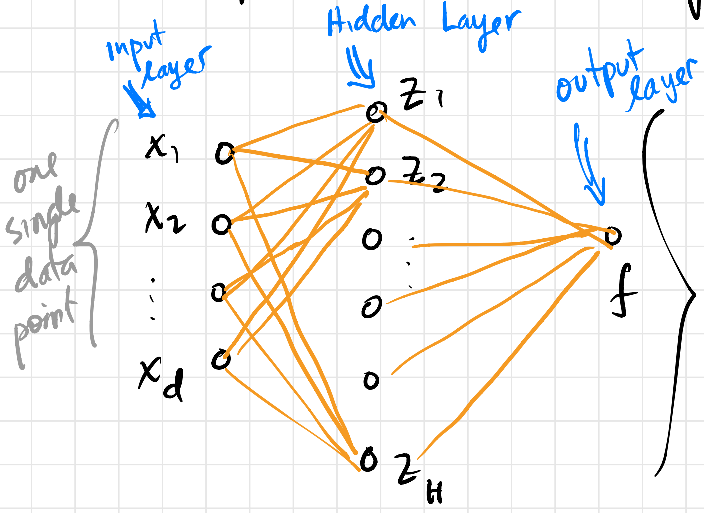
So the model becomes

$$f(\vec{x}; \vec{v}, W) = f\left(\sum_{h=1}^{H} v_h \underbrace{g(\vec{w}_h^T \cdot \vec{x} + w_{0h})}_{\equiv z_h} + v_0\right)$$

$\equiv z_h \Rightarrow$ "neurons"

This model can be represented schematically as

$W \equiv \{w_{hd}\}$



This is for regression or for binary classification where there is only 1 output.

For multiclass classification we had

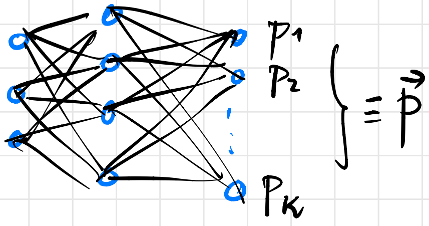
$$p(C_k|\vec{x}) = \sigma_k(\vec{a}) \rightarrow \text{softmax}$$

So we have $k-1$ outputs (since the last one is obtained by

$$p(C_K|\vec{x}) = 1 - \sum_{k=1}^{k-1} p(C_k|\vec{x})$$

$$P_k \equiv p(C_k|\vec{x})$$
$$= f\left(\sum_{h=1}^{H} v_{hk}\, g\left(\sum_{d=1}^{D} w_{hd} x_d + w_{ho}\right) + v_{ko}\right)$$

- Now there is a compact way of writting this function containing the predictor for all data points $\vec{x}_i$



$$P = \{\vec{p}(\vec{x}_1), \vec{p}(\vec{x}_2), \ldots, \vec{p}(\vec{x}_N)\} \qquad \dim P = N \times K$$

$$X = \{\vec{x}_1, \vec{x}_2, \ldots, \vec{x}_N\} \qquad \dim X = N \times D$$

$$W = \{w_{dh}\} \qquad \dim W = D \times H$$

$$W_0 = \{w_{ho}\} \qquad \dim W_0 = N \times H$$

$\rightarrow$ all raws are identical

$V = \{ v_{hk} \}$ $\quad$ $\dim V = H \times K$

$V_0 = \{ v_{k0} \}$ identical rows $\quad$ $\dim V_0 = N \times K$

$$\boxed{P = f\big(g(X \cdot W + W_0) \cdot V + V_0 \big)}$$

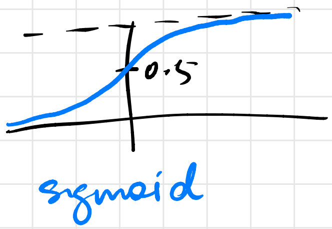
↳ 1-Hidden Layer Neural network.

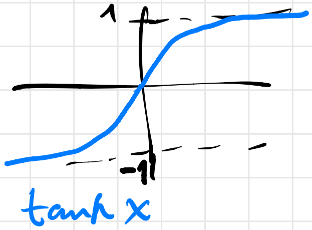
What about a 2-hidden layer network?

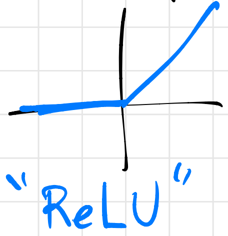
$$P = f\Big(g_2 \big[\underbrace{g_1(X W_1 + W_{10})}_{\text{has this instead of just } X} W_2 + W_{20} \big] V + V_0 \Big)$$

and similarly for neural nets with many hidden layers.

There are many possibilities for the shape of the activation functions (counted 30+ in Wikipedia)







All of them are able to achieve very good results (modulo training problems with some of them)

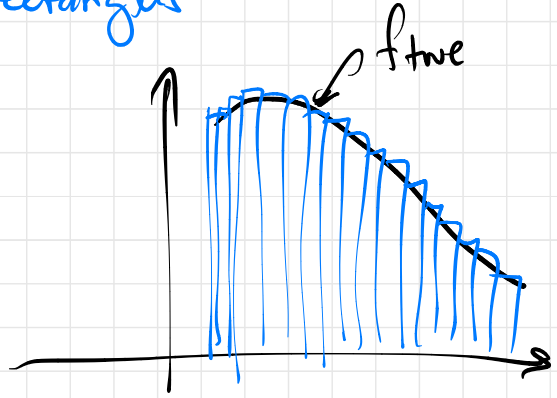
This has to do with:

<u>The Universal Approximation Theorem</u> (Cybenko, '89)

"A feedforward, single hidden layer NN containing finite # (#) of neurons can approximate any continuous function under mild assumptions on the activation function."

In practice: all it requires is to be able to approximate any function using an arbitrary # of rectangles

{see supplementary material}



Training a 1HLNN

This is just the MLE of the parameters of the network, where the likelihood is the usual cross-entropy (in the case of multiclass classification).

$$C(\vec{w}) = -\sum_{i=1}^{N}\sum_{k=1}^{K} \underbrace{t_{ik}}_{\text{observed output}} \ln \underbrace{y_k(\vec{x}_i, \vec{w})}_{\text{prediction}}$$

$D = \{\vec{x}_i, \vec{t}_i\}$

Differentiating $C(\vec{w})$ wrt all the parameters is straightforward (but long), so I only present here the results:

$$\frac{\partial C}{\partial v_{0k}} = \sum_i (Y-T)_{ik}$$

$Z = \{z_{ih}\}$

$$\frac{\partial C}{\partial V} = Z^T \cdot (Y-T)$$

$$\frac{\partial C}{\partial w_{oh}} = \sum_i \left[(Y-T)V^T * Z * (\mathbf{1}-Z) \right]_{ih}$$

element-by-element product

$$\frac{\partial C}{\partial W} = X^T \left[(Y-T)V^T * Z * (\mathbf{1}-Z) \right]$$