# Neural Networks for Regression and Classification

José Dorronsoro
Escuela Politécnica Superior
Universidad Autónoma de Madrid

March 2019

# Contents

# 1 Machine Learning Modeling Basics

### 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 we must perform
  - Outcome **evaluation**: how good/actionable it 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
- 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 $\ell_k$ 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 Problem**

- This is a first "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

# 2 Basic Regression

### Model Parameterization

- Usually 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**

### 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), SVMs
  - And also: 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 the sample error is

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

  and the population error is

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

  where both end in a simple **quadratic form**

- The regression problem reduces to **minimize** $\widehat{e}_S(W)$, i.e., solve the MSE problem
  - Something in principle well understood in mathematical optimization

**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 $\sigma_N^2$
- 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 $\nu$ 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
  - Forget about it and get back to get models $f$ such that $f \simeq \phi$

**$k$–NN Regression**

- A last try: we will have just one $y^p$ for each $x^p$ but we could hope to **have several $x^p$ close to a new $x$**
- This suggests to fix a number $k$ of 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 which 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 have to modify our first assumption: Predictors that are close should give predictions that are also close, **provided that there are enough of them close by**
- And 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 value
  - 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 $\widehat{e}'(w) = 0$**

- To compute $\widehat{e}'(w)$ we have

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

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

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

  where $X$ and $Y$ denote the $N$ dimensional vectors $(x^1,\ldots,x^N)^t$, $(y^1,\ldots,y^N)^t$

**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

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

- 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 $\widehat{R}\,\widehat{w} - \widehat{b} = 0$
- And obtain the sample–dependent optimal $\widehat{w}^*$ as

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

**Finding Optimal Models**

- 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

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

    over a **mini-batch** $B$ with $n_B$ samples
  - Component wise: $w_i^{k+1} = w_i^k - \rho_k\frac{\partial\widehat{e}}{\partial w_i}(w^k)$
  - $\rho_k$ 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

**Measuring Model Fit**

- First option: **Root Square Error** $RSE = \sqrt{\frac{1}{N}\sum(y^p - \widehat{y}^p)^2}$
- OK, but how good is 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 $R^2 = 1 - \frac{RSS}{TSS}$

**Regularization**

- Our regression solution $\widehat{w}^* = (X^t X)^{-1} X^t Y$ 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 $\widehat{w}^* = (X^t X + \alpha I)^{-1} X^t Y$ minimizes

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

- 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 $\alpha$?

**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**

# 3 Bias, Variance and Cross Validation

**Sample Dependence**

- Important: **everything is sample dependent** for if we change $S$ we get a different model
  - We thus write $\widehat{f}_S(x)$
- Therefore, for different samples $S, S'$ we want our models to verify

$$\widehat{f}_S(x) \simeq \widehat{f}_{S'}(x)$$

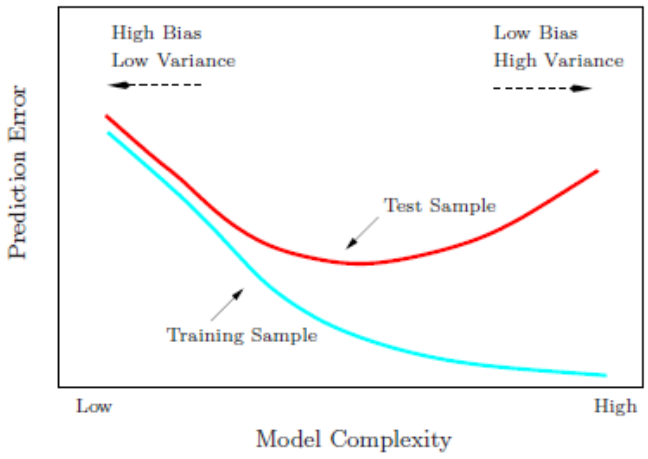
- That is, we want our models to have small **variance** with respect to sample changes

  - Intuitively this can be achieved using simple models with few parameters

- But we also want that $\widehat{f}_S \simeq \phi(x)$
- That is, we want our models to have a small **bias**, i.e., get as close as possible to the "true" model $\phi$
  - Intuitively this can be achieved using highly flexible models with many parameters
- But 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*

**Two Examples**

- In $k$–NN regression 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 $x$
- In Ridge regression the parameter that controls the tradeoff is the regularization penalty $\alpha$
- 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 large bias and small variance
- 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 are likely to be very different, resulting in larger variances

**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" thus 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 $S_1, \ldots, S_M$
  - 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 $\alpha$ in Ridge Regression

**Grid Hyper–parameter Selection**

- Consider for Ridge regression a hyper–parameter range $[0, A]$
  - $\alpha = 0$: no penalty and, thus, small bias and high variance
  - $\alpha = A$: large penalty and, thus, small variance but high bias
- Select an $L + 1$ point **grid** $[\alpha_0, \ldots, \alpha_L]$
  - For instance a uniform one $\alpha_\ell = \ell\, \frac{A}{L}$, $\ell = 0, 1, \ldots, L$

- Build $M$ **folds**: pairs $(S_m, S_m^c)$ and for each $\alpha_\ell$
  - Train $M$ Ridge models on the $S_m^c$ using the hyper–parameter $\alpha_\ell$
  - Average their $M$ validation errors $e_m$ on the $S_m$ to get the CV error $e(\alpha_\ell)$ for $\alpha_\ell$
- Finally choose the (hopefully) optimal hyper–parameter $\alpha^*$ as

$$\alpha^* = \text{arg min}_{0 \leq \ell \leq L} \; e(\alpha_\ell)$$

- $\alpha^*$ gives the model with the **best expected generalization among all possible $\alpha$ 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**

# 4 Data and Model Analysis

**And So What?**

- Key question: what are models for?
  - First answer: to be used to derive new predictions
  - Better answer: to extract knowledge and to make inference on the underlying problem
- In this light, LR models are simple, perhaps not too powerful, but certainly useful
  - They are the first tool to apply in (almost) any problem analysis
- Some questions are easier to answer for them:
  - Which variables do influence the target and which do not?
  - What are the strongest predictive variables?
  - Are there related/redundant variables?
  - Is the feature-target relationship actually linear?

**Issues with LR**

- Before building any model we must perform a prior data analysis to keep under control important issues:

  - **Collinearity**: predictor variables that are redundant
  - **Outliers**: points $(x^p, y^p)$ with a "normal" pattern $x$ but an unlikely target value $y^p$, or vicev-ersa
  - **High-leverage points**: points $(x^p, y^p)$ with an unlikely pattern $x^p$ and a reasonable target value $y^p$

- And after a model is built we **must check** whether its results agree with its assumptions
  - **Linearity** of the response–predictor relationships: if not, the LR will be poor
  - **No correlation of error terms**, i.e. our basic model assumption does hold
  - **No heteroscedasticity**, i.e., no non-constant variance of error terms, that varies on several $x$ regions

**Detecting and Handling Data Issues**

- Before **any** model is built we **must** try detect possible data inconsistencies and/or redundancies
- Outliers: will cause $(x^p, y^p)$ to be far from the line fit or the residual to be out of range
  - Can detect them with **box plots**
- Feature collinearity: look at least at the **correlation matrix** , but harder when multicollinearity present
- Analyze **feature–target scatterplots**; if possible, look also at the two–predictor scatterplots (though there are $d(d-1)/2$ of them)
- High-leverage points: $x^p$ outside the main $x$ range; harder to spot in multidimensional models
- We consider all this over the Boston Housing dataset

**Housing: First Conclusions on the Data**

- Collinearity: some predictor variables may be redundant
  - AGE–DIS: proportion of units built prior to 1940 and weighted distances to five employment centres
  - RAD–TAX: accessibility to radial highways and full-value property-tax rate
  - NOX–INDUS
- Outliers: points $(x^p, y^p)$ with a normal pattern $x$ but an unlikely target value $y^p$
  - ???
- High-leverage points (HLPs): perhaps at variables
  - ZN: proportion of residential land zoned for lots over 25,000 sq.ft.
  - CHAS: 1 if tract bounds Charles river; 0 otherwise
  - B: $1000(Bk - 0.63)^2$, with $Bk$ the proportion of black persons by town
  - But have to look at HLPs as D-dimensional points and not features

**Detecting and Handling Model Issues**

- After the model is built we check whether it supports the basic LR assumptions
- Linearity: **a residual plot should not have any structure**
- Uncorrelated error terms: **residuals do not change rather smoothly**
- Error histograms should be **symmetric and sharp at 0**
- Heteroscedasticity: **residual plots do not show a "funnel" like structure**
- **Always address these possible problems**: if not, we may be fooling ourselves with an untenable model
- Let's build LR models over the Boston Housing data

**Housing: First Conclusions on the Linear Model**

- Recall the first things to look at after LR model building:
  - Linearity of the response-predictor relationships?
  - No correlation of residuals?
  - No heteroscedasticity?
- Linearity of the response-predictor relationships: not bad
  - If perfect fit, $y$ and $\widehat{y}$ in diagonal; here in near diagonal
- Correlation of residuals only for large targets
  - Perhaps we should think about two separate models
- No heteroscedasticity, i.e., constant variance of residuals
  - No funnel appears in target-residual representation but there is still a bias
- Build a second model?

**Takeaways on Data and Model Analysis**

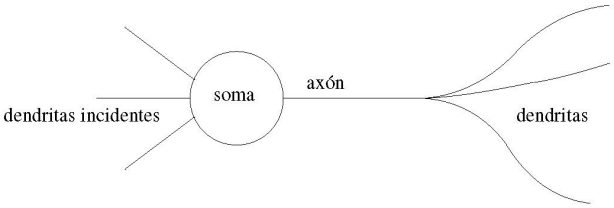
1. Before any model building we must **analyze and understand our data**
2. We must understand the **assumptions our model implies on the data**
   - If they aren't true the model won't be very good
3. This **must be checked** after the model is built
4. LR are the **first models to build**, to have a benchmark and to better understand the problem and its data
5. And, morever, we must **always**
   - Tune the hyperparameters for our models
   - Try out many different models
   - Explore several feature representations for our data

# 5 Multilayer Perceptrons

## 5.1 Classical MLPs

### Basic Neural Models

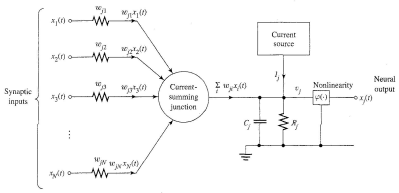
- Basic model: Ramón y Cajal's neuron (1900)




- 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 7000 connections, often recurrent

### 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

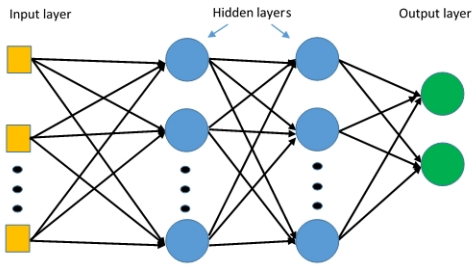
- In Artificial Neural Networks (and in ML) one considers just Q1 and (usually) outside any neuro-computational framework

**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 unit connections 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** 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 of its activation $o_i^h = \varphi(a_i^h)$
- In vector form:

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

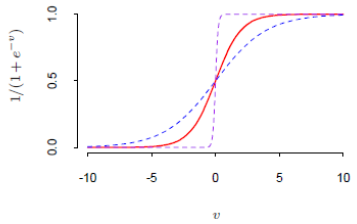
  where $\varphi$ is applied over each unit

### 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 and Hyperbolic Tangent

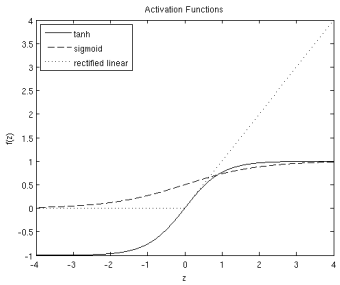
- Sigmoid and tanh: smooth version of Heaviside step function



- Classical choices:
  - Hyperbolic tangent for hidden units
  - Linear outputs for modelling (and sometimes) classification problems
  - Sigmoid outputs for classification problems (as in logistic regression)

**ReLUs**

- ReLU transfer function: $r(x) = \max(0, x)$




From Stanford's UFLDL Tutorial

- We have $r'(x)$ either 0 or 1 (hoping $x = 0$ never happens!!)
  - Many gradient elements will go to 0
  - Many units (neurons) will be "dead", with a 0/constant activation no matter the input

**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 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 $o$,

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

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

**The Simplest MLP II**

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

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

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

- 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

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

**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 $\phi$, 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

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

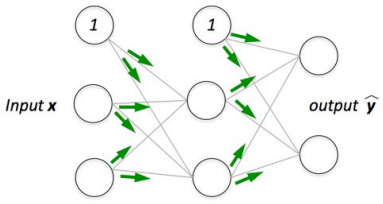
- We have therefore two tasks:
  - Compute $\nabla e$
  - Exploit it to build MLPs
- We will exploit $\nabla e(\mathcal{W})$ through **optimization methods** after we compute it

**SHL Forward Pass I**

- We 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
  - $a = W^H x + b^H$, $o = \varphi(a)$,
  - Or unit–wise: $a_h = \sum W^H_{hi} x_i + b^H_i$, $o_h = \varphi(a_h)$
  - $y = w^O \cdot o + b^O = \sum_h w^O_h o_h + b^O$
- 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

**Computing the Gradient**

- If $w_{ij}$ is the weight connecting unit $j$ to unit $i$, we have

$$\frac{\partial e^\ell}{\partial w_{ij}} = \frac{\partial e^\ell}{\partial a_i}\frac{\partial a_i}{\partial w_{ij}} = \frac{\partial e^\ell}{\partial a_i} o_j = \delta_i o_h$$

- In the output layer $e^\ell = \frac{1}{2}(y - \widehat{y})^2$ and $a^O = \widehat{y}$, and thus,

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

- 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 I**

- 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, therefore, we have

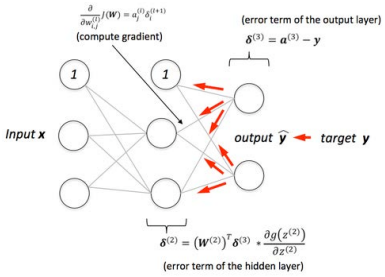
$$\frac{\partial a^O}{\partial a_h^H} = \frac{\partial a^O}{\partial o_h^H}\frac{\partial o_h^H}{\partial a_h^H} = \frac{\partial a^O}{\partial o_h^H}\varphi'(a_h^H) = w_h^O \varphi'(a_h^H)$$

- Moreover, $o_j^D = x_j$ and, therefore,

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

**SHL Gradient Backprop II**

- Graphically we have the following scheme:




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

**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 will use 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. Once an MLP is defined (feedforward and backward passes), MLP training reduces to a (usually difficult and costly) **optimization problem**

## 5.2 Unconstrained Smooth Optimization

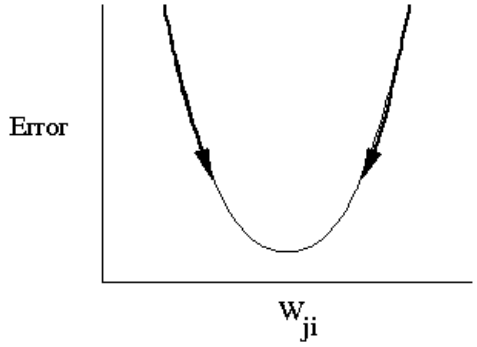
**Back to Optimization**

- 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

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

- $\rho_k$ is the **learning rate** (LR)
- With a small $\rho_k$ 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 $\rho_k^*$ 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 $\nabla e$

**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 $\Delta w$ 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 $\rho_k$ 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 $\nabla f$
- 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

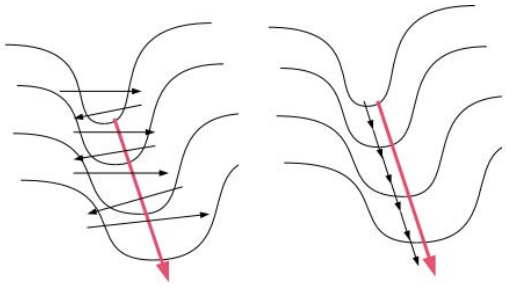
$$\mathcal{H}_{(i,j)(i,j)}(w) = \left(\frac{\partial f}{\partial w_{ij}}\right)^2$$

**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

**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

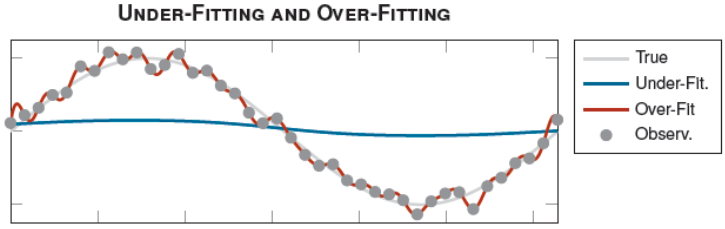
**Takeaways on MLP Optimization**

- **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

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

**Regularized Algorithms**

- The preceding methods apply straightforwardly to $e_R$
- Gradient descent becomes

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

- Newton steps are now

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

- And the Gauss–Newton approximation to $\mathcal{H}_R(w)$ is

$$\mathcal{H}_R(w) = \mathcal{H}(w) + \lambda I \sim E[\nabla f(w) \nabla f(w)^\tau] + \lambda I$$

  that is **definite positive** and, hence, **invertible**

**How to choose $\lambda$**

- Again, the correct choice of $\lambda$ 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 $\lambda$ is chosen by using CV by exploring a discrete set of values $\lambda_j$,
- The same can essentially done for any other hyper–parameter: number of hidden layers? number of hidden units? learning rate?? minibatch size???

**MLP Ensembles**

- Recall that $e(w)$ 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
- This suggests
  - To start from $K$ independent initial weights and get $K$ optimal weight sets $w_k^*$

  - To 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 $\epsilon_k^p$ 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**

## 5.3 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 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
- 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 $\nabla e^\ell$**

- Computing a local gradient $\nabla e^\ell$ 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 partials with respect the bias, but they are much less
- Thus, the overall cost of computing $\nabla e^\ell$ 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 $\nabla e$**

- For a mini–batch of size $N_b$, the cost of the mini–batch gradient $\nabla e$ 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 $\times$ cost of $\nabla e$, 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 an extra cost

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

  for extra hidden layers
- 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

# 6 Basic Classification

## 6.1 The Classification Model

### 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 $\omega$ 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 $\omega$ and becomes a random variable
- A $\omega$ has a **prior probability** $\pi_m$ 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 $\pi_m$ 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

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

- 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** $\delta_B$

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

- With some extra work we can show that this **Bayes Classifier** $\delta_B$ 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 $\pi_m$ or (much harder) $f(x|m)$

**Approximating the Bayes Classifier**

- To define $\delta_B$ we need to know the prior probabilities $\pi_m$ and the prior densities $f(x|m)$
- A reasonable choice for $\pi_m$ 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**

## 6.2 Nearest Neighbor Classification

**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 $\delta_{kNN}(x) = \arg\max_m n_m(x)$

- 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
  - $P(C_m \cap B_r(x)) \simeq \frac{n_m(x)}{N_m}$
  - 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 $\delta_{kNN}$ might be close to $\delta_B$, 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!!

## 6.3 Measuring Classifier Accuracy

### True/False Positives/Negatives

- 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 $\widehat{y} = 0$
- The standard way of presenting these data 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

### 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
- Ideal classifier: high recall, high precision (i.e., effective and efficient!!)

# 7 Logistic Regression

**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 $\delta_0$ 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 (LR)**

- 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 LR 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

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

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

**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

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

- 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 $\ell$ at $w^k$, which may or may not be invertible
  - Everything is fine if the $w^k$ are close enough to the optimum $^*$ 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 $S = \{(x^1, y^1), \ldots, (x^N, y^N)\}$, we define a **sample dependent loss function**

$$L(w) = L(w|S) = L\left(y^1, \ldots, y^N, f(x^1; w), \ldots, f(x^N; w)\right)$$

- $L(w)$ is often minimized from some $w^0$ by **iterations**

$$w^{k+1} = w^k - \rho_k G(w^k, S)$$

  with $\rho_k$ 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 **mini-batch learning**

# 8 Multiclass Log–Loss NN Classification

### 1–hot Encoding and Posteriors

- The standard labelling of multiclass problems is the 1–hot encoding of class $k$ by the vector $e_k = (0, \ldots, \underbrace{1}_{k}, \ldots, 0)$
- Then if $x \in C_k$, its label $y = (y_1, \ldots, y_K)^t$ is $e_k$ and

$$P(k|x) = P(k|x)^1 = P(k|x)^{y_k} = \prod_{c=1}^{K} P(c|x)^{y_c}$$

- Then for a sample $S = \{x^p, y^p = e_{c(p)}\}$ and a posterior probability model $P(c|x, w)$, the probability of getting $S$ is

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

### The Cross Entropy Loss

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

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

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

$$L(w) = -\ell(w; S) = -\sum_{p=1}^{N} \sum_{c=1}^{K} y_c^p \log P(c|x^p; w)$$

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

### MLPs for Classification

- We consider an input layer and a number of hidden layers
- Targets are now the 1–hot encodings of the 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** $\sigma_j(x) = \frac{e^{w_j \cdot x}}{\sum_1^K e^{w_k \cdot x}}$
- For two classes this becomes $\sigma_1(x) = \frac{e^{w_1 \cdot x}}{e^{w_0 \cdot x} + e^{w_1 \cdot x}} = \frac{1}{1 + e^{(w_0 - w_1) \cdot x}}$

  - We thus get the **sigmoid activation** of Logistic Regression
- NN training is again reduced to the minimization of a function, now the log–loss
- And essentially all the previous discussion on MLP regression carries over to classification

**What's New from Regression?**

- Some things change from regression, some don't
- We should check feature correlations: they will affect most models
  - Scatter plots $(x_i, y)$ are usually less informative
- Important: **positive and negative–class feature histograms**
- The **bias–variance trade–off** is subtler in classification
- Accuracy, recall, precision are the usual model quality measures
  - But NN training does not directly minimize them
- We use CV with **stratified folds** to estimate generalization performance
- We also use CV for hyperparameter estimation, as regularization will also be needed
  - In LR we should minimize $-\ell(w_0, w; S) + \frac{\alpha}{2}\|w\|^2$

**How to Handle Posterior Probabilities**

- If possible, we don't want labels as model outputs but **posterior probabilities**
- Most models give them as pairs

$$(\widehat{P}(0|x), \widehat{P}(1|x)) = (\widehat{P}(0|x), 1 - \widehat{P}(0|x))$$

- In principle we would decide 1 if $\widehat{P}(1|x) > 0.5$ and viceversa, but this may be too crude
- It may be advisable to set a **decision threshold** $0 < \theta < 1$ and decide 1 if $\widehat{P}(1|x) > 1 - \theta$ and 0 if $\widehat{P}(1|x) < \theta$
- For **imbalanced** problems where $\pi_0 \gg \pi_1$ (usually the interesting ones) we would have $\widehat{P}(1|x) \simeq 0$ for most $x$
  - In this case we may choose another $\theta \ll 0.5$ and **suggest** 1 if $\widehat{P}(1|x) > \theta$

**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 **Logistic Regression** and the numerical minimization of its (minus) log–likelihood
4. We have introduced several measures of **classifier performance**
5. We have reviewed some **practical issues** of classification