Neural Network Basics

José R. Dorronsoro
Dpto. de Ingeniería Informática
Escuela Politécnica Superior
Universidad Autónoma de Madrid

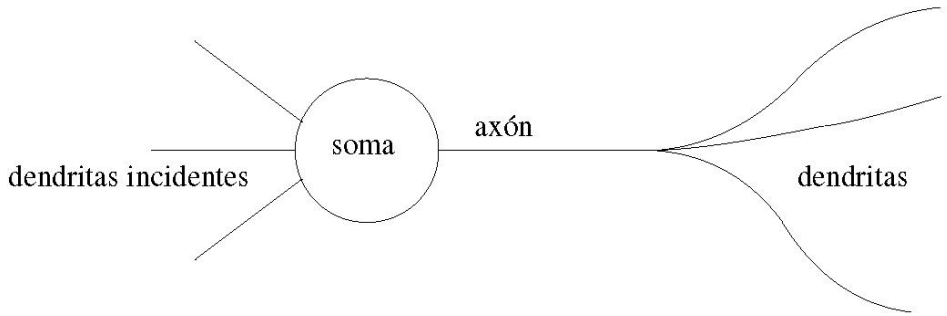
2022

At The Beginning

1. Motivation
   - At The Beginning ...
   - Linear Regression Basics
2. Multilayer Perceptrons
   - Classical MLPs
   - MLP Training
   - MLP Regularization

Basic Neural Models

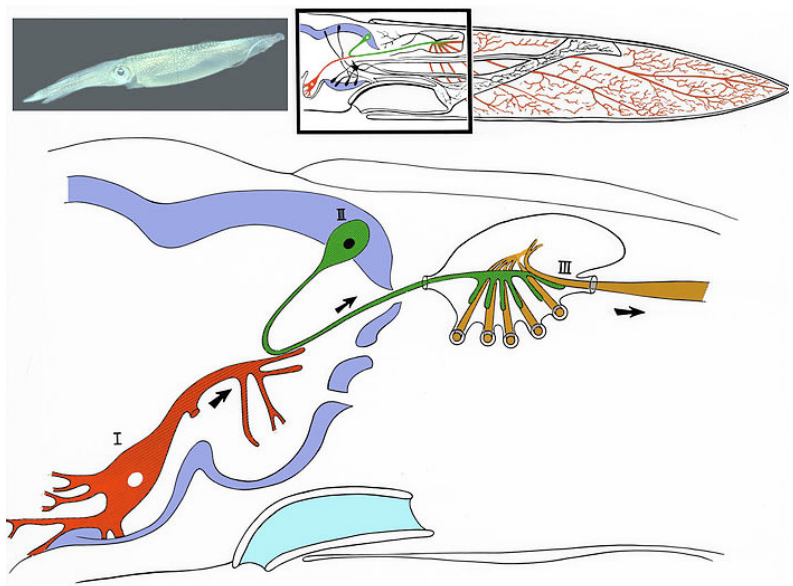
- Basic model: Ramón y Cajal's neuron (1900)



- Basic behavior: the neuron either fires or stays at rest depending basically on its inputs
- This translates into an electrical nerve impulse or **action potential**
- The brain has about 10^{11} neurons
  - Each one has about 7,000 connections
  - These connections are often **recurrent**

Hodgkin–Huxley

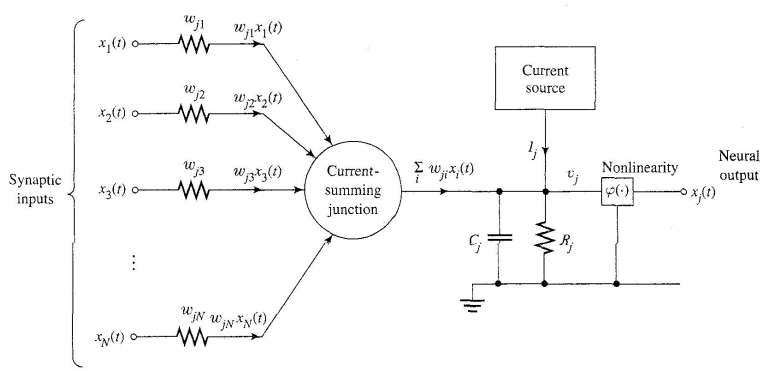
- They developed (circa 1935) the first electro-physiological model to describe the generation and propagation of action potentials in neurons
- They used the giant squid axon for this



From Wikipedia's Squid Giant Synapse

Electronic Neuron

- Electronic version: McCulloch–Pitts (1940)



  - Since $I = \frac{V}{R} = wV$, with w the conductance, the McCulloch–Pitts neuron output is

$$H\left(\sum_{j=1}^{d} w_j V_j + I\right) = V$$

    with the Heaviside function H ensuring a 0–1 output

- The **perceptron** is a mathematical formulation of the McC–P neuron where **the conductances are to be learned**

Rosenblatt's Perceptron

- Given a sample $S = \{(x^p, y^p)\}$ with $y^p = \pm 1$, **Rosenblatt's Perceptrons** (PCPs) are linear machines $w \cdot x$ such that

$$\begin{aligned} w \cdot x^p &> 0 \quad \text{if} \quad y^p = 1 \\ w \cdot x^p &< 0 \quad \text{if} \quad y^p = -1 \end{aligned}$$

- This can be seen as a **homogeneous** classification problem
- In more compact form we want for all p

$$y^p \;\; w \cdot x^p > 0$$

- If such a **separating** w exists, it can be computed in many ways
- But Rosenblatt's goal was to **learn** w, i.e., to arrive at a separating w by repeatedly examining the (x^p, y^p) and adjusting w if necessary

Rosenblatt's Delta Rule

- The **Delta Rule** algorithm

  Start with $w = 0$
  While stopping condition not met:
  Get a new pattern $(x^p, y^p) = \left(x^{p(t)}, y^{p(t)}\right)$
  if $y^p w \cdot x^p \leq 0$ then:
  $w = w + y^p x^p$

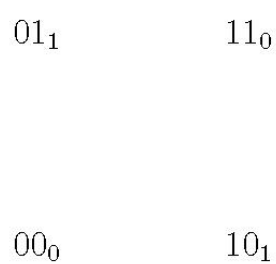
- **Novikov's Theorem**: if the sample S is **linearly separable**, the Delta rule yields a separating w in a **finite number of steps**
- Thus, the perceptron **learns** (well, in some sense!!)

The Problems with Perceptrons

- Rosenblatt's Perceptron caused a big stir on the feasibility of **Conexionist Artificial Intelligence** but soon its first winter came
- Observation 1: A PCP can learn (obviously) **only linearly separable problems**; for instance, it can learn the AND and OR predicates

(a) AND *(b)* OR

- But a **PCP cannot learn the XOR predicate**



Thus . . .

- In practice we can expect the fraction of linearly separable two class problems to be very small
- Thus, linear PCPs will not be useful
- But a **one hidden layer** PCP can solve XOR and, in fact, **separate any convex region** from the rest of the space
- And a **two hidden layer** PCP can separate any **polyhedral region** and hence **solve any classification problem**
- The same is essentially true for regression problems
- Thus, multilayer perceptrons held a great potential
- But **no algorithm is known to learn these 2–hidden layer PCPs**
- And the interest in PCPs quickly decayed
- Another point of view had to be pursued

Linear Regression Basics

1. Motivation
   - At The Beginning ...
   - Linear Regression Basics
2. Multilayer Perceptrons
   - Classical MLPs
   - MLP Training
   - MLP Regularization

Linear Models

- Assuming $x \in R^d$, the basic linear model is

$$f(x) = w_0 + \sum_1^d w_i x_i = w_0 + w \cdot x$$

- w_0 complicates notation; to drop it we center x and y so that $E[x_i] = E[y] = 0$; then $w_0 = 0$
- Then we are left with the simpler **homogeneous** model $f(x) = w \cdot x$
- In practice we will always **normalize** x, for instance to have 0 mean and 1 standard deviation (std) on each feature
  - But not y if we may help it (or it is easy to reverse this)
- But: how do we find w?

1–dimensional Linear Regression (LR)

- Assume that features X and target Y are **centered**, i.e., have 0 means
- For 1-dimensional patterns x the LR model then becomes $f(x) = w\ x$
- And the error is then the function

$$\begin{aligned}\widehat{e}(w) &= \frac{1}{2N}\sum_{p=1}^{N}(w\ x^p - y^p)^2 = \frac{1}{2N}\sum_{p}\left(w^2\ (x^p)^2 - 2x^p\ y^p\ w + (y^p)^2\right) \\ &= w^2\left(\frac{1}{2N}\sum_{p}(x^p)^2\right) - w\left(\frac{1}{N}\sum_{p}x^p\ y^p\right) + \frac{1}{2N}\sum_{p}(y^p)^2\end{aligned}$$

- Thus, $\widehat{e}(w) = aw^2 + bw + c$ with $a > 0$ and it has obviously a minimum w^*
- To find it we just solve $\widehat{e}'(w) = 0$

Solving $\widehat{e}'(w) = 0$

- We first compute $\widehat{e}'(w)$, for which we have

$$\widehat{e}'(w) \quad = \quad w \left(\frac{1}{N} \sum_p (x^p)^2 \right) - \frac{1}{N} \sum_p x^p \, y^p$$

- The optimal w^* solves $\widehat{e}'(w) = 0$ and is given by

$$w^* = \frac{\frac{1}{N} \sum_p x^p y^p}{\frac{1}{N} \sum_p (x^p)^2} \quad = \quad \frac{\frac{1}{N} X^t Y}{\frac{1}{N} X^t X} = \frac{\text{covar}(x, y)}{\text{var}(x)}$$

  where X and Y denote the $N \times d$ **data matrix** (vector here) and the $N \times 1$ **target vector**

General Linear Regression

- Assume again that X and Y are centered
- The LR model becomes now $f(x) = \sum_1^d w_i x_i = w \cdot x$
- If Y is the $N \times 1$ **target** vector and we organize the sample S in a $N \times d$ **data matrix** X, the sample mse is given by

$$\begin{aligned} \widehat{e}(w) &= \frac{1}{2N} \sum_p (w \cdot x^p - y^p)^2 = \frac{1}{2N} (Xw - Y)^t (Xw - Y) \\ &= \frac{1}{2N} (w^t X^t X w - 2 w^t X^t Y + Y^t Y) \end{aligned}$$

- Now we have to solve $\nabla \widehat{e}(w) = 0$, i.e., $\frac{\partial \widehat{e}}{\partial w_i}(w) = 0$
- It is easy to see that

$$\nabla \widehat{e}(w) = \frac{1}{N} X^t X w - \frac{1}{N} X^t Y$$

Solving the Linear Equations

- The optimal $\widehat{w}^*$ must verify $\nabla \widehat{e}(\widehat{w}) = \widehat{R}\ \widehat{w} - \widehat{b} = 0$, where

$$\widehat{R} = \frac{1}{N} X^t X, \ \ \widehat{b} = \frac{1}{N} X^t Y$$

- Over the original, non–centered data matrix we have

$$\widehat{R} = \frac{1}{N} (X - \overline{X})^t (X - \overline{X});$$

  i.e., $\widehat{R}$ is the **sample covariance matrix**

- If $\widehat{R}$ is **invertible**, we just solve the linear system $\widehat{R}\ \widehat{w} = \widehat{b}$
- And obtain the sample–dependent optimal $\widehat{w}^*$ as

$$\widehat{w}^* = \widehat{R}^{-1} \widehat{b} = \left(X^t X\right)^{-1} X^t Y = \text{covar}(X)^{-1}\ \text{covar}(X, Y)$$

The Problems with Linear Models

- Their learning is analytic
  - Great at first sight: a unique, precise solution
  - But changing the sample even slightly implies redoing everything from scratch
- Their learning is not cheap: computing the covariance matrix has a $O(N \times d^2)$ cost and invert it has a $O(d^3)$ cost
  - For big data problems it may not possible to solve analytically the equation $\nabla \widehat{e}(w) = 0$
- They result in simple but relatively poor models:
  - They have relatively **small variance** (hence, are robust w.r. small sample changes)
  - But they have relatively **large bias**

Gradient Descent Minimization

- The simplest alternative to the first two problems is **gradient descent**:
  - Starting from some random w^0, we iteratively compute

$$w^{k+1} = w^k - \rho_k \nabla \widehat{e}(w^k) = w^k - \frac{\rho}{n_B}\left(X_B^t X_B w^k - X_B^t Y_B\right)$$

    over a **mini-batch** B with n_B samples X_B and targets Y_B
  - Component wise: $w_i^{k+1} = w_i^k - \rho_k \frac{\partial \widehat{e}}{\partial w_i}(w^k)$
  - ρ_k is the **learning rate**
- If $w^k \to w^*$, then $\nabla \widehat{e}(w^*) = 0$
  - Since our problems have obviously minima, this should be enough
- To fix the third, we must enrich the basic linear model

By the Way ...

- We have just built our first neural network: the linear regression architecture can be seen as a **multilayer perceptron (MLP) with no hidden layers**
- Thinking of this as an MLP seems far-fetched, as we are missing the hidden layers and the much more complicated gradient computation
- But the layer architecture is there and we can also have learning as iterative error minimization, i.e., **training**
- In fact, we will see that MLPs perform **linear regression on the last hidden layer outputs**
- And the process from the input to the last hidden layer produces an **enhanced representation** of the initial features upon which the regression error should be better
- The modern neural networks start from here

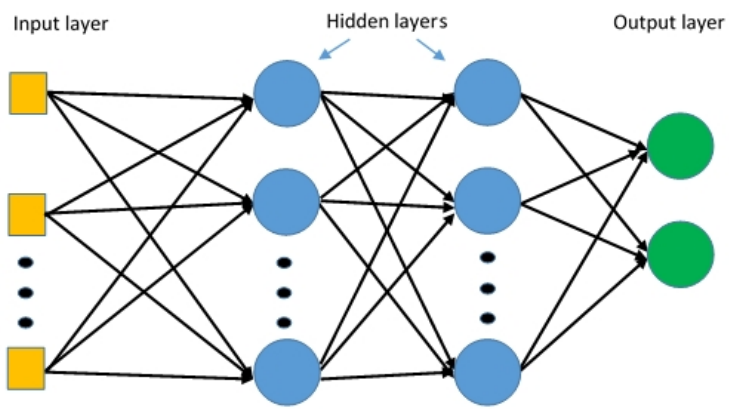
Classical MLPs

1 Motivation
- At The Beginning ...
- Linear Regression Basics

2 Multilayer Perceptrons
- Classical MLPs
- MLP Training
- MLP Regularization

MLP Architecture

- General layout:
  - An input layer (input)
  - One or several hidden layers
  - One output layer
- Feedforward connections only



- Example: TensorFlow Playground

MLP Connections

- **No feedback or lateral conections**
- **Fully connected layers**
- **Linear weight connections** between units and (usually) **non linear activations** inside each unit
  - Combined effect of sucesive layers: potentially highly non–linear transformation
- General processing: layered and feedforward
- In practice (1990s), one hidden layer and only sometimes two
- Later (around 2010): Deep Networks with “many" (from 3 to 10) layers
- Now: almost any architecture, weights in the millions

Unit Activation and Output

- The **activations** a_i^h of a unit in layer h receives the **outputs** o_j^{h-1} from processing in the previous layer

$$a_i^h = \sum_{j=1}^{n_{h-1}} w_{ij}^h o_j^{h-1} + b_i^h,$$

- In matrix/vector form:

$$a^h = W^h o^{h-1} + b^h$$

- **Output** o_i^h of a unit: **non linear processing** $o_i^h = \varphi(a_i^h)$ of its activation
- In vector form:

$$o^h = \varphi(a^h),$$

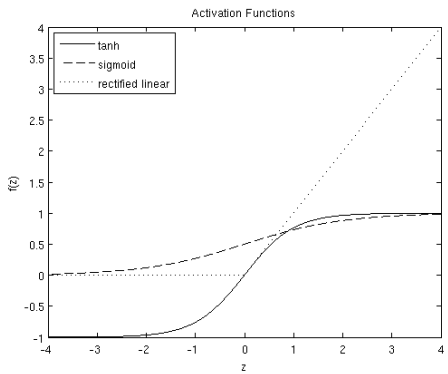
where φ is applied over each unit

Activation Functions

- Choices for f:
  - Heaviside (in the very first Rosenblatt's Perceptrons): $\varphi(a) = 0$ if $a \leq 0$, $\varphi(a) = 1$ if $a > 0$
  - **Identity/linear**: $\varphi(a) = a$
  - **Sigmoid**:
$$\varphi(a) = \sigma(a) = \frac{1}{1 + e^{-a}}$$
  - **Hyperbolic tangent**:
$$\varphi(a) = \tanh(a) = \frac{e^{a} - e^{-a}}{e^{a} + e^{-a}}$$
  - **Rectified Linear Units** (ReLUs): $\varphi(a) = r(a) = \max(0, a)$

Sigmoid, Hyperbolic Tangent, ReLUs

- Sigmoid and tanh: smooth version of Heaviside step function
- For ReLUs $r'(x)$ is either 0 or 1 (hoping $x = 0$ never happens!!)
  - Many gradient elements will go to 0 and many units will have constant activations no matter their input



From Stanford's UFLDL Tutorial

- Usual activation choices:
  - For hidden units: hyperbolic tangent and, currently, ReLU
  - Linear outputs for regression, sigmoid/soft max outputs for classification

The Forward Pass

- We need first to compute an MLP's outputs through the **forward pass**
- At each layer h we compute two variable sets
  - The linear **activations** $a^h = W^h o^{h-1} + b^h$ with o^{h-1} the previous layer's output and W^h, b^h the **weights** and **biases** connecting layer $h-1$ with layer h
  - The nonlinear **outputs** $o^h = \varphi(a^h)$,
- In unit–wise terms: $a_i^h = \sum w_{ij}^h o_j^{h-1} + b_i^h$, $o_i^h = \varphi(a_i^h)$
- Concatenating these operations we arrive at the outputs $\widehat{y} = f(x; \mathcal{W})$ of the network, with $\mathcal{W} = (W^1, b^1, \ldots, W^H, b^H)$ the network's weight set
- Usual choices: ReLU activations, linear outputs for regression, sigmoid outputs for 2-class classification

MLP Error Function

- MSE is the standard error function for regression MLPs

$$\begin{aligned} e(\mathcal{W}) &= \frac{1}{2} E_{x,y}\left[(y - f(x;\mathcal{W}))^2\right] = E_{x,y}\left[e^{\ell}(x,y;\mathcal{W})\right] \\ &= \int e^{\ell}(x,y;\mathcal{W}) p(x,y) dxdy \end{aligned}$$

with y the target associated to input x and $e^{\ell}(x,y;\mathcal{W})$ denotes the **local error**

$$e^{\ell}(x,y;\mathcal{W}) = \frac{1}{2}(y - \widehat{y})^2 = \frac{1}{2}(y - f(x;\mathcal{W}))^2$$

- For a finite sample X^p, y^p, $1 \leq p \leq N$, the above becomes

$$e(\mathcal{W}) = \frac{1}{N} \sum_{1}^{N} (y^p - f(x^p;\mathcal{W}))^2$$

Minimizing the MSE

- The general idea would be to obtain the optimal $\mathcal{W}^*$ as a solution of $\nabla e(\mathcal{W}) = 0$, where we have

$$\begin{aligned} \nabla e(\mathcal{W}) &= E_{x,y}\left[\nabla_{\mathcal{W}} e^{\ell}(x, y; \mathcal{W})\right] \\ &= E_{x,y}\left[\nabla_{\mathcal{W}} f(x; \mathcal{W})(f(x; \mathcal{W}) - y)\right] \end{aligned}$$

  for we have

$$\begin{aligned} \nabla_{\mathcal{W}} e^{\ell}(x, y; \mathcal{W}) &= -(y - f(x; \mathcal{W}))\ \nabla_{\mathcal{W}} f(x; \mathcal{W}) \\ &= \nabla_{\mathcal{W}} f(x; \mathcal{W})(f(x; \mathcal{W}) - y) \end{aligned}$$

- We have therefore two tasks:
  - Compute ∇e^{ℓ} (and then $\nabla e = E[\nabla e^{\ell}]$)
  - Exploit it to build MLPs
- We will exploit $\nabla e(\mathcal{W})$ through **optimization methods** after we compute it

The Forward Pass (again)

- Recall that we compute an MLP's outputs through the **forward pass**
- At each layer h we compute (and store) first the linear **activations** $a^h = w^h o^{h-1} + b^h$ with o^{h-1} the previous layer's output and W^h, b^h the **weights** and **biases** connecting layer $h-1$ with layer h
- We the compute (and store) the nonlinear **outputs** $o^h = \varphi(a^h)$,
- In unit–wise terms:

$$a_i^h = \sum w_{ij}^h o_j^{h-1} + b_i^h, \quad o_i^h = \varphi(a_i^h)$$

- And we also have

$$\frac{\partial a_i^h}{\partial w_{ij}^h} = o_j^{h-1}, \quad \frac{\partial o_i^h}{\partial a_i^h} = \varphi'(a_i^h)$$

Computing the Gradient

- The key tool is the **chain rule** that we will apply backwards from the output layer
- If w_{ij}^h is the general weight connecting unit j in layer $h-1$ to unit i in layer h, we have

$$\frac{\partial e^\ell}{\partial w_{ij}^h} = \frac{\partial e^\ell}{\partial a_i^h}\frac{\partial a_i^h}{\partial w_{ij}^h} = \frac{\partial e^\ell}{\partial a_i^h} o_j^{h-1} = \delta_i^h o_j^{h-1}$$

- That is, we have had an easy to compute term $\frac{\partial a_i^h}{\partial w_{ij}^h}$, and a more complicated but crucial one $\frac{\partial e^\ell}{\partial a_i^h}$
- We refer to $\frac{\partial e^\ell}{\partial a_i^h} = \delta_i^h$ as the **generalized error**

Gradient at the Output Layer

- Let's assume that we have layers $1, 2, \ldots, H-1, H$, with H the output layer
- For simplicity, consider a regression problem with linear outputs
- In the output H layer we have $e^\ell = \frac{1}{2}(y - \widehat{y})^2$ and $\widehat{y} = o^H = a^H = \sum_j w_j^H o^{H-1}$
- Thus, the partial derivative is straightforward

$$\delta^H = \frac{\partial e^\ell}{\partial a^H} = \frac{\partial e^\ell}{\partial \widehat{y}} = \widehat{y} - y$$

  i.e., the generalized error δ^H is here the standard error
- Therefore, we have for the output layer H

$$\frac{\partial e^\ell}{\partial w_j^H} = (\widehat{y} - y)\frac{\partial a^H}{\partial w_j^H} = (\widehat{y} - y)o_j^{H-1}$$

- Now it is (quite) easy to extend this to other layers

Backprop for Other Layers

- Let w_{ij}^h be the weight connecting unit j in layer $h-1$ with unit i in layer h; recall that then

$$\frac{\partial e^\ell}{\partial w_{ij}^h} = \frac{\partial e^\ell}{\partial a_i^h} \frac{\partial a_i^h}{\partial w_{ij}^h} = \frac{\partial e^\ell}{\partial a_i^h} o_j^{h-1}$$

- And if we have already computed the generalized errors $\delta_k^{h+1} = \frac{\partial e^\ell}{\partial a_k^{h+1}}$ for layer $h+1$, we have

$$\begin{aligned} \frac{\partial e^\ell}{\partial a_i^h} &= \sum_k \frac{\partial e^\ell}{\partial a_k^{h+1}} \frac{\partial a_k^{h+1}}{\partial a_i^h} = \sum_k \delta_k^{h+1} \frac{\partial a_k^{h+1}}{\partial a_i^h} \\ &= \sum_k \delta_k^{h+1} \frac{\partial a_k^{h+1}}{\partial o_i^h} \frac{\partial o_i^h}{\partial a_i^h} = \left[\sum_k \delta_k^{h+1} w_{ki}^{h+1} \right] \varphi'(a_i^h) \end{aligned}$$

Backprop and Deep Networks

- The previous formulae work for any number of hidden layers
  - We can work with **deep** MLPs
- Moreover, the chain rule can be automated and modern NN packages compute gradients automatically for very general NN architectures
  - Basic tool: automatic symbolic differentiation on computational graphs
  - This is one of the key reasons of the great success of deep neural nets
- Some examples: convolutional nets, residual nets, recurrent NNs, . . .

MLP Training

1 Motivation
- At The Beginning ...
- Linear Regression Basics

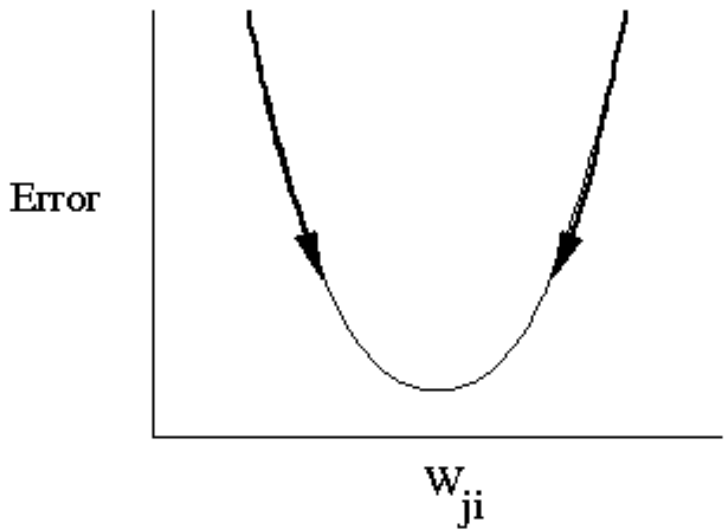
2 Multilayer Perceptrons
- Classical MLPs
- MLP Training
- MLP Regularization

Back to Optimization

- To build a ML model $\equiv$ to **minimize a loss function**
- General optimization theory is a key tool in Machine Learning (ML)
- There are two optimization set ups in ML
  - **Unconstrained** optimization, slightly simpler and the one used for MLPs
  - **Constrained** optimization, wider and more complex
- In ML we have also to consider the optimization of differentiable and also non differentiable error functions
- MLP optimization: **unconstrained and differentiable**
- And also **batch**, i.e., over the entire sample, **mini–batch** over subsamples or **on line**, pattern by pattern

Gradient Descent

- We “stretch" the weight set $\mathcal{W}$ into a vector w
- We recall that $-\nabla e(w)$ is the maximum descent direction
- First idea: to build a (hopefully convergent) sequence w^k iterating (small) steps along $-\nabla e(w^k)$



Gradient Descent II

- In more detail, we start from a random w^0 and compute

$$w^{k+1} = w^k - \rho_k \nabla_w e(w^k)$$

- ρ_k is the **learning rate** (LR)
- With a small ρ_k we ensure $e(w^{k+1}) < e(w^k)$ (although with possibly a very small descent)
- GD is called a **first order method** in part because they only use ∇e
- Variants of more powerful second order methods (such as Newton's or Gauss-Newton's) can be applied to small problems
- But for big data only small cost variants of GD can be applied, such as **Adam**

Adam

- Adam is currently the most widely used gradient-descent method for deep NN training
- At each step t Adam uses a new random mini–batch to
  - Update **exponentially smoothed** averages m_t of the gradient g_t and v_t of the the squared gradient $g_t^2 = g_t \odot g_t$ as

$$m_t = \beta_1 m_{t-1} + (1-\beta_1) g_t, \;\; v_t = \beta_2 v_{t-1} + (1-\beta_2) g_t^2;$$

  - Since we have

$$E[m_t] \simeq (1-\beta_1^t) E[g_t], \;\; E[v_t] \simeq (1-\beta_2^t) E[g_t^2],$$

    we compute bias corrections $\widehat{m}_t$, $\widehat{v}_t$ as

$$\widehat{m}_t = \frac{1}{1-\beta_1^t} m_t, \;\; \widehat{v}_t = \frac{1}{1-\beta_2^t} v_t;$$

  - Update weights as $\mathcal{W}_t = \mathcal{W}_{t-1} - \alpha \frac{\widehat{m}_t}{\sqrt{\widehat{v}_t}+\epsilon}$

Understanding Adam

- Default values $\alpha = 0.001, \beta_1 = 0.9, \beta_2 = 0.999$, and $\epsilon = 10^{-8}$ usually work fine
- What is going on in Adam?
- At each step we work with estimates of the average local gradient:

$$\widehat{m}_t \simeq E[\nabla_{\mathcal{W}} f]; \quad \widehat{v}_t \simeq \sigma(g_t) \simeq E[(\nabla_{\mathcal{W}} f)^2]$$

- Thus, we can first see the Adam iterations

$$\mathcal{W}_t = \mathcal{W}_{t-1} - \alpha \frac{\widehat{m}_t}{\sqrt{\widehat{v}_t} + \epsilon}$$

  as a kind of “normalized" gradient descent
- And in more detail, since we can expect $E[(\nabla_W f)^2] \simeq E[\nabla^2_{W^2} f]$, we can see Adam as a “dampened" variant of diagonal Gauss–Newton’s steps

When to Stop Training

- Typically the $e(\mathcal{W}_t)$ error diminishes towards an asymptotic minimum
  - If many units are used, we may arrive to 0 error, which usually implies **overfitting**
- First solution: to use a separate **validation subset** V and stop training when the error in V, i.e., the **validation error** starts growing
  - This **early stopping** is available in the main NN packages
  - But what do we do for small samples?
- Second solution: **get a good regularization**
  - Now training stops because of reasons such as computational cost, but not because of overfitting risk
- We must combine both, to save on training time and because overfitting is a serious MLP risk

MLP Regularization

1 Motivation
- At The Beginning ...
- Linear Regression Basics

2 Multilayer Perceptrons
- Classical MLPs
- MLP Training
- MLP Regularization

MLPs and Universal Approximation

- In regression we assume $y = \phi(x) + n$ and our goal is to get $f \simeq \phi$
- We say that $\mathcal{F} = \{f(x; \mathcal{W})\}$ is a **Universal Approximation Family** over a domain $\mathcal{R}$ if

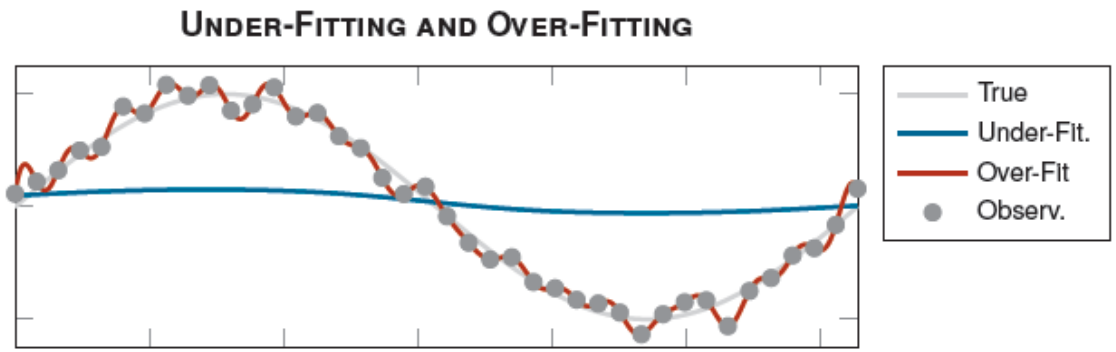
  For any $\epsilon > 0$ and any reasonable ϕ, we can find an $f(x; \mathcal{W}_{\phi,\epsilon})$ s.t.

$$e(\mathcal{W}_{\phi,\epsilon}) = \int (\phi(x) - f(x; \mathcal{W}_{\phi,\epsilon}))^2 p(x) dx \leq \epsilon$$

- Notice that Universal Approximation is just what we need in regression
- In fact a **Single Hidden Layer MLP with enough hidden units is an effective universal approximator**
- But it also implies the risk of **overfit** whose control is crucial

Overfitting in MLPs

- Since MLPs are a UAF, they can also approximate the noise in the sample
  - Given $S = \{(x^p, y^p)\}$ if we allow enough hidden units in a SHL MLP we can arrive to a $\mathcal{W}^*$ s.t. $y^p = f(x^p; \mathcal{W}^*)$
  - We get thus a sample error $\hat{e}(\mathcal{W}^*) = 0$ but possibly with a very high generalization error
- I.e., MLPs may have **very small bias but possibly large variance**



(*Ph.D. Thesis of Carlos Alaíz*)

Regularization vs Overfitting

- Why is there overfitting?
  - Because we may end up having too many weights with respect to sample size
  - Because we allow these weights to explore the entire weight space
- To avoid this, we limit the growth of a w weight by adding a **regularization** term $g(\|w\|)$ to $e(w)$ with $g(r)$ increasing and minimize

$$e_R(w) = e(w) + g(\|w\|)$$

- We thus have to **balance**
  - The minimization of $e(w)$ to get a good model
  - The minimization of $g(\|w\|)$ to prevent overfitting
- This balanced learning results in better generalization

L_2 Regularization

- The simplest regularization procedure adds a quadratic penalty to the square error e

$$e_R(w) = e(w) + \frac{\lambda}{2}\|w\|^2,$$

  with λ the **weight decay** factor
  - This is known as L_2 or Tikhonov's regularization
  - Or as **Ridge Regression** for linear models
- This adds a simple λw^k term to the gradient and gradient descent becomes

$$w^{k+1} = w^k - \rho_k(\nabla_w e(w^k) + \lambda w^k)$$

- Only weights are regularized, not biases

How to choose λ

- Again, the correct choice of λ is crucial
- A small $\lambda \ll 1$ results in a small regularization effect and overfitting risk appears
- A large $\lambda \gg 1$ causes learning to forget about $e(w)$ and the model will be essentially constant and will underfit
- Usually λ is chosen by using CV by exploring a discrete set of values λ_j,
- The same can essentially done for any other hyper–parameter:
  - Number of hidden layers and of hidden units?
  - Learning rate??
  - Minibatch size???

MLP Ensembles

- Recall that for MLPs $e(w)$ does not have a single minimum
- Moreover, the final MLP depends on the random initial w^0
- And mini–batch training adds extra randomness to the final model
- Therefore, **you may never get the same MLP twice!!!**
- But we can turn this to our advantage by
  - Starting from K independent initial weights and get K optimal weight sets $\mathcal{W}_k^*$
  - Giving as the output the average $f_e(x) = \frac{1}{K}\sum_1^K f(x; \mathcal{W}_k^*)$
- We expect outputs of the form $\widehat{y}_k^p = y^p + \epsilon_k^p$ with the ϵ_k^p independent
- Hence $\frac{1}{K}\sum_k \epsilon_k^p \simeq 0$ and $\frac{1}{K}\sum_k \widehat{y}_k^p \simeq y^p$

Takeaways on MLPs

- They enhance linear or logistic regression by improving the initial features
  - The last hidden layer contains them
  - Linear or logistic regression is applied on the last hidden layer
- MLP training not only solves this but also **learns an enhanced representation** of the initial variables
  - Thus MLP perform a kind of automatic feature engineering
- MLPs have a **high risk of overfitting** and must be **regularized**
- MLP **ensembles** can take advantage of the random component in training
  - They imply extra costs but ensembles are **embarrassingly parallelizable**
- The MLP training cost increases **linearly** with sample size
  - This is the key reason for their use in Big Data