Neural Network Basics

José Dorronsoro
Escuela Politécnica Superior
Universidad Autónoma de Madrid

2021

Contents

1 Multilayer Perceptrons

1.1 Basic Modeling

What Is Machine Learning (ML)?

- Lofty definition: make machines learn!!!
  - Have to make "machines" and "learn" more precise
- The machines of ML: mathematical input–output processes that lend themselves to some form of (numerical) parameterization
- The learning process: adjust the machine's parameters until a goal is reached
- New thing: "goal"?
  - At first sight, get something done
  - Ultimately, to minimize some error measure
- Summing things up: a ML process tries to find a concrete mathematical/algorithmic **input–output parameterized transformation** that **minimizes an error measure** by iteratively **adjusting the transformation's parameters**

Where Lies ML?

- In the middle of a possibly long process chain
- Before ML starts we must
  - Go from **raw to organized** data: accesing, gathering, cleaning, formatting, ...
  - Go from **organized to** (potentially) **informative** data: extracting basic and derived features
- After ML finishes and start applying a model, we must perform
  - Outcome **evaluation**: how good/actionable is
  - Outcome **exploitation**: collect, organize, act
  - **Individual model maintenance**: monitor performance, tune hyper–parameters
  - **Modeling life cycle maintenance**: discard old models, introduce new ones and **communicate** our work/results

Supervised/Unsupervised Models

- ML model types: **supervised, unsupervised**
- Supervised models:
  - Targets y^p are known and the model tries to predict or estimate them
  - These known targets guide, or **supervise**, model building
  - Main emphasis here

- Unsupervised models:
  - There are no predetermined or supervising outputs
  - But nevertheless the model is supposed to learn relations or find structure in the data
  - Often as a first step towards a supervised model
- Semisupervised models lie in between: some patterns have targets, some don't

Regression and Classification

- Problems (usually) to be solved by ML models: regression, classification
- In both patterns come in pairs (x, y)
  - x: inputs, predictors, features, independent variables
  - y: target, response, dependent variable; numerical in regression, class labels in classification
- **Regression**: the desired output y is **regressed** into the inputs x to derive a model $\widehat{y} = f(x)$
  - We want $y \simeq \widehat{y}$ so having $y - \widehat{y}$ "small" is the natural goal
- **Classification**: inputs are derived from several classes $C_1, \ldots, C_K$, to which labels ℓ_k are assigned
  - The model now assigns a label $\ell(x)$ to an input x
  - If x is derived from C_k we want to have $\ell(x) = \ell_k$
  - Here having $\ell(x) - \ell_k$ "small" may not make sense

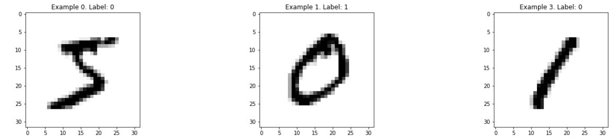
The Boston Housing Regression Problem

- This is a first, widely used "toy" problem
- We want to estimate the median of house values over an area from some information about it which we believe relevant
- Features x: several real estate–related variables of Boston areas
  - CRIM: per capita crime rate by town
  - RM: average number of rooms per dwelling
  - NOX: nitric oxides concentration (parts per 10 million)
  - AGE: proportion of owner-occupied units built prior to 1940
  - LSTAT: % lower status of the population
  - ...
- Target y: MEDV, median value of owner-occupied homes in $1,000's

The MNIST Classification Problem

- This is a well known benchmark, moderately large classification problem

- We have 28×28 grey scale image rasters with handwritten digits 0 to 9
- We want to build a classifier that assigns each raster to one of the 0 to 9 digits
- Features x: 28×28 image matrices
- Target y: labels `[0, 1, ..., 9]`
- Examples (0-padded to 32×32):



Model Parameterization

- Often individual models are selected through (ideally optimal) **parameter sets**
  - The parameters (weights) $w \in R^M$ select a concrete f in a model family $\mathcal{F}$
- **Parametric** models have a fixed functional form $f(x) = f(x; W)$
  - Simplest example: **linear regression**, where x has dimension d, $M = d + 1$ and $W = (w_0, w)$

$$f(x; w_0, w) = w_0 + \sum_{j=1}^{d} w_j x_j = w_0 + w \cdot x$$

- **Semi–parametric** models: also use weights but without a predefined functional form
  - MLPs but also RF or GBR
- **Non parametric** models do not use weights nor follow any broad functional form; Nearest Neighbor models

Issues in Model Building

- There are some initial questions when working with models from a given family $\mathcal{F}$:
  - Which assumptions do they make?
  - How do they work on the input features?
  - How we do build them?
- In turn, these questions lead to another two:
  - How do we select the best model from the given family for the problem at hand?
  - How do we control the model building procedure?
- All of them address fundamental issues that require a **moderately deep** understanding of what is going on under the model's hood
- This understanding is usually framed in **mathematical language**

1.2 Regression Basics

How to Build Regression Models

- In general we have a sample $S = \{x^p, y^p\}$, $1 \leq p \leq N$, with x^p the **features** and y^p the **targets**
- We want to build a model $\widehat{y} = f(x)$ so that $\widehat{y}^p = f(x^p) \simeq y^p$;
  - I.e., we want to **regress** y to the $f(x)$
- The concrete f is chosen within a certain family $\mathcal{F}$
  - Examples here: linear regression, multilayer perceptrons (MLPs)
  - And also: SVMs, Random Forests (RF), Gradient Boosting (GB), nearest neighbor (NN)
- Natural option to ensure $f(x^p) \simeq y^p$: choose f to minimize the sample **Mean Square Error (MSE)**

$$\widehat{e}(f) = \widehat{e}_S(f) = \frac{1}{2N} \sum_{p=1}^{N} (y^p - f(x^p))^2$$

- Thus, the model we select is $\widehat{f} = \widehat{f}_S = \arg\min_{f \in \mathcal{F}} \widehat{e}_S(f)$

Model Estimation as Error Minimization

- For a parametric or semiparametric $f(x; W)$ we can write $\widehat{e}_S(f) = \widehat{e}_S(f(\cdot; W)) = \widehat{e}_S(W)$
- The problem to solve becomes

$$\widehat{W}^* = \widehat{W}^*_S = \arg\min_W \widehat{e}_S(W), \text{ i.e., } \widehat{e}_S(\widehat{W}^*) \leq \widehat{e}_S(W)\ \forall W$$

- In linear regression (LR) the sample error is

$$\widehat{e}(w_0, w) = \frac{1}{2N} \sum_p (y^p - w_0 - w \cdot x^p)^2$$

  i.e., the **sample** error, with a corresponding **population** error

$$e(w_0, w) = \frac{1}{2} E_{x,y}[(y - f(x))^2] = \frac{1}{2} E[(y - w_0 - w \cdot x)^2 p(x, y) dx dy]$$

- LR reduces to **minimize the quadratic form** $\widehat{e}_S(W)$

Regression Assumptions

- **Key assumption**: x and y are related as $y = \phi(x) + n$ where
  - $\phi(x)$ is the **true** underlying function
  - n is **additive noise** with 0 mean and finite variance σ_N^2
- Our sample is just a particular instance of a deeper **sample generation process**
- Thus x, n are produced by **random variables** X, N

  - And so is y, given by $Y = \phi(X) + N$
- Moreover, X and N are **independent distributions** with densities $q(x)$, $\nu(n)$
- Thus, X and Y (or X and N) have a joint density

$$p(x, y) = p(x, \phi(x) + n) = q(x)\ \nu(n) = q(x)\ \nu(y - \phi(x))$$

MSE Decomposition

- We can decompose the MSE error of any model f as

$$\begin{aligned}
2\text{mse}(f) = E_{x,y}[(y - f(x))^2] &= \int (n + \phi(x) - f(x))^2 q(x)\nu(n)dxdn \\
&= \int \left(n^2 + 2n(\phi(x) - f(x)) + (\phi(x) - f(x))^2\right) q(x)\nu(n)dxdn \\
&= \int n^2\nu(n)dn + \int (\phi(x) - f(x))^2 q(x)dx + \\
&\quad 2\left(\int n\nu(n)dn\right)\left(\int (\phi(x) - f(x))q(x)dx\right) \\
&= \sigma_N^2 + E_x[(\phi(x) - f(x))^2]
\end{aligned}$$

- Thus **for any model we have mse$(f) \geq \sigma_N^2$ always**
- And we should focus on achieving on $f \simeq \phi$ (which we don't know!!)

The Best Regression Model

- It is easy to see that the best f is simply $f(x) = E_y[y|x]$, for

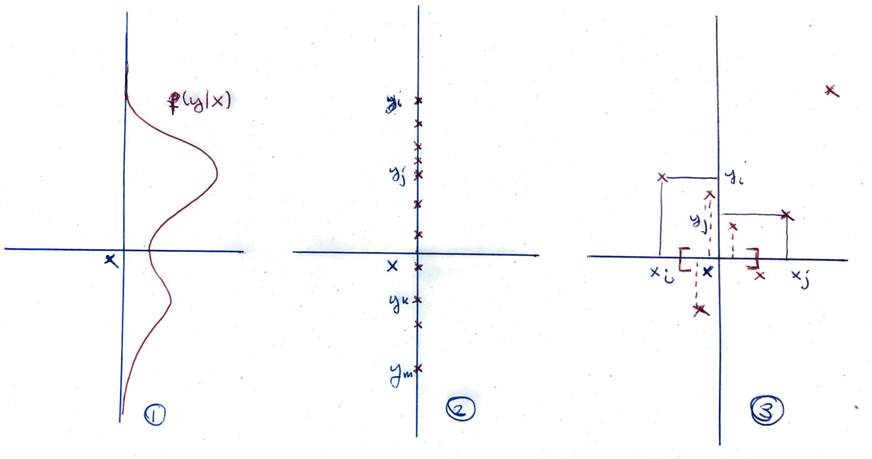
$$E_y[y|x] = E_n[\phi(x) + n] = \int (\phi(x) + n)\, \nu(n)dn = \phi(x)$$

- Have we finished? In theory yes; in practice, not at all!!!
  - We do not know ν and, thus, cannot compute the required integral
  - If we would have several M values y^j for any x, we could try $\widehat{\phi}(x) = \frac{1}{M}\sum_1^M y^j$
  - But this doesn't happen either
- Now we have two options:
  - Try to stretch the $E[y|x]$ approach, for instance, by **k–NN Regression**
  - Forget about it and get back to get models f such that $f \simeq \phi$

From The Best Regressor to k-NN

- The left hand side shows the ideal situation but the right one is what we should expect

k–NN Regression



- We can try to use the y^p values of several x^p close to a new x
- We fix k neighbors $x^{p_1}, \ldots, x^{p_k}$ of x and estimate $\hat{y} = \hat{y}(x)$ as

$$\hat{y}(x) = \frac{1}{k} \sum_{j=1}^{k} y^{p_j}$$

- $\hat{y}(x) = \hat{Y}_k^{NN}(x)$ is the k–**Nearest Neighbor (NN)** regressor
  - This can be refined to weighted versions, such as

$$\hat{y}(x) = \frac{1}{C_k(x)} \sum_{j=1}^{k} \frac{1}{\|x^{p_j} - x\|^2} y^{p_j}$$

  with $C_k(x) = \sum_{j=1}^{k} \frac{1}{\|x^{p_j} - x\|^2}$ a normalizing constant

- But we need that close predictors give predictions that are also close, **and** that there are enough of them close by
  - This is **very unlikely**

The Curse of Dimensionality

- Even for low dimensions and large samples, **the sample space is essentially empty**

- Assume we have 1,000 d–dimensional x patterns whose features have values between 1 and 10
  - In dimension $d = 1$ there are 100 patterns per unit length
  - But when $d = 3$ we have just 1 pattern per volume unit
  - And if $d = 6$ we have just 1 pattern per 1,000 units of volume
  - And in dimension 10 (not a big one nowadays) we have just ... !!!
- Thus, for most problems, **there never will be enough close points**
- As a consequence, to get k observations we may go too far away from x and the average will not be meaningful
- Therefore, unless we deal with violently non–linear problems, a simple linear model may be better than k–NN regression for moderate dimensions

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

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

  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

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

- 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

$$\nabla\widehat{e}(w) = \frac{1}{N}X^tXw - \frac{1}{N}X^tY = \widehat{R}w - \widehat{b}$$

Solving the Linear Equations

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

$$\widehat{R} = \frac{1}{N}X^tX,\ \ \widehat{b} = \frac{1}{N}X^tY$$

- 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

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

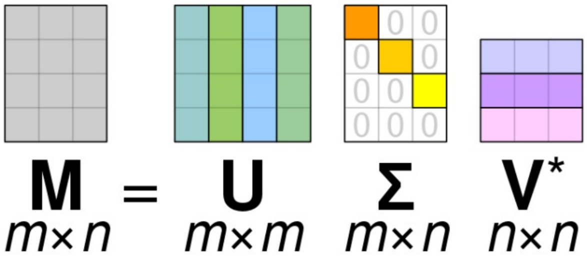
- $\widehat{R}$ is oftent inverted through the **Singular Value Decomposition** (SVD) of the data matrix X

SVD

- The **Singular Value Decomposition** (SVD) of a $N \times d$ matrix X is $X = UDV^t$ where
  - U, V are orthonormal matrices with dimensions $N \times d$ and $d \times d$
  - D is a $d \times d$ diagonal matrix with diagonal elements $\delta_1 \geq \delta_2 \geq \dots$
- The columns of U and V are called the left and right singular vectors; the elements δ_i are the singular values
- Then $X^tX = VDU^tUDV^t = VD^2V^t$ and $(X^tX)^{-1} = VD^{-2}V^t$
  - D^{-2} is a diagonal matrix with $(D^{-2})_{ii} = 1/D^2_{ii}$
- The columns of U and V are unique up to a sign change
  - So different packages may give seemingly different U and V

SVD II

- Sometimes U is given as an $N \times N$ orthogonal matrix and D as an $N \times d$ matrix with zeros in its last $N - d$ rows
- We can represent graphically such a SVD decomposition as follows:



Taken from *Wikipedia*

- We may see this as M performing first a rotation with V^* (our V^t), then feature dilations with Σ (our D) and, finally, another rotation with U
  - Actually both are a roto-reflections

Gradient Descent MSE Minimization

- 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 normal equation $\nabla\widehat{e}(w) = 0$

- The simplest numerical alternative is **gradient descent**:

  - Starting from some random w^0, we iteratively compute

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

    over a **mini-batch** B with n_B samples

  - 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

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 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 feature set upon which the regresion error should be better

Measuring Model Fit

- First option: **Root Square Error** $RSE = \sqrt{\frac{1}{N}\sum(y^p - \widehat{y}^p)^2}$

- OK, but how to compare this? We must always have a **base model** to benchmark our results

- Simplest "model": a constant w_0, which yields the mean $\overline{y} = \frac{1}{N}\sum_1^N y^p$, with square error

$$\frac{1}{N}\sum(y^p - \overline{y})^2 = \text{Var}(y)$$

- We can compare our model against this base model by computing

$$\frac{\sum(y^p - \widehat{y}^p)^2}{\sum(y^p - \overline{y})^2} = \frac{RSE^2}{\text{Var}(y)}$$

- The widely used R^2 coefficient is simply $R^2 = 1 - \frac{RSE^2}{\text{Var}(y)}$

Regularization

- Our regression solution $\widehat{w}^* = \left(X^t X\right)^{-1} X^t Y$ won't work if $X^t X$ is not invertible
  - For instance, when some features are correlated
- We could fix this working instead with $X^t X + \alpha I$ for some $\alpha > 0$
  - It is then always positive definite and we can thus invert it
- To make this practical, note that $\widehat{w}^* = \left(X^t X + \alpha I\right)^{-1} X^t Y$ minimizes

$$e_R(w) = \frac{1}{2N} \sum_p (y^p - w \cdot x^p)^2 + \frac{\alpha}{2} \|w\|^2,$$

- This is the **Ridge Regression** problem
  - Our first example of **regularization**, a key technique in Machine Learning
  - **All ML models must be regularized in some way**
- Important issue: how to find the right choice for α?

Takeaways on Linear Regression

1. We introduced **supervised** models
2. We have reviewed the essentials of the **linear regression model** (always the first thing to try)
3. We have considered model estimation as a problem on **error minimization**
4. We have seen how to build linear models **analytically and numerically**
5. We have seen how to **measure model fit**
6. We have introduced **regularization**

1.3 Cross Validation, Bias and Variance

Sample Bias and Variance

- All built models are **sample dependent**
- With several **independent** samples $S_1, \ldots, S_M$, it is natural to use as our best final model the average of their associated $\widehat{f}_{S_m}(x)$ models, i.e.,

$$\frac{1}{M} \sum_1^M \widehat{f}_{S_m}(x) \simeq E_S[\widehat{f}_S(x)] = \widehat{f}_N(x)$$

  - Here we take the (theoretical) expectation $E_S[\widehat{f}_S(x)]$ over all samples S of size N
- $\hat{f}_N(x) = E_S[\widehat{f}_S(x)]$ is our ideal **best model**
- The **variance** of the $\widehat{f}_S(x)$ estimates is then

$$V_N(x) = E_S\ [(\widehat{f}_S(x) - \widehat{f}_N(x))^2]$$

Bias Versus Variance

- Recall that our goal is to estimate ϕ in our regression model $y = \phi(x) + n$
- Ideally we would like to have a model such that

$$\widehat{f}_N(x) - \phi(x) \simeq 0,$$

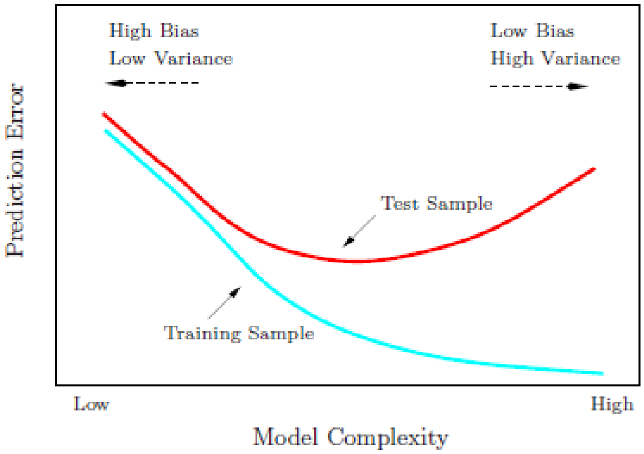
  i.e., a model with small **bias**

  - This should be achievable if we have rich, highly flexible models
  - Or with essentially no regularization (or both!)

- But we would also like to have a robust model building procedure which results in a small **variance** $V_N(x) \simeq 0$
  - This should be achievable if models are simple with few parameters
  - Or with more severe regularization if not
- Obviously both goals are contradictory to a large extent

The Bias–Variance Tradeoff

- There is thus a **tradeoff** between bias (low for complex models) and variance (low for simple models)



Taken from *Hastie et al., p. 38*

Bias vs Variance in k-NN Regression

- Recall that in k-NN regression we choose the k nearest neighbors $x^{p_1}, \ldots, x^{p_k}$ of x and estimate $\hat{y} = \hat{y}(x)$ as
$$\hat{y}(x) = \frac{1}{k} \sum_{j=1}^{k} y^{p_j}$$
- Here the parameter that controls the tradeoff is just k
- If $k = N$, the sample size, the N–NN estimator is just the mean: $Y_N^{NN} = \overline{y}$, with very small variance but large bias (it's an obviously bad model!!)
- If $k = 1$, the 1–NN estimator will have smaller bias but a large variance: changing the sample is very likely to change the sample point nearest to x

Bias vs Variance in Ridge Regression

- Recall that the Ridge Regression loss is
$$e_R(w) = \frac{1}{2N} \sum_p (y^p - w \cdot x_p^p)^2 + \frac{\alpha}{2} \|w\|^2,$$
- Here the parameter that controls the tradeoff is α
- If $\alpha \gg 1$, any non zero w implies a large regularization penalty
  - It is thus likely that $w \simeq 0$ and the Ridge model reduces again to the mean $\overline{y}$, with small variance but possibly a large bias
- But if $\alpha \simeq 0$, w can wander on the entire $\mathbf{R}^d$
  - The bias will be then smaller, but the weights w_S and $w_{S'}$ from different samples may be very different, particularly if the covariance is nearly singular

Evaluating Expected Performance

- It is obvious that before we start applying a model, we should have a reasonably accurate idea of its performance in practice
- I.e., we want to estimate the model's **generalization performance**
- Estimating the generalization performance **only over the sample S used for training results in misleading error values**
- The preceding suggests to have M independent subsamples S_m and then
  - To compute $\widehat{f}_M(x) = \frac{1}{M} \sum_m \widehat{f}_{S_m}(x) \simeq \widehat{f}_N(x)$
  - To get the error estimate $\widehat{e} = \frac{1}{N} \sum_p (y^p - \widehat{f}_M(x^p))^2$ over a new, **unseen** sample $S' = \{(x^p, y^p)\}$
- But since usually we only have a single S, we "simulate" this by **Cross Validation** (CV) to get our first realistic generalization error estimates

Cross Validation

- In Cross Validation (CV) we
  - Randomly split the sample S in M subsets $S_1, \dots, S_M$
  - Work with M **folds**: pairs (S_m, S_m^c), with
    $$S_m^c = S - S_m = \cup_{i \neq m} S_i$$
  - Build M different models **using the S_m^c as training subsets**
  - Compute their errors e_m on the folds' **validation subsets** S_m
  - Use these errors' average as a first estimate of the true model performance
- CV can and **must be used** in any model building procedure
  - Most data science packages have tools to simplify this
- We will also use CV to find **optimal model hyper–parameters** such as α in Ridge Regression

Grid Hyper–parameter Selection

- Consider for Ridge regression a hyperparameter range $[\epsilon, A]$, with $0 < \epsilon < A$
  - In general $\epsilon \simeq 0$, $A \gg 1$
  - $\alpha = \epsilon$: essentially no penalty, small bias and high variance
  - $\alpha = \Lambda$: large penalty, small variance but high bias
- Select an $L + 1$ point **grid** $\{\alpha_0 = \epsilon, \alpha_1, \dots, \alpha_L = A\}$
  - The α_j can be equi–spaced, log equi–spaced, random, ...
- At each α_ℓ
  - Train M models on the S_m^c using the hyperparameter α_ℓ
  - Average their test errors e_m on the S_m to get the error $e(\alpha_\ell)$ at α_ℓ
- And choose the final (hopefully) optimal hyperparameter α^* as
  $$\alpha^* = \text{arg min}_{0 \leq \ell \leq L} e(\alpha_\ell)$$
- α^* gives the model with the **best expected generalization among all possible α choices**

Takeaways on Bias, Variance and CV

1. We have stressed that **any model estimation is sample–dependent** and that this has to be controlled
2. We have introduced the **bias** and **variance** as the two key components of any model error
3. We have discussed **bias–variance trade–off**
4. We have introduced **Cross Validation** here as a tool to estimate a **model's generalization performance**
5. We have also introduced **Cross Validation** as a tool to estimate a **model's hyper–parameters**

1.4 Basic Classification

Regression vs Classification

- Recall that in regression we have numerical continuous targets y and want our predictions $\hat{y}$ to be as close to y as possible
- But in classification we have a finite number of labelled targets for which "selection by closeness" doesn't make sense
- Natural alternative: select the **most probable** label given the pattern x we have just received
  - The concrete labels used for targets do not matter much anymore
  - Model learning should thus be "target" agnostic
  - And good probability estimates should be quite useful
- Let's analyze this in an example

A First Problem: Pima Indian Diabetes

- We want to diagnose whether a person may have diabetes from some clinical measures
- Features x: clinical measures
  - `numPregnant`
  - `bloodPress`
  - `massIndex`
  - `age ...`
- Target y: 0 (no diabetes), 1 (diabetes)
- Clear goal but perhaps too radical
- Better: try to estimate the probability $P(1|x)$ of having diabetes depending on the features x we measure

Classification Setup

- We have random patterns ω from M classes, $C_1, \dots C_M$
- Over each pattern we "measure" d features $x = x(\omega) \in \mathbb{R}^d$
  - x inherits the randomness in ω and becomes a random variable
- A ω has a **prior probability** π_m of belonging to C_m
- Inside each class C_m there is a **conditional class density** $f(x|m)$ that "controls" the appearance of a given x
- The π_m and $f(x|m)$ determine the **posterior probability** $P(m|x)$ that x comes from class C_m

- **Intuition:** we should assign x to the class with the largest $P(m|x)$, that is, work with the classifier

$$\delta(x) = \text{arg max}_m P(m|x)$$

Computing Posterior Probabilities I

- **Bayes rule**: $P(B|A) = \frac{P(A \cap B)}{P(A)}$
- This requires to work with probabilities, not densities, but $P(\{x\}) = P(m \cap \{x\}) = 0$ and

$$P(m|x) = \frac{P(m \cap \{x\})}{P(\{x\})} = \frac{0}{0} = \ldots ???$$

- But we can use the approximation

$$\begin{aligned} P(m|x) &\simeq P(m|B_r(x)) = \frac{P(C_m \cap B_r(x))}{P(B_r(x))} = \frac{P(B_r(x)|m)P(C_m)}{P(B_r(x))} \\ &= \frac{\pi_m P(B_r(x)|m)}{P(B_r(x))} = \pi_m \frac{\int_{B_r(x)} f(y|m)dy}{\int_{B_r(x)} f(z)dz} \end{aligned}$$

  where we assume that features x are measured independently from classes m

Computing Posterior Probabilities II

- Remember the Fundamental Theorem of Calculus:
  if $F(x) = \int_a^x f(y)dy$,

$$\lim_{\epsilon \to 0} \frac{1}{2\epsilon} \int_{x_0-\epsilon}^{x_0+\epsilon} f(y)dy = \frac{dF}{dx}(x_0) = f(x_0)$$

- In d dimensions it becomes

$$g(w) = \lim_{r \to 0} \frac{1}{|B_r(w)|} \int_{B_r(w)} g(z)dz$$

- Putting everything toghether, we arrive

$$\begin{aligned} P(m|x) &= \lim_{r \to 0} P(m|B_r(x)) = \pi_m \lim_{r \to 0} \frac{\int_{B_r(x)} f(y|m)dy}{\int_{B_r(x)} f(z)dz} \\ &= \pi_m \lim_{r \to 0} \frac{\frac{1}{|B_r(x)|} \int_{B_r(x)} f(y|m)dy}{\frac{1}{|B_r(x)|} \int_{B_r(x)} f(z)dz} = \frac{\pi_m \ f(x|m)}{f(x)} \end{aligned}$$

The Obviously Optimal Classifier

- This suggests that a natural choice is the **classifier** δ_B

$$\begin{aligned} \delta_B(x) &= \text{arg max}_m \ P(m|x) = \text{arg max}_m \frac{\pi_m \ f(x|m)}{f(x)} \\ &= \text{arg max}_m \pi_m \ f(x|m) \end{aligned}$$

- With some extra work we can show that this **Bayes Classifier** δ_B defines an optimal solution (in some precise sense) of the classification problem
- But ... this doesn't look too practical, for we do not know either π_m or (much harder) $f(x|m)$

Approximating the Bayes Classifier

- To define δ_B we need to know the prior probabilities π_m and the prior densities $f(x|m)$
- A reasonable choice for π_m is $\widehat{\pi}_m = \frac{N_m}{N}$, where N_m is the number of patterns of C_m in the sample
- But effective multidimensional density estimates are rather difficult, because of the **curse of dimensionality**
  - Densities generalize histograms
  - Good histograms need accurate counts of elements nearby
  - But in high dimensions there won't be nearby elements!!
- Options:
  - Restrict possible density models: **logistic regression**
  - Assume no model and apply a **Nearest Neighbor (NN) strategy**

The k–NN Classifier

- Very simple: at any x consider the subset $N_k(x)$ of its k closest sample points and
  - Let $n_m(x)$ the number of elements of class m in $N_k(x)$
  - Notice that $0 \leq n_m(x) \leq k$
  - Define $\delta_{kNN}(x) = \text{arg max}_m\ n_m(x)$
- That is, $\delta_{kNN}(x)$ assigns x to the class that has more patterns in $N_k(x)$
- We can partially justify this definition from a Bayesian point of view
- Assume that $B_r(x)$ is the smallest ball that contains $N_k(x)$ and consider the approximations
  - $P(B_r(x)|m) \simeq \frac{n_m(x)}{N_m}$
  - Similarly, $P(B_r(x)) \simeq \frac{k}{N}$
  - And $\pi_m \simeq \frac{N_m}{N}$

k–NN and the Bayes Classifier

- We then have

$$\begin{aligned} P(m|x) &\simeq P(m|B_r(x)) = \frac{\pi_m P(B_r(x)|m)}{P(B_r(x))} \\ &\simeq \frac{N_m}{N} \frac{n_m(x)}{N_m} \frac{1}{\frac{k}{N}} = \frac{n_m(x)}{k} \end{aligned}$$

- Therefore δ_{kNN} might be close to δ_B, for

$$\begin{aligned}\delta_{kNN}(x) &= \text{arg max}_m\, n_m(x) = \text{arg max}_m \frac{n_m(x)}{k} \\ &\simeq \text{arg max}_m\, P(m|x) = \delta_B(x)\end{aligned}$$

Some k–NN Issues

- **Q1: How do we choose k?** Using CV, of course
- There are no closed form solution and we have to deal again with the bias–variance tradeoff
  - Small variance with large k: if $k = N$, k–NN classification returns the majority class
  - Small bias with small k: if $k = 1$ a point very close to x should be in the same class
  - But also large variance: the nearest point to x in another sample may well belong to a different class
- **Q2: Is k–NN always meaningful?**
  - Well, only if classes are concentrated and far from each other
  - In other words, when the classification problem is easy!!

Measuring Classifier Accuracy

- Consider a two class problem with labels $y = 0, 1$
- We will call patterns with label 1 **positive** and those with label 0 **negative**
  - Usually the positive patterns are the interesting ones: sick people, defaulted loans, . . .
- Let $\widehat{y} = \widehat{y}(x)$ the label predicted at x; we say that x is a
  - **True Positive** (TP) if $y = \widehat{y} = 1$
  - **True Negative** (TN) if $y = \widehat{y} = 0$
  - **False Positive** (FP) if $y = 0$ but $\widehat{y} = 1$
  - **False Negative** (FN) if $y = 1$ but $\widehat{y} = 0$
- These values are combined in many performance measures

Classifier Metrics

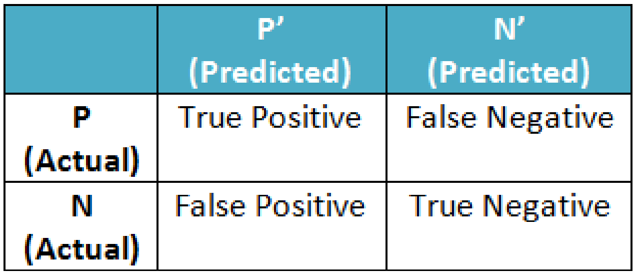
- The classifier **accuracy** is $acc = \frac{TP+TN}{N}$
  - The first thing to measure but ...
- It may not be too significant when the number N_0 of negatives is $\gg N_1$, the number of positives; then
  - The classifier $\delta(x) = 0$ will have a high accuracy $N_0/N \simeq 1$

  - But it will also be useless!!

- First variant: Precision, Recall
  - **Recall**: $TP/(TP + FN)$, i.e., the fraction of positives detected
  - **Precision**: $TP/(TP + FP)$, i.e., the fraction of true alarms issued
- Recall measures how many positive cases we recover, i.e., how effective is our method
- Precision measures the effort we need for that, i.e., its efficiency
- Ideally: high recall, high precision (i.e., effective and efficient!!)
- Another way of presenting the performance is through the **confusion matrix**

The Confusion Matrix

- Standard layout



	P’ (Predicted)	N’ (Predicted)
P (Actual)	True Positive	False Negative
N (Actual)	False Positive	True Negative

- Other layouts:
  - **Positives (with label 1) at bottom** (as done in `confusion_matrix` of `sklearn`)
  - Predicted values in rows, real values in columns
- Accuracy and the confusion matrix can be extended to multi-class problems
  - But not other concepts such as recall or precision

Linear Regression for Classification?

- k–NN Classifier is simple but also crude; have to look elsewhere
- Building a regression model with targets given by some coding of class labels usually doesn’t make sense
- However, for a binary 0–1 response, it can be shown that the $w_0 + w \cdot x$ obtained using linear regression is in fact an estimate of $P(1|x)$
  - We may thus fix a threshold δ_0 and decide 0 if $w_0 + w\dot{x} < \delta_0$ and 1 otherwise
  - However, we may end up with probability estimates less than 0 o bigger than 1!!!

- Better idea: try to transform the linear output $w_0 + w \cdot x \in (-\infty, \infty)$ into a probability $P(1, x) \in (0, 1)$
  - Logistic regression does this through the **logistic** or **sigmoid** function

Logistic Regression (LogR)

- We assume
$$P(1|x) = P(1|x; w_0, w) = \frac{1}{1 + e^{-(w_0 + w \cdot x)}}$$
  - Then $0 \leq P(1|x) \leq 1$ for any x
- We then have
$$P(0|x) = 1 - P(1|x) = \frac{e^{-(w_0 + w \cdot x)}}{1 + e^{-(w_0 + w \cdot x)}} = \frac{1}{1 + e^{w_0 + w \cdot x}}$$
  - Notice that if $w_0 + w \cdot x = 0$, $P(1|x) = P(0|x) = 0.5$
- The ratio $\frac{P(1|x)}{P(0|x)} = e^{w_0 + w \cdot x}$ is called the **odds** of x and its log the **log odds** or **logit**
- Thus, the basic assumption in LogR is that the **logit is a linear function** $w_0 + w \cdot x$ of x
- We have the model $f(x; w)$; we need a **loss** function $L(w_0, w)$ to minimize for which we use the sample's **likelihood**

Estimating w_0^*, w^*

- Assume a single sample x, y and two possible model coefficients w_0, w and w_0', w'
- Denoting by $p = P(y|x; w_0, w)$ and $p' = P(y|x; w_0', w')$, it is clear that we should prefer w_0, w if $p > p'$ and w_0', w' if not
  - In other words, we prefer the coefficients that give a **higher posterior probability**
- For an **independent** sample $S = \{(x^p, y^p)\}$, its joint probability under a posterior model $p = P(y|, x, w_0, w)$ is
$$P(Y|X; w_0, w) = \prod_{p=1}^{N} P(y^p|x^p; w_0, w)$$
- And, again, given two possible model coefficients w_0, w and w_0', w', **we should prefer** w_0, w **iff**
$$P(Y|X; w_0, w) > P(Y|X; w_0', w')$$

Sample's Likelihood

- Therefore, we can estimate the optimal w_0^*, w^* as
$$w_0^*, w^* = \text{arg max}_{w_0, w} P(Y|X; w_0, w)$$

- By the independence assumption we have

$$\begin{aligned} P(Y|X;w_0,w) &= \prod_{p=1}^{N} P(y^p|x^p;w_0,w) \\ &= \left\{\prod_{y^p=1} P(1|x^p)\right\}\left\{\prod_{y^p=0} P(0|x^p)\right\} \\ &= \prod_{p=1}^{N} P(1|x^p)^{y^p}\ \ P(0|x^p)^{1-y^p} \end{aligned}$$

  because

  - If $y^p = 1$, $P(1|x^p) = P(1|x^p)^{y^p} = P(1|x^p)^{y^p}\ \ P(0|x^p)^{1-y^p}$, and
  - If $y^p = 0$, $P(0|x^p) = P(0|x^p)^{1-y^p} = P(1|x^p)^{y^p}\ \ P(0|x^p)^{1-y^p}$

Max Log–Likelihood Estimation

- The **log–likelihood** of w_0, w given S is then

$$\begin{aligned} \ell(w_0,w;S) &= \log P(Y|X;w_0,w) \\ &= \sum_p \{y^p\ \log p(1|x^p) + (1-y^p)\ \log p(0|x^p)\} \\ &= \sum_p y^p\ \log\frac{p(1|x^p)}{p(0|x^p)} + \sum_p \log p(0|x^p) \\ &= \sum_p y^p\ (w_0 + w\cdot x^p) - \sum_p \log(1+e^{w_0+w\cdot x^p}) \end{aligned}$$

- We can thus estimate the optimal $\widehat{w}_0^*, \widehat{w}^*$ as

$$\widehat{w}_0^*, \widehat{w}^* = \text{arg min}_{w_0,w}\ -\ell(w_0,w;S)$$

- Extra bonus: $-\ell$ is a convex differentiable function of (w_0, w) and, thus, it is enough to solve $\nabla\ell(w_0,w) = 0$

Newton–Raphson Solution

- However, $\nabla\ell(w_0,w) = 0$ doesn't admit a closed form solution, but only an iterative, numerical one
- We apply the **Newton–Raphson** iterative method, here equivalent to the general Newton method for function minimization
- For a general weight vector w, and starting with an initial random w^0, Newton's iterations are

$$w^{k+1} = w^k + (\mathcal{H}_\ell(w^k))^{-1}\nabla\ell(w^k)$$

- $\mathcal{H}_\ell(w^k)$ denotes the **Hessian** of ℓ at w^k, which may or may not be invertible

  - Everything is fine if the w^k are close enough to the optimum w^* but far away things may get tricky

- Just as before, we can add a regularization term $\frac{\alpha}{2}\|w\|^2$ to avoid invertibility problems
- The iterations in Logistic Regression are again typical of many of the model building methods used in Machine Learning

Recap: Learning in ML

- The general approach to **learning** is the following:
  - A **model** $f(x;w)$ is chosen
  - Given a sample $S = \{(x^1, y^1), \dots, (x^N, y^N)\}$, we define a **sample dependent loss function**
    $$L(w) = L(w|S) = L\left(y^1, \dots, y^N, f(x^1; w), \dots, f(x^N; w)\right)$$
- $L(w)$ is often minimized from some w^0 by **iterations**
  $$w^{k+1} = w^k - \rho_k G(w^k, S)$$
  with ρ_k a **learning rate** and G some vectorial function
  - When $G(w) = \nabla L(w)$ we have **gradient descent**
  - When $G(w) = \mathcal{H}(w)^{-1}\nabla L(w)$ we obtain **Newton's method**
- When the entire sample S is used at each iteration, we speak of **batch learning**
- When only single patterns (x^p, y^p) or small subsamples are used, we speak of **on–line** or **minibatch learning**

Takeaways on Classification

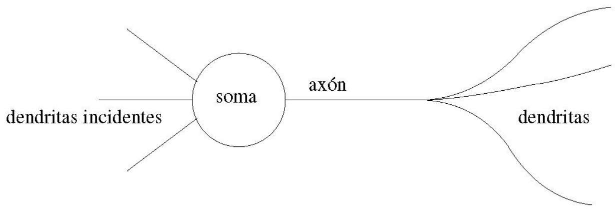
1. We have introduced the classification problem as one of **computing posterior probabilities**
2. We have found the **optimal Bayes classifier** and **approximated it by k–NN**
3. We have introduced several measures of **classifier performance**
4. We have introduced **Logistic Regression** and the numerical minimization of its (minus) log–likelihood
5. We have put Logistic Regression modeling in the general framework of Machine Learning models

2 Multilayer Perceptrons

2.1 Rosenblatt's Perceptrons

Basic Neural Models

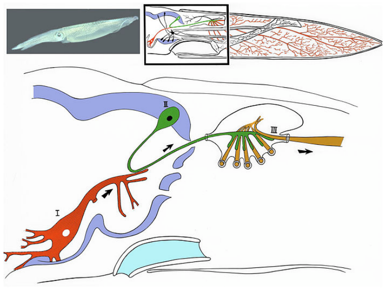
- Basic model: Ramón y Cajal's neuron (1900)



- 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



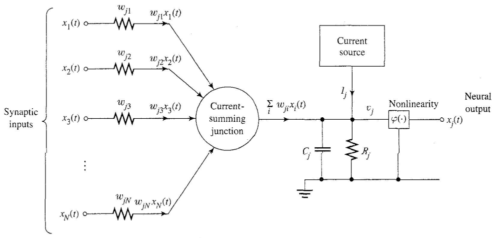
From Wikipedia's Squid Giant Synapse

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:
        $w = w + y^p x^p$
```

- Writing w for the new vector and w' for the previous one, notice that

$$y^p \;\; w \cdot x^p = y^p \;\; w' \cdot x^p + \|x^p\|^2 > y^p \;\; w' \cdot x^p$$

Novikov's Theorem

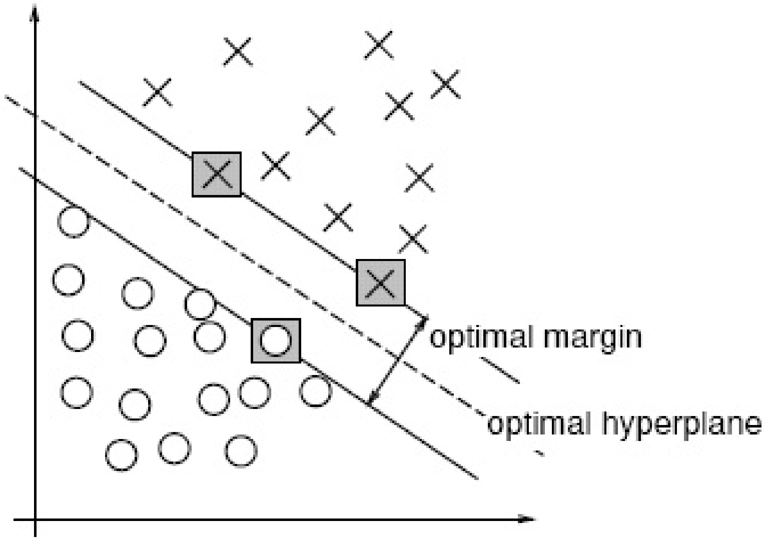
- It states that if the sample S is **linearly separable**, the Delta rule yields a separating w in a **finite number of steps**
  - I.e., it **learns**!!
- More precisely, if for some separating hyperplane w with $\|w\| = 1$ we have $\min_p y^p w \cdot x^p = \gamma > 0$, then **a separating w' can be found after at most $T = R^2/\gamma^2$ wrong classifications, where** $R = \max_p \|x^p\|$
- Notice that γ is the **margin** of the w hyperplane. Thus, if

$$\gamma^* = \max_{\|w\|=1} m(w, S) = \max_{\|w\|=1} \min_p y^p w \cdot x^p$$

  is the **maximum margin**, we will need at most $R^2/(\gamma^*)^2$ iterations to train a Perceptron
- The maximum margin hyperplane gives the best (smallest) Novikov bound
- And problems with smaller margins require more iterations

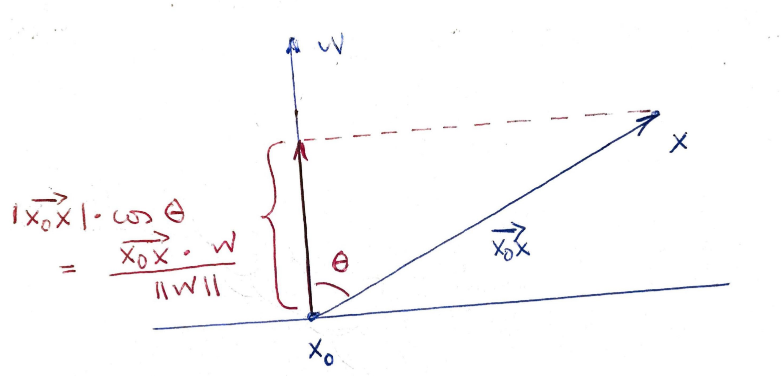
Margins and Generalization

- Intuitively, when the margin is large, the problem is easier



Distance to a Line

- The margin of w is just the smallest distance between sample points and the hyperplane defined by w
- This extends to the multidimensional case: if $\|w\| = 1$ and $b = 0$, the distance of x to the hyperplane defined by w is $|w \cdot x| = yw \cdot x$


w
x
$|\overrightarrow{x_0x}| \cdot \cos\theta$
$= \frac{\overrightarrow{x_0x} \cdot w}{\|w\|}$
θ
$\overrightarrow{x_0x}$
x_0

- Thus the Delta Rule **corrects weight vectors with negative margins**

Novikov's Proof Sketch

- Assume x^t is the t–th pattern such that $y^t w^{t-1} \cdot x^t \leq 0$
- Then $\|w^t\|^2 = \|y^t x^t + w^{t-1}\|^2 \leq \|x^t\|^2 + \|w^{t-1}\|^2$ and, thus,

$$\|w^t\|^2 \leq \|x^t\|^2 + \ldots + \|x^1\|^2 + \|w^0\|^2 \leq tR^2$$

  if $w^0 = 0$ and $\|x^p\| \leq R$ for all p

- Also, if w' is a 1–norm separating hyperplane i.e., $\min_p y^p w' \cdot x^p = \gamma > 0$ for all p, we have

$$w^t \cdot w' = (y^t x^t + w^{t-1}) \cdot w' \geq \gamma + w^{t-1} \cdot w' \geq \ldots \geq t\gamma + w^0 \cdot w'$$

- Putting both together we have for all t for which x^t is erroneous

$$t\gamma \leq w^t \cdot w' \leq \|w^t\| \leq \sqrt{t}R$$

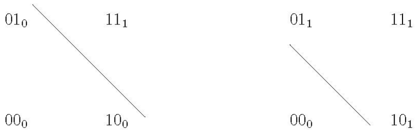
  which implies $t \leq R^2/\gamma^2$

- Thus, the Delta Rule will stop after at most R^2/γ^2 iterations

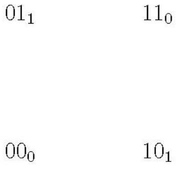
Problems with Perceptrons

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

Dichotomies



(a) AND *(b)* OR



- Observation 2: for PCPs to be useful, **linearly separable problems must be frequent**
- Q: how frequent are they?
- A **dichotomy** for $S = \{x^1, x^2, \ldots, x^N\}$ is any separation of S in two classes
- It corresponds to any asignment of the ± 1 values to the x^p
- The total number of dichotomies is 2^N
- Thus, the **total number of 2-class problems over** S **is** 2^N

Cover's Theorem

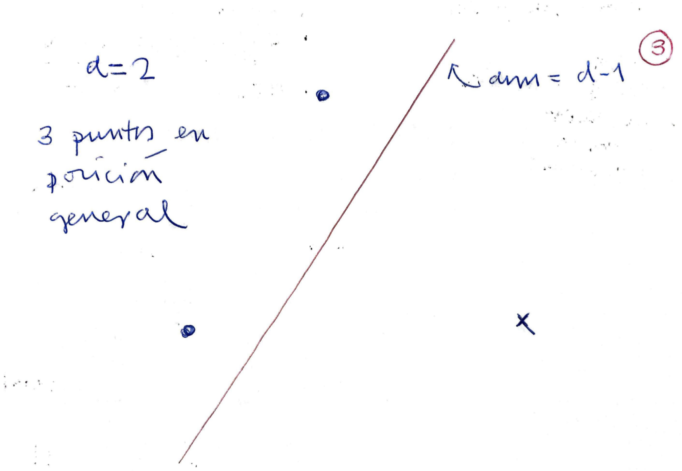
- We have to compare the total number of dichotomies with the number of dichotomies that are **linearly separable**
- The points in a sample S are said to be in **general position** if there are not $d+1$ patterns in a $d-1$–dimensional hyperplane
- **Theorem** If S is in general position, the number $L(N, d)$ of linearly separable dichotomies is

$$L(N,d) = \left\{ \begin{array}{cc} 2^N & \text{si } N \leq d+1 \\ 2\displaystyle\sum_{i=0}^{d} \binom{N-1}{i} & \text{si } N \geq d+1 \end{array} \right\}$$

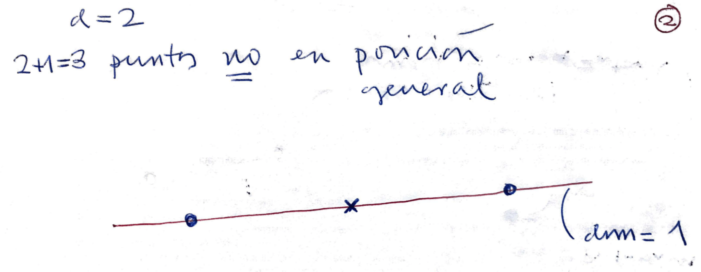
Points in General Position

- Consider $d = 2$, $3 = d+1$ points and a $1 = d-1$-dimensional hyperplane

Points Not in General Position



- Consider now $d = 2$ and $3 = d + 1$ points **not** on a $1 = d - 1$-dimensional hyperplane (i.e., a line)



Thus . . .

- Notice that for d fixed,

$$\frac{L(N, d)}{2^N} \to 0$$

  as $N \to \infty$

- In practice we can expect $N \gg d$ and the fraction of separable dichotomies will 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
- But: **no algorithm is known to learn these 2–hidden layer PCPs**

Why Not Try to Minimize Some Loss?

- Rosenblatt's Perceptrons are natural **online** machines:
  - Patterns appear sequentially and each may change the current weights
- We can also define a perceptron loss over a given sample as

$$L(w|S) = - \sum_{\{p: y^p w \cdot x^p < 0\}} y^p w \cdot x^p$$

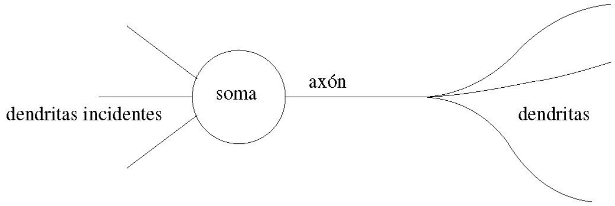
- The local error would thus be $-y^p w \cdot x^p$ when $y^p w \cdot x^p < 0$ and 0 otherwise
- Since $\nabla_w (-y^p \ w \cdot x^p) = -y^p \ x^p$, the Delta Rule can be seen as a kind of local gradient descent on the loss $L(w|S)$:

$$w^t = w^{t-1} + y^{p(t)} x^{p(t)} = w^{t-1} - \nabla_w \left(-y^{p(t)} \ x^{p(t)} \cdot w \right)$$

- However $L(w|S)$ is not differentiable and it is not clear what to make of $\nabla_w L(w|S)$ ...

What Can We Do?

- First option: work with linear models but somehow ensure that $d \gg N$
  - Transform the original d–dimensional features x in new D–dimensional ones such that $D \gg d$
  - We overcome then that, in practice, we usually have $N \gg d$
  - This is the goal of kernel–based Support Vector Machines
- Second option: build trainable PCPs replacing the Heaviside function
  - Work with differentiable PCP transfer function
  - Transform PCP training into a differentiable optimization problem
  - This is what we will do next
- Before doing so, note that **Rosenblatt's PCPs can be seen as the first Machine Learning model**
- And modern ML started around 1985 along the PCP's ideas



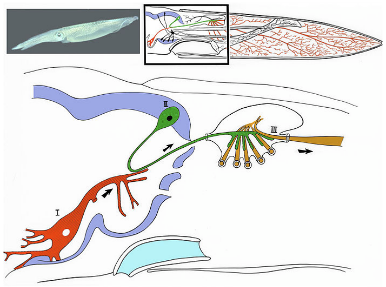
2.2 Classical MLPs

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
- The brain has about 10^{11} neurons with each one having about 7,000 connections, often recurrent

Hodgkin–Huxley

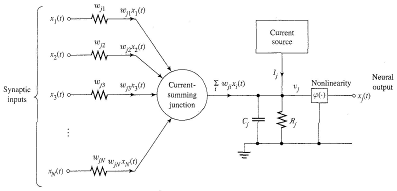
- They developed (circa 1935) the first model to describe the generation and propagation of electrical **action potentials** in neurons
- They used the giant squid axon for this



From Wikipedia's Squid Giant Synapse

McCulloch–Pitts

- Idealized electronic version of a neuron's working (1943)



- Taking weights $w = 1/R$ as conductances, x as potentials and $wx = \frac{x}{R}$ as intensities, the McCulloch–Pitts neuron outputs a potential x_j

$$x_j = H\left(\sum_{j=1}^{N} w_{jk}x_k + I_j\right)$$

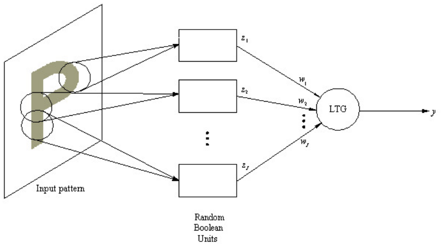
  - The Heaviside function H ensures a 0, 1 output
  - I.e., the neuron fires or does not fire

Basic Questions

- Q1: How to adjust the w_j and I values? I.e., how to "learn" them?
  - It will depend on the problem at hand but it is relatively easy for isolated neurons
- Q2: How to model and reproduce the joint behavior of groups of neurons?
  - Related to the previous questions but quite difficult!!
    - ∗ It is very hard to measure the joint behavior of groups of neurons
    - ∗ It is very hard to take into account the recurrence present in real neurons
- In Artificial Neural Networks (and in ML) one considers just Q1 and (usually) outside any neuro-computational framework

The Perceptron

- Proposed by **Frank Rosenblatt** (1957) as a first step to build an artificial retina



- Key step: separate two classes of vectors x labelled with a target $y = \pm 1$
- Want a weight vector w such that $w \cdot x > 0$ if $y = 1$ and $w \cdot x < -1$ if $y = -1$
- In a unified form we want $y \; w \cdot x > 0$
- First success: this can be done by **learning**

The Delta Rule

- Connectionist paradigm: learning means that weights change as new inputs are received
  - They go through a series of steps $w = w^k$ as they receive the k-th pair x^p, y^p
  - In the Perceptron's case, change w when $y^p \; w \cdot x^p \leq 0$
- Perceptron's adaptive steps:

```
w = 0
while there remain wrongly classified patterns:
    get a new pattern (x_p, y_p)
    if y_p * w.dot(x_p) <= 0:
        # w's response was wrong so we change it
        w = w + y_p * x_p
```

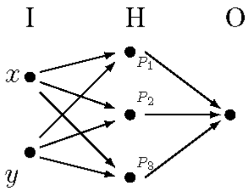
- Then the Perceptron **learns**: if the patterns can be separated, after a finite number of steps we arrive at a w **that separates the patterns**
- But ...

The Connectionist Winter

- Perceptron learning caused a lot of excitement at the time
- However, it could not solve simple separation problems such as the XOR:

01_1	11_0
00_0	10_1

- This can be solved adding more layers to the basic Percetron

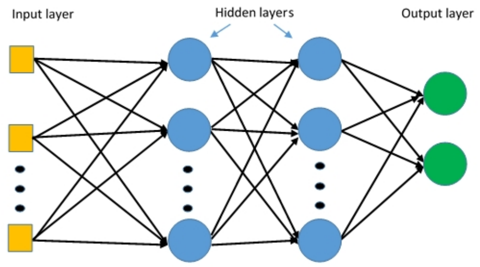


i.e., to work with **Multilayer Perceptrons** (MLPs)

- But **no algorithm to make them learn** was found, and interest in PCPs decayed

MLP's Rebirth

- New proposal (circa 1985): keep the architecture but look to MLP learning as a form of **error minimization**
- MLP's 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 anything!

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**:

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

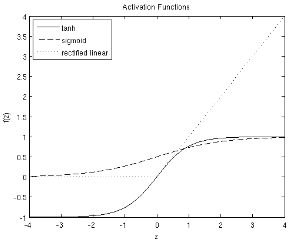
  - **Hyperbolic tangent**:

$$\varphi(a) = \tanh(a) = \frac{e^a - e^{-a}}{e^a + e^{-a}}$$

  - **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 Simplest MLP I

- The Single Hidden Layer (SHL) MLP
  - D inputs (determined by the problem at hand)
  - One hidden layer with H units (number to be chosen) and tanh (or what you prefer) activation
  - One linear or sigmoid output (according to the problem at hand)
- Input–hidden processing: denoting inputs by x and the hidden unit ouput as o^H,

$$o_h^H = \tanh\left(b_h^H + \sum_{j=1}^{D} W_{hj}^H x_j\right)$$

  - In matrix/vector form: $o^H = \tanh\left(W^H x + b^H\right)$

The Simplest MLP II

- Hidden–output processing: since targets are 1–dimensional, we have for the outputs $\widehat{y}$

$$\widehat{y} = \sum_{h=0}^{H} w_h^O o_h^H + b^O,$$

  - In vector form: $\widehat{y} = w^O \cdot o^H + b^O$
  - That is, a **linear model** on the last hidden layer outputs o^H
- Global process:

$$\widehat{y} = f(x; w^O, W^H, b^O, b^H) = b^O + \sum_h w_h^O \tanh\left(b_h^H + \sum_j W_{hj}^H x_j\right)$$

  - Or in matrix/vector form

$$\widehat{y} = f(x; w^O, W^H, b^O, b^H) = b^O + w^O \cdot \tanh\left(b^H + W^H x\right)$$

- Or in Python `y_pred = np.tanh(x.dot(w_H.T)+ b_H).dot(w_O.T)+ b_O`

MLPs and Universal Approximation

- Recall that the regression 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.

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

- 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 we have to be able to build them

MLP Error Function

- MSE is the standard error function for regression MLPs

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

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

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

MSE Gradient

- The general idea would be to obtain $\mathcal{W}^*$ 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
  - Exploit it to build MLPs
- We will exploit $\nabla e(\mathcal{W})$ through **optimization methods** after we compute it

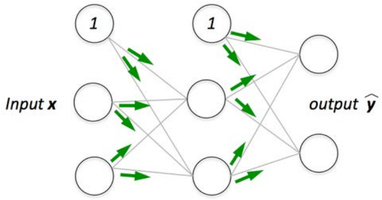
SHL Forward Pass I

- We need thus to compute ∇e, for which we need to compute quite a few auxiliary variables
- We apply the preceding to a single hidden layer (SHL) MLP with
  - A single output unit and input–to–hidden weight matrix $W^H = (W^H_{hj})$ and bias b^H vector and
  - A hidden–to–output weight vector $w^O = (w^O_1, \ldots, w^O_H)$ and scalar bias b^O

- Recall that the forward pass can be computed as follows
  - $a^H = W^H x + b^H$, $o^H = \varphi(a^H)$,
  - Or unit–wise: $a_h^H = \sum W_{hi}^H x_i + b_i^H$, $o_h^H = \varphi(a_h^H)$
  - $y = w^O \cdot o^H + b^O = \sum_h w_h^O o_h^H + b^O$
- Straightforward to program

SHL Forward Pass II

- Graphically we have the following scheme:

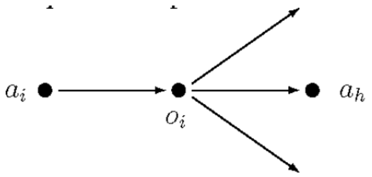


From Sebastian Raschka's A Visual Explanation of the Back Propagation Algorithm for Neural Networks, KDnuggets

- We turn next to the gradient computation
  - Do we need to work on that? Current Deep NN backends compute the error gradient "automatically"
  - But, as Andrej Karpathy says, Yes, you should understand backprop

Backprop's Basic Idea

- Key idea: use the activations a_i^h as intermediate variables



- This suggests to apply the chain rule with the a_h in the next layer as intermediate variables
- Let w_{ij}^h be the weight connecting unit j in layer $h-1$ with unit i in layer h; then

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

Backprop's Details

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

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

  - It has a $O(1)$ cost, as we can precompute the bracketed term

- Thus we start at the easy partials $\frac{\partial e^\ell}{\partial a_i^{\mathcal{L}}}$ and $\frac{\partial e^\ell}{\partial w_{ij}^{\mathcal{L}}}$ at the last layer $\mathcal{L}$ and go backwards

SHL Gradient Backprop I

- We refer to $\frac{\partial e^\ell}{\partial a_i} = \delta_i$ as the **generalized error**
- In the output layer $e^\ell = \frac{1}{2}(y - \widehat{y})^2$ and $a^O = \widehat{y}$, and thus,

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

  i.e., the generalized error δ^O is here the standard error

- Therefore, in the output layer we have

$$\frac{\partial e^\ell}{\partial w_h^O} = (\widehat{y} - y) \frac{\partial a^O}{\partial w_h^O} = (\widehat{y} - y) o_h^H$$

SHL Gradient Backprop II

- In the hidden layer we **backpropagate** the error $\delta^O = \widehat{y} - y$:

$$\delta_h^H = \frac{\partial e^\ell}{\partial a_h^H} = \frac{\partial e^\ell}{\partial a^O} \frac{\partial a^O}{\partial a_h^H} = \delta^O \frac{\partial a^O}{\partial a_h^H} = (\widehat{y} - y) \frac{\partial a^O}{\partial a_h^H}$$

- And since $y = a^O = \sum_h w_h^O o_h + b^O = \sum_h w_h^O \varphi(a_h^H) + b^O$,

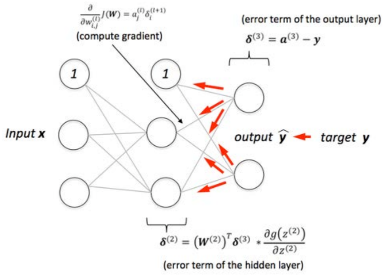
$$\frac{\partial a^O}{\partial a_h^H} = \frac{\partial a^O}{\partial o_h^H} \frac{\partial o_h^H}{\partial a_h^H} = \frac{\partial a^O}{\partial o_h^H} \varphi'(a_h^H) = w_h^O \varphi'(a_h^H)$$

- Moreover, $a_h^H = \sum_j w_{hj}^H x_j + b_h^H$, and

$$\frac{\partial e^\ell}{\partial w_{hj}^H} = \frac{\partial e^\ell}{\partial a_h^H} \frac{\partial a_h^H}{\partial w_{hj}^H} = \delta_h^H x_j = (\widehat{y} - y) w_h^O \varphi'(a_h^H) x_j$$

SHL Gradient Backprop III

- Graphically we have the following scheme:


$\frac{\partial}{\partial w_{i,j}^{(l)}} J(W) = a_j^{(l)} \delta_i^{(l+1)}$
(compute gradient)
(error term of the output layer)
$\delta^{(3)} = a^{(3)} - y$
1
1
Input x
output $\hat{y}$
target y
$\delta^{(2)} = (W^{(2)})^T \delta^{(3)} * \frac{\partial g(z^{(2)})}{\partial z^{(2)}}$
(error term of the hidden layer)

From Sebastian Raschka’s A Visual Explanation of the Back Propagation Algorithm for Neural Networks, KDnuggets

Backprop for Other Layers

- Once we analyze the SHL Backprop, we see that it can be extended easily to other layers
- Let’s w_{ij}^h be the weight connecting unit j in layer $h-1$ with unit i in layer h; 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 $\delta_k^{h+1} = \frac{\partial e^\ell}{\partial a_k^{h+1}}$, we have

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

  with a $O(1)$ cost, as we can precompute the bracketed term

Neural Network Classification

- Recall that our Logistic Regression model gave us estimate of $P(0|x; w_0, w), P(1|x; w_o, w)$
- We then wrote the likelihood of a given sample as

$$P(Y|X; w_0, w) \quad = \quad \prod_{p=1}^N P(y^p|x^p; w_0, w)$$

- And then derived the optimum parameters w_0^*, w^* as

$$w_0^*, w^* = \text{arg max}_{w_0, w} P(Y|X; w_0, w)$$

- Notice that we can see the output of a logistic regression model as that of an **NN with no hidden layers and sigmoid outputs**

Neural 2-class Classification

- We can follow the same path for a 2 class classification problem changing that output for the one
$$f(x;\mathcal{W}) = w_0 + w\cdot\Phi(x,\mathcal{W}^H)$$
with w_0, w the weights from the last hidden layer (LHL) to the output and $\Phi(x,\mathcal{W}^H)$ the map from inputs to the LHL outputs
- And now derive the optimum parameters $\mathcal{W}^*$ as
$$\begin{aligned}\mathcal{W}^* &= \text{arg min}_{\mathcal{W}} & -\sum_{p=1}^{N}\log P(y^p|x^p;\mathcal{W})\\ &= \underset{w_0,w,\mathcal{W}^H}{\text{arg min}} & -\sum_p y^p\ (w_0 + w\cdot\Phi(x^p,\mathcal{W}^H)) +\\ && \sum_p \log(1+e^{w_0+w\cdot\Phi(x^p,\mathcal{W}^H)})\end{aligned}$$
- Forward and backpropagation are also straightforward here
- We change things a little bit for K class problems

MLPs for K-class Classification

- We consider an input layer and a number of hidden layers
- Targets are the 1–hot encodings of class labels, so we use K outputs
- We want the MLP's k–th output to estimate the posterior $P(k|x)$
- The natural output layer activation is thus the **softmax function**
$$\sigma_j(o) = F_j(x;\mathcal{W}) = \frac{e^{w_j\cdot o}}{\sum_1^K e^{w_k\cdot o}}$$
  - Here o are the last hidden layer outputs and w_j is the weight vector in $\mathcal{W}$ which connects the last hidden layer with the network's j output
  - Moreover $\sum_j F_j(x;\mathcal{W}) = 1$ and we assume $P(j|x) \simeq F_j(x;\mathcal{W})$
- For two classes this becomes $\sigma_1(o) = \frac{e^{w_1\cdot o}}{e^{w_0\cdot o}+e^{w_1\cdot o}} = \frac{1}{1+e^{(w_0-w_1)\cdot o}}$
  - We thus get the **sigmoid activation** of Logistic Regression

Multiclass Log–Loss NN Classification

- We change slightly the preceding for a K class classification problem
- We define an MLP $F(x;\mathcal{W}) \in \mathbf{R}^K$ with softmax outputs
  - Recall that $0 \le F_k(x;\mathcal{W}) \le 1$ and $\sum_k F_k(x;\mathcal{W}) = 1$

  - And also we assume that $P(k|x) \simeq F_k(x; \mathcal{W})$

- We label now the multiclass targets via the 1–hot encoding, with the target of class k being the vector $e_k = (0, \ldots, \underbrace{1}_{k}, \ldots, 0)$

- Then, if $x \in C_k$, its label is $y = (y_1, \ldots, y_K)^t$ is e_k, with $y_k = 1$, and

$$P(k|x) = P(k|x)^1 = P(k|x)^{y_k} = \prod_{c=1}^{K} P(c|x)^{y_c} \simeq \prod_{c=1}^{K} F_c(x; \mathcal{W})^{y_c}$$

The Sample Likelihood

- We have a sample $S = \{x^p, y^p\}$ with (x^p, y^p) in class $c(p)$ and, hence, $y^p = e_{c(p)}$

- We also consider a posterior model $P(c|x, w) = F_c(x, \mathcal{W})$, the probability of getting $S = (X, Y)$ is

$$\begin{aligned} P(Y|X; \mathcal{W}) &= \prod_{1}^{N} P(c(p)|x^p; \mathcal{W}) = \prod_{1}^{N} \prod_{c=1}^{K} P(c|x^p; \mathcal{W})^{y_c^p} \\ &\simeq \prod_{1}^{N} \prod_{c=1}^{K} F_c(x; \mathcal{W})^{y_c^p} \end{aligned}$$

- Thus, we estimate $\mathcal{W}$ by maximizing the sample's **likelihood**

$$\mathcal{P}(Y|X; \mathcal{W}) = \prod_{p=1}^{N} \prod_{c=1}^{K} F_c(x^p; \mathcal{W})^{y_c^p}$$

The Cross Entropy Loss

- As before, we will work with the sample's log–likelihood, i.e.

$$\begin{aligned} \ell(\mathcal{W}; S) &= \log \mathcal{P}(Y|X; \mathcal{W}) = \sum_{p=1}^{N} \sum_{c=1}^{K} \log \left(F_c(x^p; \mathcal{W})^{y_c^p} \right) \\ &= \sum_{p=1}^{N} \sum_{c=1}^{K} y_c^p \log F_c(x^p; \mathcal{W}) \end{aligned}$$

- The **cross–entropy** loss is now simply the negative log–likelihood $-\ell(\mathcal{W}; S)$, i.e.,

$$L(\mathcal{W}) = -\ell(\mathcal{W}; S) = -\sum_{p=1}^{N} \sum_{c=1}^{K} y_c^p \log F_c(x^p; \mathcal{W})$$

- It is now straightforward to carry this into a NN setting

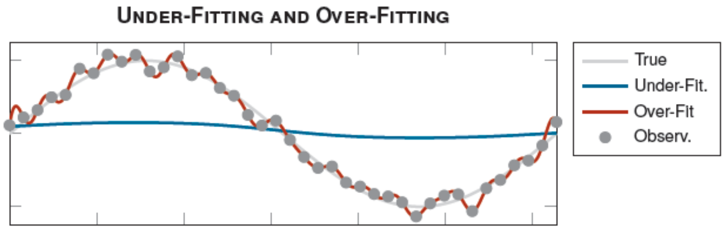
Takeaways on Clasical MLPs

1. They have a layered structure with outputs computed in a **forward pass** using differentiable activations
2. Usual activations: ReLUs, sigmoid, tanh, linear
3. MLPs are **universal approximators**: this is indispensable for regression but has to be handled with care
4. MSE is the usual regression cost; we use a variant of cross entropy in classification
5. The error function gradients are computed by **backpropagation** of generalized errors
6. Backprop is basically a very simple procedure than can be **largely automated**
7. Regression and classification problems fit very well in this setting

2.3 MLP Regularization

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 w^* s.t. $y^p = f(x^p; w^*)$
  - We get thus a sample error $\hat{e}(w^*) = 0$ 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
- We can avoid this wandering if we limit w's growth, for which we add a **regularization** term $g(\|w\|)$ to $e(w)$ with $g(r)$ increasing
- Working with $e_R(w) = e(w) + g(\|w\|)$ we have to **balance** the minimization of $e(w)$ and that of $g(\|w\|)$

- 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

  - Also known as Tikhonov's regularization (or Ridge Regression for linear models)

- The desired effect is to constrain the evolution of w:

  - In fact, the regularized loss is essentially the **Lagrangian of the constrained problem**

  $$\min_w e(w) \text{ subject to } \|w\|^2 \leq \rho, \quad \rho > 0$$

- The gradient becomes $\nabla e_R(w) = \nabla e(w) + \lambda w$ and gradient descent becomes

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

- And the Hessian is $\mathcal{H}_R(w) = \mathcal{H}(w) + \lambda I$

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 w_k^*
  - Giving as the output the average $f_e(x) = \frac{1}{K}\sum_1^K f(x; w_k^*)$
- 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 MLP Regularization

- MLPs have a **high risk of overfitting**
- Thus, they must be **regularized** to avoid overfitting
  - The regularization hyperparameter is chosen through **cross validation**
- MLP training has two **random components**: the initial point and minibatch selection
  - Thus, each training will converge to a **different optimum**
- MLP **ensembles** can take advantage of this
  - They imply extra costs but ensembles are **embarrassingly parallelizable**

2.4 Unconstrained Smooth Optimization

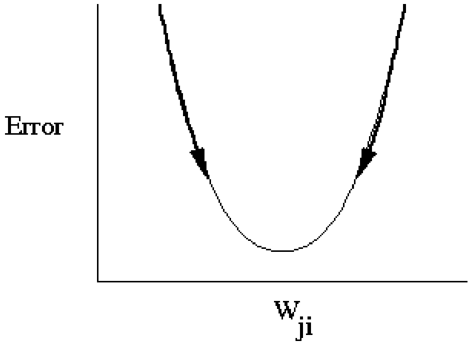
Back to Optimization

- 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)
- We can get a better iteration $w^{k+1} = w^k - \rho_k^* \nabla_w e(w^k)$ using a ρ_k^* given by

$$\rho_k^* = \arg\min_{\rho} e(w^k - \rho \nabla_w e(w^k));$$

  this is known as **line minimization**

- These GD methods are called **first order methods** in part because they only use ∇e

One Dimensional Newton's Method

- Assume a quadratic function $q(w) = aw^2 + bw + c$, with $a > 0$, and a minimum at w^*
- We can reach w^* from any w with a step Δw such that

$$0 = q'(w + \Delta w) = 2a(w + \Delta w) + b$$

- We have thus $\Delta w = \frac{-b-2aw}{2a}$, that is,

$$w^* = w - \frac{2aw + b}{2a} = w - \frac{1}{q''(w)} q'(w)$$

- This leads to **Newton's method**: minimize a general f iteratively using steps

$$w^{k+1} = w^k - \rho_k \frac{1}{f''(w^k)} f'(w^k)$$

  with ρ_k a suitable learning rate

One Dimensional Newton's Method II

- Notice how the $\frac{1}{f''(w_k)}$ acts as a **self adjusting learning rate**
- When the parabola is very sharp (i.e., $a \gg 1$), we will be close to the minimum and the descent can overstep it
- But then $f''(w_k)$ will be big and, hence, $\frac{1}{f''(w_k)}$ **moderates** the gradient step
- Conversely, when the parabola is very wide (i.e., $a \simeq f''(w_k) \ll 1$), the minimum will be far away and the gradient step small descent will
- But then $f''(w_k)$ will be rather small and, hence, $\frac{1}{f''(w_k)}$ **accelerates** the gradient step
- We can extend this to several dimensions

Multidimensional Newton's Method

- For a d dimensional w, the Taylor expansion of e at an optimum w^* is

$$e(w) \approx e(w^*) + \frac{1}{2}(w - w^*)^t \cdot \mathcal{H}(w^*) \cdot (w - w^*)$$

  - $\mathcal{H}(w^*)$ is the **Hessian** of e at w^* and $\nabla e(w^*) = 0$

- It follows that $\nabla e(w) \approx \mathcal{H}(w^*) \cdot (w - w^*)$ and, therefore,

$$w^* \approx w - \mathcal{H}(w^*)^{-1} \nabla_w e(w)$$

- This suggest to derive the w^k by

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

  which is known as **Newton's Method** (NM)

Variants of Newton's Method

- Theoretically NM converges very fast near w^*, but
  - Far from w^* convergence is not guaranteed
  - Moreover $\mathcal{H}(w^k)$ may not be invertible
  - Besides, computing $\mathcal{H}(w^k)$ is cumbersome and costly
- The **Gauss–Newton (GN)** approximation

$$\mathcal{H}(w) \simeq E[\nabla e(w) \nabla e(w)^\tau],$$

  holds for any quadratic cost and simplifies the third problem

- The **Levenberg–Marquardt (LM)** method deals with the other two problems combining

  - Gradient descent “away” from w^*
  - Gauss–Newton “near” w^*

although “away” and “near” have to be properly addressed

One–dimensional GN Approximation

- Assume $e(w) = \frac{1}{2}\int (f(x;w) - y)^2 p(x,y)dxdy$; then

$$\begin{aligned} e'(w) &= \int (f(x;w) - y)\frac{\partial f}{\partial w}(x,y)p(x,y)dxdy; \\ e''(w) &= \int \left(\frac{\partial f}{\partial w}\right)^2 p(x,y)dxdy \\ &\quad + \int (f(x;w) - y)\frac{\partial^2 f}{\partial w^2}(x,y)p(x,y)dxdy \end{aligned}$$

- Near a minimum w^* we may expect $f(x;w) \simeq y$ and, therefore,

$$e''(w) \simeq \int \left(\frac{\partial f}{\partial w}\right)^2 p(x,y)dxdy > 0$$

- Thus, for square errors, we can use first derivates to approximate $e''(w)$

General GN Approximation I

- In the general case we have

$$\nabla e(w) = E\left[\nabla f(x;w)\left(f(x;w) - y\right)\right]$$

- And, therefore,

$$\begin{aligned} \nabla^2 e(w) &= E\left[\nabla^2 f(x;w)\left(f(x;w) - y\right)\right] + \\ &\quad E\left[\nabla f(x;w)\nabla f(x;w)^\tau\right] \end{aligned}$$

- The second term is easy to compute once we have ∇f
- If $w \approx w^*$, $f(x;w) \approx y$; therefore $f(x;w) - y \approx 0$,
  - We can ignore the first, more complex, term

General GN Approximation II

- We arrive at $\nabla^2 e(w) \simeq E\left[\nabla f(x;w)\nabla f(x;w)^\tau\right]$ or, equivalently,

$$\mathcal{H}_{(i,j)(p,q)}(w) = \left(\frac{\partial^2 e}{\partial w_{ij}\partial w_{pq}}(w)\right) \simeq \left(E\left[\frac{\partial f}{\partial w_{pq}}\frac{\partial f}{\partial w_{ij}}\right]\right)_{(i,j)(p,q)}$$

- $\mathcal{J} = E\left[\nabla f(x; w) \nabla f(x; w)^\tau\right]$ is **Fisher's information matrix**
  - $\mathcal{J}$ is **semidefinite positive**
  - And thus not necessarily invertible
- Often only its diagonal is considered and we have

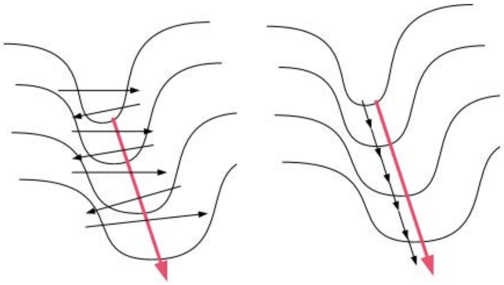
$$\mathcal{H}_{(i,j)(i,j)}(w) = \left(\frac{\partial f}{\partial w_{ij}}\right)^2$$

Advanced Optimization

- There are many more proposals in unconstrained optimization
- The **Conjugate Gradient** (CG) and **Quasi–Newton** (QN) methods are important in classical MLP training
- The basic idea in CG is to replace gradient descent directions $g_k = -\nabla e(w_k)$ with new conjugate directions that try to keep somehow the previous "good directions"
- The basic idea in QN is to iterate as in NM but with simple approximations $\mathcal{A}_k$ to $\mathcal{H}^{-1}(w^k)$ that converge to $\mathcal{H}^{-1}(w^*)$
- When training "small" NNs the **Limited-memory Broyden–Fletcher–Goldfarb–Shanno** (L-BFGS) QN variant is often used

Accelerating Gradient Descent

- A possibility on the error surface of a NN is to have many long, narrow ravines
  - Gradients bounce in the narrow section, but may be very small along the wider one



Momentum

- **Momentum** pushes them forward in the wider section (but we have to cope with a new parameter)
- Momentum tries to maintain descent's inertia with a term $\Delta^k = w^k - w^{k-1}$, i.e.,

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

  - More or less, what a **heavy ball** would do

- The goal is to keep w^k advancing in "plateaux", i.e., small gradient zones
- Momentum can be seen as a crude approximation of a CG step
- Nice explanation at Why Momentum Really Works

Nesterov's Accelerated Gradient

- Let's rewrite momentum in two steps
  1. Define $\Delta^{k+1} = -\rho_k \nabla_w e(w^k) + \mu_k \Delta^k$ and
  2. Apply
  $$w^{k+1} = w^k + \Delta^{k+1} = w^k - \rho_k \nabla_w e(w^k) + \mu_k \Delta^k$$
- **Nesterov's Accelerated Gradient** is a variant of this
$$\begin{aligned} \widetilde{\Delta}^{k+1} &= -\rho_k \nabla_w e(w^k + \mu_k \widetilde{\Delta}^k) + \mu_k \widetilde{\Delta}^k; \\ w^{k+1} &= w^k + \widetilde{\Delta}^{k+1} \end{aligned}$$
- In convex optimization it improves GD and is often highly effective in Deep Network training

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
  $$E[m_t] \simeq (1-\beta_1^t) E[g_t], \quad E[v_t] \simeq (1-\beta_2^t) E[g_t^2],$$
  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 $W_t = W_{t-1} - \alpha \frac{\widehat{m}_t}{\sqrt{\widehat{v}_t}+\epsilon}$

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:
$$\widehat{m}_t \simeq E[\nabla_W f]; \ \ \widehat{v}_t \simeq \sigma(g_t) \simeq E[(\nabla_W f)^2]$$

- Thus, we can first see the Adam iterations

$$W_t = W_{t-1} - \alpha \frac{\widehat{m}_t}{\sqrt{\widehat{v}_t} + \epsilon}$$

  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 $e(w_k)$ error diminishes towards an asymptotic minimum
  - If many units are used, we arrive to 0, 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
  - But: How to choose V? What do we do for small samples?
- Second solution (better): **get a good regularization** (and forget about overfitting)
  - A low CV error is also a low validation error
  - Now training stops because of reasons such as computational cost, but not because of overfitting risk

Takeaways on MLP Optimization

- **Gradient descent** is the simplest method but possibly also the slowest
- **Momentum** can be used to speed it up
- **Newton's method** is much faster but may be very costly and difficult to apply in full form
- All the previous methods require the (usually tricky) selection of a **learning rate**
- Second order methods such as Conjugate Gradient and **Quasi–Newton** avoid learning rates and are more efficient but costlier
- **Limited Broyden–Fletcher–Goldfarb–Shanno** (L-BFGS) is currently the option of choice for "small" MLPs

2.5 Computational Costs of MLPs

MLPs and Big Data?

- Many Vs in Big Data: **Volume, Velocity**, Variety, Veracity, Value, ...
- Velocity: information flows in data streams that require fast processing and feed back
  - MLPs are rather fast
  - Less than linear models but much more so than SVMs

- Volume is probably the greatest attractive of Big Data
  - Huge samples and/or very large pattern dimension
  - Large impact in model training

Volume in MLPs

- Parallelism is the first answer to Big Volume
- On a standalone machine it may be
  - Passive: let the SO distribute work among several CPU cores, or use GPUs with low level parallelized libraries such as Linpack or BLAS for linear algebra
  - Active: explicitly exploit a problem's parallelism programming an algorithm in, say, OpenMP
- It is easy to passively parallelize the training of several MLPs
  - But memory costs may multiply
- Training an isolated MLP is not CPU parallelizable:
  - There is a sequential layer dependence in Backprop
  - Each training iteration depends on the previous one
  - But matrix-vector multiplications can exploit GPU parallelism

Handling Huge Sample Training

- Two consequences of a large N are
  - Sample doesn't fit in memory and we have to split it somehow
  - Training gets "lost", for in the global gradient $\nabla e = E[\nabla e^{\ell}]$ we average many local gradients that may cancel each other out
- The fist problem has been always present in fields such as analog signal filtering
- Solution for both: adaptive or **on line**, i.e., pattern by pattern, weight updates
  - It also allows a relatively simple setting for a theoretical analysis
  - But not used today; instead, medium–to–large NNs are trained using **mini–batches**

MLP Complexity

- MLPs are fast to apply but costly to train
- How can we estimate this cost?
- MLP traning cost is determined by
  - Its **architecture**, that determines the number of weights to fit and that is also dependent on the input dimension D

  - The full or mini–batch **sample size** N, that determines the cost of the averages to be computed
  - The **training method**, with more or less iterations that, in turn, are more or less costly

Forward Pass Complexity

- The number of weights in a single hidden layer (SHL) MLP with D inputs, one output and one hidden layer with H units is

$$(D+1) \times H + (H+1) \simeq HD$$

- Each extra hidden layer pair with (H_1, H_2) units adds $(H_1+1) \times H_2 \simeq H_1 H_2$ weights
- The cost per pattern of the feedforward step between layers $h-1, h$ is basically that of $H_{h-1} \times H_h$ floating point operations (FPOs)
- Thus, for a general MLP the cost in floating point operations of a forward pass is $\simeq N \times (\sum_h H_h \times H_{h-1})$
  - Very fast on GPUs as it theoretically becomes $O(N \times \sum_h H_h)$

The Cost of Computing ∇e^ℓ

- Computing a local gradient ∇e^ℓ in a SHL MLP with square error and one output essentially requires to compute
  - H components for the hidden to output connections, with a $O(1)$ cost each, for $\frac{\partial e^\ell}{\partial w_h^O} = (\widehat{y} - y) o_h$
  - DH components for the input to hidden connections with essentially an $O(1)$ cost each, for $\frac{\partial e^\ell}{\partial w_{h,j}^H} = \left(\sum_{i=1}^L \delta_i w_{ih}^O \right) \sigma^{'}(a_h) x_j$
- In these we are omitting the cost of the partials with respect the bias, but they are much smaller
- Thus, the overall cost of computing ∇e^ℓ on a SHL MLP is essentially $O(DH)$
- More layers with H_h units add a cost $O(H_{h-1} \times H_h)$
  - $O(\sum_h H_h)$ on GPUs

The Cost of Computing ∇e

- For a mini–batch of size N_b, the cost of the mini–batch gradient ∇e of a SHL MLP is $O(N_b \times D \times H)$
- And extra layers add a cost $O(N_b \times H_{h-1} \times H_h)$
- This is of the same order of magnitude than the cost of the forward pass
- This also dominates the $O(H_{h-1} \times H_h)$ cost of updating the (H_{h-1}, H_h) weights in gradient descent

- And all these costs have to be multiplied by the number of training **epochs**

Training Complexity

- The important term in the overall training cost is nEps × cost of ∇e, with nEps the number of epochs
  - One epoch = one pass on the entire sample
- Thus, the **globally dominant term** in a SHL MLP is

  $$\text{nEps} \times \text{cost of } \nabla e = O(\text{nEps} \times N \times D \times H)$$

  with N here the entire sample size
- And for each hidden layer we add an extra cost

  $$O(\text{nEps} \times N \times H_{h-1} \times H_h)$$

- Thus, training many layered, large MLPs can be **very costly**
  - But **linear in sample size** N

Takeaways on MLP's Cost

- MLP complexity is determined by its **architecture** $\{H_h\}$**, training procedure and sample size** N
- The forward and backward MLP passes have basically the **same complexity**
- Their cost per **pattern and layer** is $H_{h-1} \times H_h$
- For gradient descent these costs are **multiplied by the number nEps of epochs**
- First order methods essentially **do not add extra complexity**
- Second order methods add extra per iteration costs but should **require less iterations**
- Single MLP training is **not CPU parallelizable**
- But **GPUs** can greatly improve MLP processing costs