Neural Networks for Regression and Classification

José Dorronsoro
Escuela Politécnica Superior
Universidad Autónoma de Madrid

March 2018

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
- ML is in the middle of the global process chain but also in the middle of some subchains

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
  - Sometimes as a first step towards a supervised model

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 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 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

Wind Energy Forecasting

- This is a second, real regression problem
- We want to estimate the hourly energy production of a wind farm from NWP variables which we believe relevant

- The **features** are the NWP variables
  - U, V surface wind components
  - U, V 100–meter wind components
  - Temperature
  - Pressure
  - . . .
- The **target** is the energy produced during the outgoing hour

The ML Cycle in Wind Energy

- Raw data: historic wind energy production data plus NWP files from weather forecasters
  - Possibly huge files with special formats
  - We have to extract the relevant NWP information, organize them in a suitable way and pair it with the energy data
- The ML core: whatever set of (non–linear) regression algorithms which you may think useful
- After ML is finished
  - Collect, organize and save the different model outputs
  - Select one single model output or some combination (more ML) of them as your system's output
  - Compute uncertainty estimates
  - Combine your outputs with someone's else
  - And keep up the entire process

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 $\mathcal{F}$
- **Parametric** models have a fixed functional form $f(x) = f(x; W)$
- Simplest example: linear regression, where $M = d$ 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; NN models

Issues in Model Building

- There are two initial questions when working with models from a given family $\mathcal{F}$:
  - How do they operate?
  - How we do build them?
- In turn, these two 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 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 $\widehat{e}_S(f) = \widehat{e}_S(W)$
- The problem to solve becomes

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

- In linear regression

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

  which ends up 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 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

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

- Thus for any model we have $\mathrm{mse}(f) \geq \sigma_N^2$ always
- And we should focus on achieving on $f \simeq \phi$

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
  - 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

- Are we done? Not at all!!
- 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 by unit interval
  - But when $d = 3$ we have just 1 pattern per unit of volume
  - 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
- 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 $e(w)$

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

- The problem 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

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

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

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

  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

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

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

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

Solving the Linear Equations

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

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

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

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

  i.e., $\widehat{R}$ is the **sample covariance matrix**
- If $\widehat{R}$ is invertible, we just solve the linear system $\widehat{R}\, \widehat{w} - \widehat{b} = 0$
- And obtain the sample–dependent optimal $\widehat{w}^*$ as

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

Finding Optimal Models

- For general regression models it may not possible to solve analytically the equation $\nabla \widehat{e}(W) = 0$
  - For LR and big data, covariance matrices over large datasets or dimensions may not be computed
  - Numerical methods are needed
- The simplest numerical alternative is **gradient descent**:

  - Starting from some random W^0 we iteratively compute

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

  - 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

Measuring Model Fit

- First option: **Root Square Error** $RSE = \sqrt{\frac{1}{N}\sum(y^p - \widehat{y}^p)^2} = \sqrt{\frac{1}{N}RSS}$
- OK, but how good is this? We must always have a **base model** to benchmark our results
- Simplest "model": the mean $\overline{y} = \frac{1}{N}\sum_1^N y^p$, with square error

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

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

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

- The widely used R^2 coefficient is simply $R^2 = 1 - \frac{RSS}{TSS}$

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 $\alpha > 0$
- To make this practical, note that $\widehat{w}^* = \left(X^t X + \alpha I\right)^{-1} X^t Y$ minimizes

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

- 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**

3 Bias, Variance and Cross Validation

Sample Dependence

- Important: **everything is sample dependent** for if we change S we get a different model
- We get sample–dependent weights $\widehat{W} = \widehat{W}_S$ and model $\widehat{f}_S(x) = \widehat{f}(x; \widehat{W}_S)$
- We must control their dependence on the concrete S sample used to build it
- Moreover, we must apply our model on new, **unseen** samples
- We must have a sample generating procedure that ideally gives homogeneous samples and a robust model building methodology
- Both together should (reasonably) guarantee that, for two S, S',

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

Sample Bias and Variance

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

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

  - The expectation $E_S[\widehat{f}_S(x)]$ is taken over all possible 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

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

Bias Versus Variance

- 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 with rich, highly flexible models
  - Or with essentially no regularization

- But we would also like to have a model such that

$$V_N(x) \simeq 0,$$

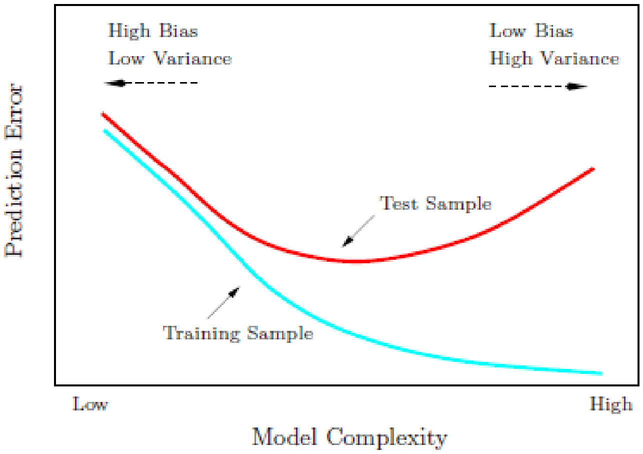
  i.e., a model with small **variance** $V_N(x)$

  - This should be achievable with simple models with few parameters
  - Or with more severe regularization

- 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 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 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
  - Compute $\widehat{f}_M(x) = \frac{1}{M}\sum_m \widehat{f}_{S_m}(x) \simeq \widehat{f}_N(x)$
  - 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 apply **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 α 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**

$$G = \left\{ 0, \frac{A}{L}, \frac{2A}{L}, \ldots, \frac{\ell A}{L}, \ldots, \frac{LA}{L} = A \right\}$$

- Build M **folds**: pairs (S_m, S_m^c) and for each $\alpha_\ell = \frac{\ell}{L} A$, $0 \le \ell \le L$
  - Train M Ridge models on the S_m^c using the hyper–parameter α_ℓ
  - Average their M validation errors e_m on the S_m to get the CV error $e(\alpha_\ell)$ for α_ℓ
- Finally choose the (hopefully) optimal hyper–parameter α^* as

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

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 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 viceversa
  - **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 if 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
- Feature collinearity: look at least at the correlation matrix
- Analyze feature–target scatterplots; if possible, look also at the two–predictor scatterplots (though there are $d(d-1)/2$ of them)
- 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
- 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 blacks 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

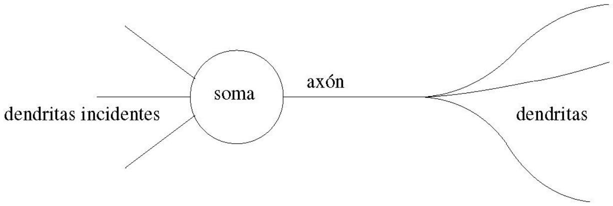
2. We must understand the assumptions our model implies on the data
    - If they aren't true the model won't be very good
3. This must be checked after the model is built
4. LR models are simple but their assumptions are of interest to any other model
5. LR are the first models to build, to have a benchmark and to better understand the problem and its data
6. And
    - Always tune the hyperparameters for our models
    - Always try out many different models
    - Always explore several feature representations for our data

5 Multilayer Perceptrons

5.1 At the Beginning ...

Basic Neural Models

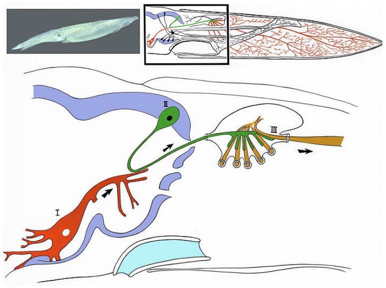
- Basic model: Ramón y Cajal's neuron (1900)



- 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

Hodgkin–Huxley

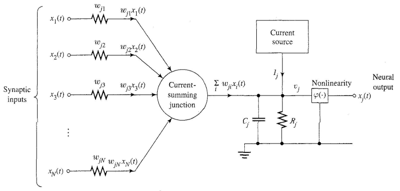
- They developed (circa 1935) the first model to describe the generation and propagation of electrical **action potentials** in neurons



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

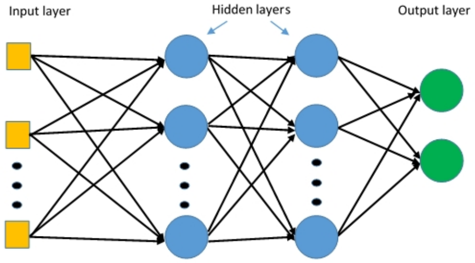
Basic Questions

- Q1: How to adjust the w_j and I values? 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!!
    - Because of the difficulty of measuring the joint behavior of groups of neurons
    - Because of taking into account the recurrence present in real neurons
- In Artificial Neural Networks (and in ML) one considers just Q1 and outside any neurocomputational framework

5.2 Classical MLPs

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
- 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
- Combined effect of sucesive layers: potentially highly non–linear transformation

Unit Activation and Output

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

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

  where f is applied over each unit

Activation Functions

- Choices for f:
  - Heaviside (in Rosenblatt's Perceptrons): $\varphi(a) = 0$ if $a \leq 0$, $\varphi(a) = 1$ if $a > 0$
  - Identity/linear: $\varphi(a) = a$
  - Sigmoid:

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

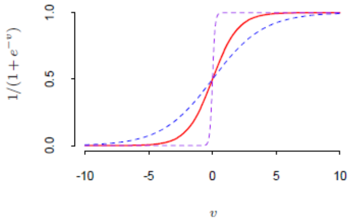
  - Hyperbolic tangent:

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

  - Rectified Linear Units (ReLUs): $\varphi(a) = r(a) = \max(0, a)$

Sigmoid and Hyperbolic Tangent

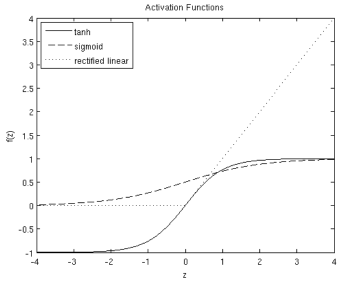
- Sigmoid and tanh: smooth version of Heaviside step function



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

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

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 or several linear or sigmoid outputs (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: assuming 1–dimensional targets, we have for the outputs $\widehat{y}$

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

  - In vector form: $\widehat{y} = w^O \cdot o + b^O$
- 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

- We say that $\mathcal{F} = \{f(x; W)\}$ is a **Universal Approximation Family** over a domain $\mathcal{R}$ if

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

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

- Notice that Universal Approximation is just what we need in regression
- In fact a **Single Hidden Layer (SHL) 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

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

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

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

MSE Gradient

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

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

  for we have

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

- We have therefore two tasks:
  - Compute ∇e
  - Exploit it to build MLPs

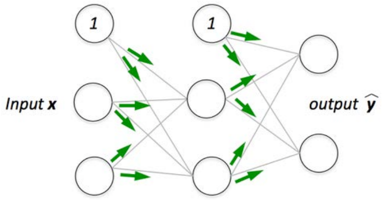
- We will exploit $\nabla e(W)$ through **optimization methods** after we compute it

SHL Forward Pass I

- We apply the preceding to a single hidden layer 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 $w^O = (w^O_1, \dots, w^O_H)$ 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

SHL Generalized Errors

- In general we have

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

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

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

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

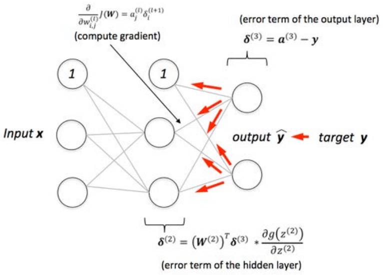
$$\delta^H_h = \frac{\partial e^\ell}{\partial a^H_h} = \frac{\partial e^\ell}{\partial a^O}\frac{\partial a^O}{\partial a^H_h} = (\widehat{y} - y)\frac{\partial a^O}{\partial a^H_h} = \delta^O \frac{\partial a^O}{\partial a^H_h}$$

SHL Gradient Backprop I

- In the output layer we have
$$\frac{\partial e^\ell}{\partial w_h^O} = (\widehat{y} - y)\frac{\partial a^O}{\partial w_h^O} = (\widehat{y} - y)o_h^H$$
- In the hidden layer 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:


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

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

Takeaways on Clasical MLPs

1. They have a layered structure with outputs computed in a **forward pass** using differentiable activations
2. Usual activations: 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; cross entropy is used 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**

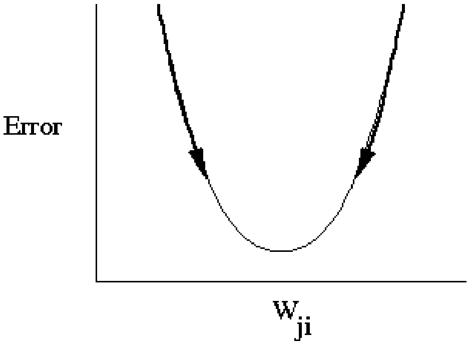
5.3 Unconstrained Smooth Optimization

Back to Optimization

- General optimization theory is a key tool in Machine Learning (ML)
- There are two optimization set ups in ML
  - **Unconstrained** optimization, slightly simpler and that of 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 recall that $-\nabla e$ 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

Newton's Method

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

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)
  - We can also work with a line minimization version of 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

$$e'(w) = \int (f(x;w) - y) \frac{\partial f}{\partial w}(x,y) p(x,y) dxdy$$

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

$$\begin{aligned} e''(w) &= \int \left(\frac{\partial f}{\partial w} \right)^2 p(x,y) dxdy \\ &\quad + \int (f(x;w) - t) \frac{\partial^2 f}{\partial w^2}(x,y) p(x,y) dxdy \\ &\simeq \int \left(\frac{\partial f}{\partial w} \right)^2 p(x,y) dxdy > 0 \end{aligned}$$

- 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**
- 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 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

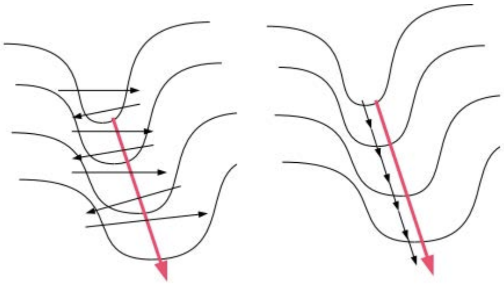
- A possibility on the error surface of a NN is to have many long, narrow ravines
  - Gradients bounce in the narrow section, but may be very small along the wider one

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.,

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

- 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 $\tilde{\Delta}^{k+1} = -\rho_k \nabla_W e(W^k) + \mu_k \tilde{\Delta}^k$ and
  2. Apply
  $$W^{k+1} = W^k + \tilde{\Delta}^{k+1} = W^k - \rho_k \nabla_W e(W^k) + \mu_k \tilde{\Delta}^k$$
- **Nesterov's Accelerated Gradient** is a variant of this
  $$\tilde{\Delta}^{k+1} = -\rho_k \nabla_W e(W^k + \mu_k \tilde{\Delta}^k) + \mu_k \tilde{\Delta}^k; \; W^{k+1} = W^k + \tilde{\Delta}^{k+1}$$
- In convex optimization it improves GD and is highly recommended 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

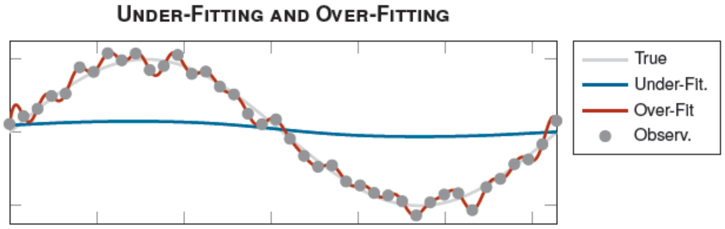
- **Gradient descent** is the simplest method but possibly also the slowest

- **Momentum** can be used to speed it up
- **Newton's method** is the fastest 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

5.4 Revisiting Bias–Variance

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 $e(W)$ that depends on and grows with $\|W\|$
- 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

- Here too 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:
  - We expect $e(W^*) \simeq \frac{\lambda}{2}\|W^*\|^2$ and, hence, $\|W^*\|^2 \lesssim \frac{2e(W^*)}{\lambda}$
  - In fact, the regularized loss can be seen as the Lagrangian of the constrained problem $\min_W e(W)$ subject to $\|W\|^2 \leq \rho$ for some $\rho > 0$

- 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 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:
  - Exploring a discrete set of values λ_j,

  - That fall in a preselected range $[\Lambda_0, \Lambda_1]$,
  - Using **cross validation** (CV)

- The same is essentially done for any other hyper–parameter

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 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 minibacth selection
- They do not thus converge to a single optimum
- MLP ensembles can take advantage of this
- They imply extra costs but ensembles are embarrassingly parallelizable

5.5 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 cores, or use 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 parallelizable:
  - There is a sequential layer dependence in Backprop
  - It is the same for training iterations

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: adaptive or **on line**, i.e., pattern by pattern, weight updates
  - Not used today: Currently medium–to–large NNs are trained using **mini–batches**
  - But allows a relatively simple setting for a theoretical analysis

MLP Complexity

- MLPs are fast to apply but costly to train
- How can we train MLPs over large sample sizes/dimensions?
- MLP traning cost is determined by
  - Its **architecture**, that determines the number of weights to fit and that is usually 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, L outputs and one hidden layer with H units is

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

  - In regression, $L=1$, so the weight number is $O(DH)$ for a regression SHL MLP
- Each extra $H_1 \times H_2$ hidden layer adds $(H_1+1) \times H_2 \simeq H_1 H_2$ weights
- For a general MLP the cost in FP operations of a forward pass is $\simeq N \times (\sum_h H_h \times H_{h-1})$
  - Very fast on GPUs

The Cost of Computing ∇e^ℓ

- Computing a local gradient ∇e^ℓ in a SHL MLP with square error and L outputs requires to compute
  - LH components for the hidden to output connections, with a $O(1)$ cost each, for $\frac{\partial e^\ell}{\partial w^O_{ih}} = (\widehat{y}_i - y_i) o_h$
  - DH components for the input to hidden connections with essentially an $O(1)$ cost each, for $\frac{\partial e^\ell}{\partial w^H_{h,j}} = \left(\sum_{i=1}^L \delta_i w^O_{ih} \right) \sigma^{'}(a_h) x_j$
- The overall cost of computing ∇e^ℓ is thus

$$O(LH + DH) \simeq O(DH)$$

  for usually $L = O(D)$
- More layers with H_h units add a cost $O(H_{h-1} \times H_h)$

The Cost of Computing ∇e

- For a mini–batch of size N, the cost of the mini–batch gradient ∇e of a SHL MLP is $O(N(DH + HL))$
- And each extra layer adds a cost $O(N \times H_{h-1} \times H_h)$
- 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

  $$\text{nEps} \times \text{cost of } \nabla e = O(\text{nEps}\ N(DH + HL))$$

  with N here the entire sample size
- And an extra cost

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

  for each extra hidden layer
- Thus, training many layered, large MLPs can be **very costly**

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 will require less iterations
- Single MLP training is not easily parallelizable
- 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
  - Given that there are infinitely many such approximations, closeness is a natural quality criterion
- 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 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

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

- 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

- Thus, we should decide according to a **classifier** function 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

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 $\delta_{kNN}(x) = \text{arg max}_m \ n_m(x)$
- 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
  - $P(C_m \cap B_r(x)) \simeq \frac{n_m(x)}{N_m}$
  - 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 δ_{kNN} should be close to δ_B, for

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

Some k–NN Issues

- **Q1: How do we choose k?** Using CV, of course
- There are no closed form solution and we have to balance again 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?**
- 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**

The Curse of Dimensionality

- This consideration reflects the **curse of dimensionality**:

  Even for low dimensions and large samples, **the sample space is essentially empty**
- 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 class counting will not be meaningful
- Therefore, unless we deal with violently non–linear classification problems, a simple model such as logistic regression may be better than k–NN for moderate dimensions

7 Classification with MLPs

7.1 Logistic Regression

Linear Regression for Classification?

- k–NN Classifier is simple but also crude; have to look elsewhere
- Building a regression model with targets 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!!!
- We know that our goal should be to estimate $P(j|m)$; let's try to attain it!

Logistic Regression (LR)

- We assume
$$P(1|x) = \frac{1}{1+e^{-(w_0+w\cdot x)}}$$
  - 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 $L(w)$ to minimize for which we use the sample's **likelihood**

Sample's Likelihood

- Assume a sample $S = \{(x^p, y^p)\}$, with y^p either 1 or 0
- If the $Y = \{y^p\}$ labels are derived **independently** from a LR model with weights w_0, w applied to the $X = \{x^p\}$, 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 $y^p = 1$, $P(1|x) = P(1|x^p)^{y^p} \;\; P(0|x^p)^{1-y^p}$ and
  - If $y^p = 0$, $P(0|x) = P(1|x^p)^{y^p} \;\; P(0|x^p)^{1-y^p}$

Max Log–Likelihood Estimation

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

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

- The optimal $\widehat{w}_0^*, \widehat{w}^*$ should have given us the likeliest sample which makes it sensible to estimate them 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, $\nabla\ell(W) = \nabla\ell(w_0, w) = 0$ 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
- 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

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 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 **mini-batch learning**

7.2 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

$$P(Y|X;W) \;=\; \prod_1^N P(c(p)|x^p;W) = \prod_1^N \prod_{c=1}^K P(c|x^p;W)^{y_c^p}$$

The Log–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 **log loss** (or cross–entropy) is now simply $-\ell(W;S)$, 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

8 Practical Classification

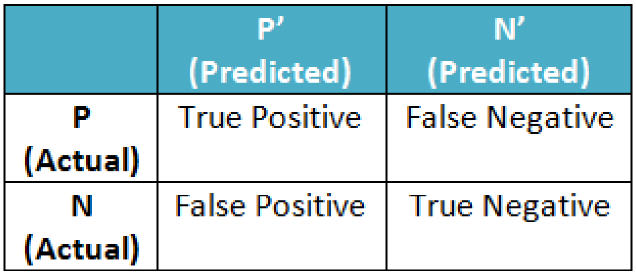
8.1 Measuring Classifier Accuracy

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 $acc = \frac{TP+TN}{N}$
- acc is the first thing to measure but it may not be too significant: if the number N_0 of negatives is $\gg N_1$, the number of positives
  - 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!!)

8.2 Practical Issues

What's New from Regression?

- Some things change from regression, some don't
- We should check feature correlations: they will affect most models
- Important: **positive and negative–class feature histograms**
  - Scatter plots (x_i, y) are usually less informative
- The **bias–variance trade–off** is subtler in classification
- Accuracy, recall, precision are the usual model quality measures
- 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 confidence threshold $\kappa < 0.5$ and decide 1 if $\widehat{P}(1|x) > 1 - \kappa$ and 0 if $\widehat{P}(1|x) < \kappa$
- 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 $\theta < 0.5$ 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 several measures of classifier performance
4. We have introduced Logistic Regression and the numerical minimization of its (minus) log–likelihood
5. We have introduced and analyzed some classification metrics
6. We have reviewed some practical issues of classification

9 Deep Networks

9.1 From MLPs to DNNs

NN's Second Spring

- There was a very intense academic interest in the (by now) standard MLPs in the 1990's
  - Several NN conferences and journals appear
- MLP working and training became well understood
  - Although losing much of neuronal plausibility
- MLPs found relevant applications in many fields
  - They were incorporated into data science tools and products
  - Although hyperparameter selection was (is) costly and had (has) to be done very carefully

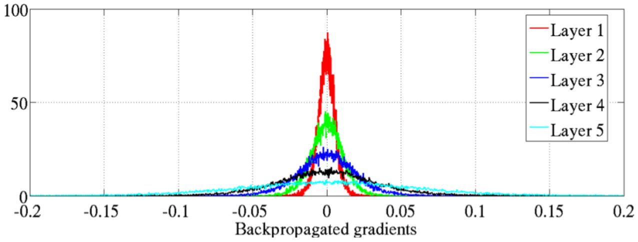
NN's Golden Autumn?

- This went on strongly until the late 90's when
  - New relevant contributions decreased
  - New competitors appeared: Boosting, SVMs, Random Forests, Gradient Boosting Regression, ...
- A nagging issue were deeper MLPs
  - One hidden layer MLPs were enough for most applications
  - But nobody knew how to train MLPs with three or more hidden layer

Vanishing Gradients

- One main obstacle was **vanishing gradients**:

- Consider the weight distribution in a 5 layer MLP



From Glorot & Bengio, AISTATS 2010

- Gradients in the last (5–th) layer are nonzero but vanish as we go back towards the first layer
- Training ceases to have any effect and learning stalls at an early, bad minimum

Towards Deep Networks

- Deep Nets: (initially) standard MLPs with 3 or more layers, either fully connected or **convolutional**
- Training impossible even in early 2000:
  - Poor results over limited HW
  - Addressable problems better solved by single layer nets
- First breakthrough around 2007: deep MLP **unsupervised pretraining** using stacked RBMs (Hinton) or autoencoders (Bengio)
- Easier fine-tunning afterwards by standard backprop

The Boom

- Interest in NNs was rekindled and around 2010 the floodgates opened:
  - Large nets with huge number of weights
  - New convolutional layers, regularizations, initializations or activations
  - New techniques appear ... that are not that different from the old ones
- **New mood**: what was impossible before is now much easier and leads to better results
- Major breakthroughs were achieved in significant problems in computer vision and speech recognition

What Is New In DNNs?

- New and fancy network structures:
  - Convolutional layers (with non–differentiable components)

  - More flexible feedforward connections

- **Automated symbolic backprop derivation**
- Network size: huge number of weights
- Very large sample size (sometimes)
- New cost functions
- New (non differentiable) activations: ReLUs
- New regularization: **dropout**, dropconnect
- Recognition that a good weight initialization is critical

Changes In DNN Training

- Some things have to change:
  - Batch training becomes unfeasible for huge samples/networks
  - Strict online learning may become impractical as single patterns may get lost in a huge network
- Minibatch training balances these extremes:
  - Choose a minibatch size M (a new DNN parameter?) and at each iteration randomly select M sample patterns
  - Perform SGD or some variant over the minibatch
  - Or even a second order method such as CG

Changes In DNN Training II

- But others do not
- Backprop is still the backbone of gradient computation
  - But it is no longer programmed but derived automatically by **symbolic differentiation**
  - Easily extended to convolutional layer weights
  - Imaginatively extended to non–differentiable elements: just pretend that they are so!
- Minibatch–based Stochastic Gradient Descent (SGD) still is the primary optimization approach
- And several hyper–parameters may still have to be chosen, with no clear cut procedures

Training Time and Technology

- Training time is a key issue as it usually shoots up:
  - Samples may be very large
  - Networks may be huge even for moderate samples

  - More hyperparameters may have to be optimized

- Heavy duty computing needed:
  - Multicore machines: very handy for hyperparameter selection; less so for single network tranining
  - Same true for cloud computing environments
  - GPUs: crucial for single network training

- Best: machines/computing centers with many GPUs

Ad Hoc Programming Tools

- Do–it–yourself programming no longer possible
- Increasingly better tools are being available with very fast evolution
- Initially PyLearn+Theano
- Next Caffe: C++ base with Python interface
- Now Keras: Python platform capable of running on top of Theano and Google's TensorFlow
- Plus open releases by large companies
  - Google‘s TensorFlow (plus TensorBoard)
  - Facebook's Torch, on top of the Lua language
  - Twitter's Autograd for Torch (improving its automatic differentiation capabilities)

Deep Nets as DAGs

- The layers of a feedforward net are nodes in a linear graph
  - Backprop is straightforward on such a graph
- But it is also very easy in nets with layers in a DAG
  - They connect different input nodes to different outputs at varying depths and with different layer processing
  - The backprop path is also straightforward
  - And the backprop components at each layer node can be "collated" to the full network gradient
- We just "program" the DAG net defining layer nodes and connecting them in a DAG
  - Then a compiling step yields the forward pass and the backward gradient
- End result: fairly fancy networks
  - Perhaps useful; certainly very costly to train

And Much Better Technologies

- Advanced hardware is a must:
  - GPUs, multicore machines, cloud
- High–level programming:
  - Python as data preprocessing/pipelining + DNN model definition + experimental setup setting + results visualization
  - Python based high–level layers to symbolic GPU backends: Keras (coding in Python), TensorBoard (point and click?)
  - Git as the code and ideas exchange tool
- New skills in high demand, perhaps having more to do with advanced systems handling than with ML
- To read on: M. Nielsen's Neural Networks and Deep Learning online book

9.2 Advanced Techniques for DNN Training

Initialization

- If layers with M_i units used, the standard procedure Glorot–Bengio (`xavier`) is

$$W_i \sim U\left[-\frac{\sqrt{6}}{\sqrt{M_i + M_{i+1}}}, \frac{\sqrt{6}}{\sqrt{M_i + M_{i+1}}}\right]$$

- It ensures $Var\left(\frac{\partial e}{\partial w_i}\right) \simeq Var\left(\frac{\partial e}{\partial w'_i}\right)$ across successive layers when $\tanh$ activations are used
- Gradient vanishing is thus avoided
- Pretraining no longer indispensable (at least for large training data sets)

Dropout Regularization

- The extremely large weight numbers of Deep Neural Networks (DNNs) make regularization mandatory
- First choice: standard Tikhonov regularization (i.e., **weight decay**) for regression DNNs with linear output units
- **Dropout** in other fully connected layers, replacing standard output processing $o_i^\ell = f(a_i^\ell) = f(w_i^\ell o^{\ell-1} + b_i^\ell)$ by

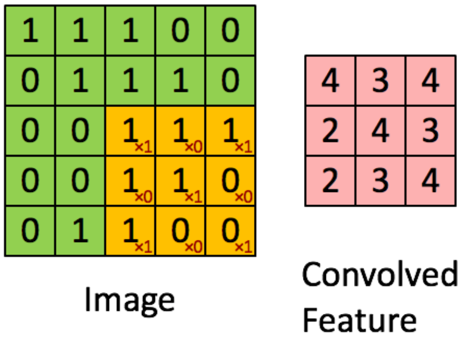
$$o_i^\ell = f(a_i^\ell) = f\left(w_i^\ell (o^{\ell-1} \odot r^\ell) + b_i^\ell\right),$$

  with each r_j^ℓ being 1 with probability p

  - It somehows sub-samples a larger network at each layer
  - Output errors are backpropagated but the final optimal weights w^* are downscaled as $w^*_f = pw^*$
  - It adds **randomness** to the final DNN model (and some **independence** for two different models)
  - Output errors are backpropagated but the final optimal weights w^* are downscaled as $w^*_f = pw^*$

Convolutional Layers

- Starting assumption: patterns organized in features having a one–, two– or multi–dimensional structure
- Basic processing: to apply a $K \times K$ convolutional filter w over an image patch x_j as $y_j = f(w * x_j + b)$



1	1	1	0	0
0	1	1	1	0
0	0	1 ×1	1 ×0	1 ×1
0	0	1 ×0	1 ×1	0 ×0
0	1	1 ×1	0 ×0	0 ×1

4	3	4
2	4	3
2	3	4

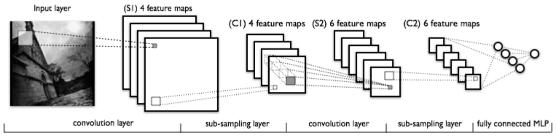
From Stanford's UFLDL Tutorial

Convolutional Layers II

- An $M \times N$ input "image" x is transformed into an $(M - K + 1) \times (N - K + 1)$ output $x' = C(x)$
- This is done over Q input **feature maps** $x_1, \ldots, x_Q$ and creates R output feature maps $x'_1, \ldots, x'_R$
- Then a **pooling** transformation $P(x')$ over $K' \times K'$ patches of each x'_j
  - Possible pooling transforms: averages, max
- We have to learn $Q \times R$ pairs of $K \times K$ filters (w_ℓ, b_ℓ)
  - And decide on K, K' and the pooling transformation
- The forward pass has a cost of $O(Q \times M \times N \times K^2 \times R)$ per pattern, which can be quite costly

Deep Convolutional NNs

- Important goals may be achieved: invariance preservation, structural feature extraction, balancing layer sizes
- Deep Convolutional NNs combine the previous steps
  - An initial number of convolutional layers, followed by
  - A number of fully connected inner product layers and, finally
  - A readout layer that yields the NN's response



A typical architecture for image processing. From Convolutional Neural Networks (LeNet) tutorial

- Possibly with connections and weights in the millions

New Optimization Techniques

- Second order methods across iterations are only possible over small minibatches
- New ideas have been progressively introduced
  - Either refinements of previous approaches more or less sidelined: **Rprop, momentum a la Nesterov**
  - Or often borrowed from other optimization contexts: **Adagrad, Adadelta, Adam**
  - Or simply (overlooked) common sense: minibatch training
- Two main goals:
  - To shorten computation time (obviously)
  - To simplify hyperparameter handling and selection (even more so!)

Avoiding Learning Rates

- First idea: apply CG on the minibatches
  - Done in some packages
- Alternative (from convex optimization) **Adagrad**

$$w_{ij}^{t+1} = w_{ij}^t - \frac{\epsilon}{\sqrt{\sum_{s=1}^t (g_{ij}^s)^2}} g_{ij}^t = w_{ij}^t - \frac{\epsilon}{\sqrt{t}} \frac{g_{ij}^t}{\sqrt{\frac{1}{t}\sum_{s=1}^t (g_{ij}^s)^2}}$$

  - Only requires to store gradient information g_{ij}^s
  - It can be seen as an extension of second order rates

- There is the hand tuned global learning rate ϵ (not too important) but each weight has its own dynamic rate
- But the denominator may grow throughout training making too small the overall learning rate
- And we do not have the same "dimensions" on the left and right equations (they do using second order information if we assume f to be unit–less)

Adadelta

- In Adadelta we get back the right units in w_{ij} by adding $\sqrt{\sum_1^t (\Delta w_{ij}^s)^2}$ into the numerator to have

$$w_{ij}^{t+1} = w_{ij}^t - \epsilon \frac{\sqrt{\frac{1}{t}\sum_1^t (\Delta w_{ij}^s)^2}}{\sqrt{\frac{1}{t}\sum_1^t (g_{ij}^s)^2}} g_{ij}^t = w_{ij}^t - \epsilon \frac{RMS_t(\Delta w_{ij}^s)}{RMS_t(g_{ij}^s)} g_{ij}^t$$

- We avoid storing momentum/gradient info working with exponentially smoothed averages

$$\begin{aligned} RMS_t(g_{ij}^s)^2 &= (1-\rho)\left[RMS_{t-1}(g_{ij}^s)\right]^2 + \rho(g_{ij}^t)^2; \\ RMS_t(\Delta w_{ij}^s)^2 &= (1-\rho)\left[RMS_{t-1}(\Delta w_{ij}^s)\right]^2 + \rho(\Delta w_{ij}^t)^2 \end{aligned}$$

  - ρ is called the **smoothing** factor
  - And $1-\rho$ sometimes the **forgetting** factor

Adam

- 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

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

  - 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 $W_t = W_{t-1} - \alpha \frac{\widehat{m}_t}{\sqrt{\widehat{v}_t}+\epsilon}$
- One can show $E[m_t] \simeq (1-\beta_1^t)E[g_t]$ and $E[v_t] \simeq (1-\beta_2^t)E[g_t^2]$
- Default values $\alpha = 0.001, \beta_1 = 0.9, \beta_2 = 0.999$, and $\epsilon = 10^{-8}$ usually work fine

Takeaways in Deep Networks

1. Right initialization crucial

2. ReLUs as new activation function

3. Dropout for fully connected layer regularization

4. Convolutional layers to be used on structured inputs (but processing much costlier)

5. New optimization ideas (Adagrad, Adadelta, Adam) to simplify handling of learning rates

6. Heavy duty computing environments, particularly for hyperparameterization

7. Need to use tools able to derive symbolic backpropagation but fancy DAG–like networks possible

8. Guidelines in Best Practices for Applying Deep Learning to Novel Applications

9.3 The Golden Era?

Renewed, Huge Interest

- Things go from a mild NN stagnation around 2000 to big explosion in the 2010s
- Relatively large number of contributions and widely attented workshops in mayor conferences (ICML, NIPS)
- Strong groups in leading companies (Google, Baidu, Facebook, Microsoft)
- Great scientific (and mediatic) success: Deep learning. LeCun, Bengio & Hinton (Nature, May 2015)
- New field arising: **Representation Learning**
- New (possible and perhaps more plausible) connections with computational neuroscience (at least for image and audio recognition?)

Great Successes

- DNNs define a rich and suggestive paradigm with impressive results in several fields
  - They vastly improve the previous state of the art (Viterbi models) in speech recognition,
  - They consistently give the best results in the latest Image Net Large Scale Visual Recognition Challenges
- Google is particularly active:
  - Caption generation from images
  - Smart Reply: automatically recommendation of responses to messages in Gmail
  - Learning to play video console games: Nature, February 2015
  - Public release of TensorFlow (plus MOOC in Udacity)
  - Beating humans at go ten years in advance: Nature, January 20156

Right Now

- Great hype and substantial entry barriers

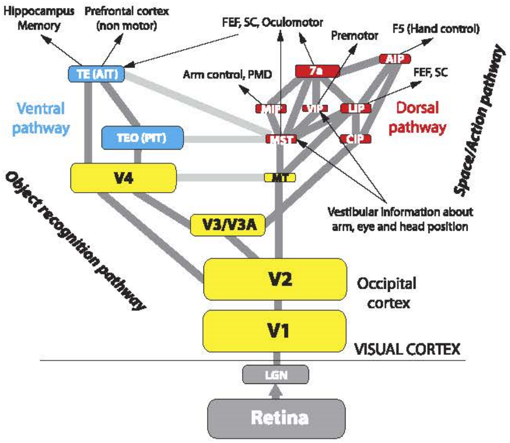
  - Of course knowledge has to be acquired (perhaps not too different from before)
  - But large computational (and technical) resources are indispensable
  - And perhaps their natural habitat are problems with extremely large training databases

- Cutting edge Deep Nets are big, complicated and nervous animals, but also full of promise

- Plausible goal: train (teach?) networks to process information in a hierarchical way

What Are DNNs Aiming At?

- Model: information processing in the visual cortex



From Kruger et al., PAMI 35, 2013

The Ideal Deep Net

- Desired working:
  - The first and intermediate layers extract information substructures
  - The final layers recompose into cognitive content

- Ultimate goal: to replicate the cortex's workings to
  - Decompose a complex tasks in elementary subtasks
  - Solve each one separately and
  - Merge these subsolutions on a complex and rich representation

- That is, to achieve a kind of cognitive "Map Reduce"

- And Deep Learning is clearly behind the **renewed conversation on AI** and its implications

Renewing The AI Conversation

- Decomposing and merging is similar to what it is being done in other AI fields (such as self driving cars)
- But also in the automatization of industrial and (increasingly) service processes
- Very likely with important economic and social disruptions
- Two very recent examples: OpenAI, NIPS 2015 Symposium

What Next?

- OpenAI: ... *to advance digital intelligence ... to benefit humanity ... unconstrained by ... financial return ...*
  - Research Director: I. Sutskever (U. Toronto–Hinton, U. Stanford–Ng, Google)
  - Sponsors: Elon Musk (Tesla), Reid Hoffman (LinkedIn), Peter Thiel (PayPal)
  - Up to 1 billion dollars pledged
- NIPS 2015 symposium Algorithms Among Us: The Societal Impacts of Machine Learning, with among others
  - Nick Bostrom, Future of Humanity Institute–Oxford U.
  - Andrew Ng, Stanford–Coursera–Baidu, The Economic Impact of Machine Learning (podcast)
  - Erik Brynjolfsson, MIT, The Second Machine Age
- We'll see!!!