Summary
CS-233 Machine Learning

Clément Charollais

Last Update : June 26, 2019

Contents

1 Classification

1.1 Bayes

Idea The classifier is expressed as conditional probability. Given a sample $\mathbf{x}$, we can estimate the probability of this sampled to be from each class C_k of the K classes according to the chosen model.

Assumption Features are mutually independent and samples are *iid*.

$$P(C_k|\mathbf{x}) = \frac{P(\mathbf{x}|C_k)\,P(C_k)}{P(\mathbf{x})}$$
$$P(C'_k|\mathbf{x}) = \frac{P(\mathbf{x}|C'_k)\,P(C'_k)}{P(\mathbf{x})}$$

$P(\mathbf{x})$ is irrelevant, thus

$$P(C_k|\mathbf{x}) \propto P(\mathbf{x}|C_k)\,P(C_k)$$
$$P(C'_k|\mathbf{x}) \propto P(\mathbf{x}|C'_k)\,P(C'_k)$$

We can then classify the sample with the classifier

$$\begin{aligned}\hat{y} &= \underset{k\in\{1,\dots,K\}}{\operatorname{argmax}} P(\mathbf{x}|C_k)\,P(C_k) \\ &= \underset{k\in\{1,\dots,K\}}{\operatorname{argmax}} P(C_k)\prod_{i=1}^{d} P(x_i|C_k)\end{aligned}$$

Here the probabilities are empirical and depend on the sample set.

Remarks

- A very strong assumption is made : features are mutually independent. This means that if that if it is not the case the classifier can potentially be very bad. In practice, the classifier can be good even if the features are not independent.
- Prior probabilities are important (if sample set is highly unbalanced, there can be problems)

1.2 *k* Nearest Neighbours

Idea Given a new sample x to classify, we assign it to the class of the majority of its k nearest points. We use a validation method to determine the value k for which the classifier performs the best.

Assumption The training set and the testing set arise from the same underlying distribution, i.e. the training set must be representative of real life.

Remarks

- The parameter k governs the degree of smoothness of the boundary between the classes.
- This classifier is very easily generalized to multi-class classification.
- The distance metric must be chosen carefully because of problems arising from noisy data or high dimensionality
- It is necessary to load the entire dataset for each classification : that's slow. Some methods exist to reduce the dataset size though : Condensed Nearest Neighbours (Fast Condensed Nearest Neighbours also exist)
- Normalization of the features is necessary in order for the features to be all of equal importance $\mathbf{x}_{\text{norm}} = \frac{\mathbf{x}-\mu}{\sigma}$. (The mean and standard deviation are made here featurewise)
- kNN performs poorly with highly unbalanced datasets.
- It is impossible to distribute the computation, very slow for large datasets

1.3 Perceptron

Idea Find a hyperplane that segregates the dataset into its two different classes.

Assumption The N samples $\mathbf{x}_n$ are linearly separable. Classifier for linearly separable dataset. Convergence ensured only when labels t_n are linearly separable, i.e. separable by a hyperplane.

Model

$$\hat{y}(\mathbf{x}) = \begin{cases} +1 & \mathbf{w}^{*T}\mathbf{x} \geq 0 \\ -1 & \text{otw.} \end{cases}$$

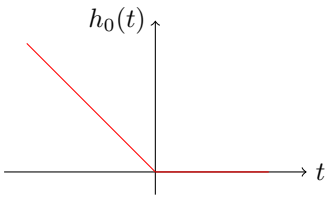
$$\mathbf{w}^* = \underset{\mathbf{w}}{\operatorname{argmin}}\, E(\mathbf{w})$$

$$E(\mathbf{w}) = -\sum_{n \in M} t_n(\mathbf{w}^T X_n) = -\sum_{n=0}^{N} h_0(t_n \mathbf{w}^T X_n)$$

$$\frac{\partial E}{\partial w_i} = -\sum_{n \in M} t_n X_{n,i}$$

$$\nabla E(\mathbf{w}) = t_n X_n$$

Where h_0 is the hinge loss function at 0, i.e. $t < 0 \Rightarrow h_0(t) = -t$, $t \geq 0 \Rightarrow h_0(t) = 0$



Training Gradient descent

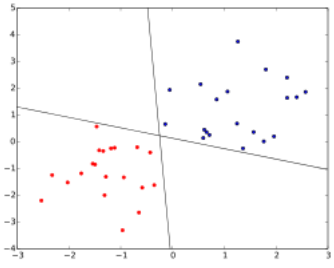
$$\mathbf{w}^{\tau+1} = \mathbf{w}^{\tau} - \eta \nabla E(\mathbf{w}^{\tau}) = \mathbf{w}^{\tau} - \eta t_n X_n$$

Algorithm 1 Perceptron

function PERCEPTRON(Training set $\chi = \{\mathbf{x}_n, t_n\}_{n=1}^N$ with $t_i \in \{-1, 1\}$)
  Center the data to do away with bias term
  Initialise weights $\mathbf{w}^{(1)} = 0$ $\forall n$
  For some random n in N
    if $y(\mathbf{x}_n) \neq t_n$ **then**
      $\mathbf{w}^{(t+1)} = \mathbf{w}^{(t)} + t_n \mathbf{x}_n$
  When no more changes occur, the hyperplane has converged and $y(\mathbf{x}) = 2\mathbb{1}(\mathbf{w}^T\mathbf{x} > 0) - 1$

Remarks

- The perceptron does not give any guarantee to the quality of the discriminator. It is equally likely to end up in each of those positions. There are infinitely many correct discriminating hyperplanes.



- The samples must be centred or a bias term must be added for the perceptron to be able to classify the samples properly.

1.4 Logistic Regression

Idea Instead of giving absolute results like the perceptron, the classifier given by the logistic regression outputs the probability that a certain sample belongs to a given class.

Model

$$P(C_1|\,\mathbf{x}) = y(\mathbf{x}) = \sigma(\mathbf{w}^T\mathbf{x} + w_0)$$
$$P(C_2|\,\mathbf{x}) = 1 - P(C_1|\,\mathbf{x}) = 1 - \sigma(\mathbf{w}^T\mathbf{x} + w_0)$$

Maximum likelihood is then used to determine the parameters of the model

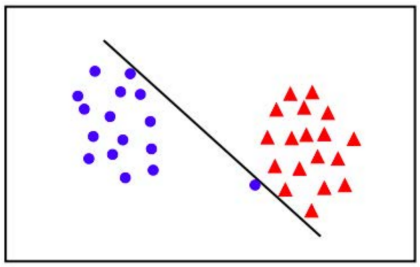
$$P(\mathbf{t}|\,\mathbf{w}) = \prod_{n=1}^{N} y_n^{t_n}(1-y_n)^{1-t_n}$$
$$E(\mathbf{w}) = -\ln P(\mathbf{t}|\,\mathbf{w}) = -\sum_{n=1}^{N}\left[t_n \ln y_n + (1-t_n)\ln(1-y_n)\right]$$
$$\nabla E(\mathbf{w}) = \sum_{n=1}^{N}(y_n - t_n)\mathbf{x}_n = X^T(\mathbf{y}-\mathbf{t})$$

Training Gradient descent

$$\mathbf{w}^{\tau+1} = \mathbf{w}^{\tau} - \eta\nabla E(\mathbf{w}^{\tau}) = \mathbf{w}^{\tau} - \eta t_n X_n$$

Remarks

- The naive Bayes hypothesis is also used here implicitly.
- The model has $D+1$ parameters for D features, which is cool
- Severe overfitting can occur for linearly separable datasets (the probabilities all go to 1). The sigmoid function becomes increasingly steep and becomes the step function. But it is possible to counteract that by adding a regularization term to the error function.



Multi-class Logistic Regression The idea is essentially the same but we compute a binary classifier for each class against all others. For k classes we thus have k classifiers. Softmax is used instead of sigmoid and the maximum likelihood becomes :

$$P(C_k|\,\mathbf{x_n}; W) = y_k(\mathbf{x}_n) = \frac{\exp(\mathbf{w}_k^T\mathbf{x}_n)}{\sum_{j=1}^{N}\exp(\mathbf{w}_j^T\mathbf{x}_n)}$$
$$P(\mathbf{t}|\,W) = \prod_{k=1}^{K}\prod_{n=1}^{N} P(C_k|\,\mathbf{x}_n)^{t_{nk}}$$
$$E(W) = -\ln P(\mathbf{t}|\,W) = -\sum_{k=1}^{K}\sum_{n=1}^{N} t_{nk}\ln(y_k(\mathbf{x}_n))$$
$$\nabla_{\mathbf{w}_k} E(W) = \sum_{n=1}^{N}(y_k(\mathbf{x}_n) - t_{nk})\mathbf{x}_n$$

1.5 Support Vector Machines

It is often the case that data is not directly linearly separable, but mapping the data to a higher dimensional feature space can yield linearly separable classes. Directly taking this approach can however have a high computation cost, hence it is preferable to use support vector machines with a kernel trick.

Idea Instead of simply trying to segregate the data int two categories, the goal of a support vector machine is to maximize the margin between the discriminator hyperplane and the closest data-points. Whereas neither a perceptron and a logistic regression classifier guarantees a margin at all, support vector machines guarantee the margin will be the largest possible.

The distance to maximize is the distance of data points to the margin which is

$$y(\mathbf{x}) = \mathbf{w}^T \phi(\mathbf{x}) + b \tag{1}$$

The size of the margin is constrained on the each of the closest point each class to the margin. It is always possible to rescale the data such that both those points are at distance 1 from the margin.

$$t_n \left(\mathbf{w}^T \phi(\mathbf{x}_n) + b\right) \geq 1 \quad \forall n \in \{0, 1, \ldots, N\}$$

With this rescaling, the problem of maximizing the margin reduces to

$$\underset{\mathbf{w},b}{\operatorname{argmax}} \left\{ \frac{1}{\| \mathbf{w} \|} \min_n \left[t_n \left(\mathbf{w}^T \phi(\mathbf{x}_n) + b\right)\right] \right\} = \underset{\mathbf{w}}{\operatorname{argmax}} \frac{1}{\| \mathbf{w} \|} = \underset{\mathbf{w}}{\operatorname{argmin}} \frac{1}{2} \| \mathbf{w} \|^2$$

This yields a quadratic programming problem with the constraints imposed by (1). which gives the following Lagrangian and setting its derivatives w.r.t. $\mathbf{w}$ and b to zero yields.

$$\begin{aligned} L(\mathbf{w}, b, \mathbf{a}) &= \frac{1}{2} \| \mathbf{w} \|^2 - \sum_{n=1}^{N} a_n \left(t_n \left(\mathbf{w}^T \phi(\mathbf{x}_n) + b\right) - 1\right) \\ \mathbf{w} &= \sum_{n=1}^{N} a_n t_n \phi(\mathbf{x}) \\ 0 &= \sum_{n=1}^{N} a_n t_n \end{aligned}$$

Hence

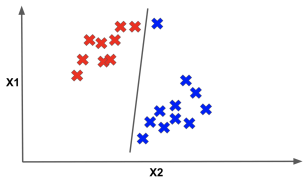
$$\begin{aligned} y(x) &= \sum_{n=1}^{N} a_n t_n \phi^T(\mathbf{x}) \phi(\mathbf{x}_n) + b \\ &= \sum_{n=1}^{N} a_n t_n k(\mathbf{x}, \mathbf{x}_n) + b \end{aligned}$$

Remarks

- $O(N^3)$ so not so efficient
- Using a kernel $k(\mathbf{x}, \mathbf{x}')$ instead of the mapping function ϕ reduces dractically the computation. The can be thought as a similarity measure.
- $a_n \neq 0$ only for a subset of the vectors and such vectors are vectors s.t. $t_n \left(\mathbf{w}^T \phi(\mathbf{x}_n) + b\right) = 1$. These are called the *support vectors* and they lie on the maximum margin hyperplanes. Once the classifier is trained, the model complexity can be drastically reduced by discarding all non support vectors.

1.5.1 Slack variables

Idea Severe overfitting can occur if outliers are present in the training dataset. To make it less sensitive we can relax the margin condition to allow some points to be misclassified at training to maximize the margin further.



We then minimize

$$C\sum_{n=1}^{N}\xi_n + \frac{1}{2}\parallel \mathbf{w} \parallel^2$$

Where C is the tradeoff parameter.

Remarks

- A small C will allow for more vectors to be misclassified.
- the tradeoff between the number of mistakes on the training data and the margin size is controlled by C.
- $a_n \neq 0$ only for a subset. For those vectors $a_n < C$ implies that the point is on the margin. $a_n = C$ implies that the point is inside the margin and is misclassified if $\xi_n > 1$ and correctly classified otherwise.

1.5.2 Kernels

Gaussian Kernel This kernel represents a gaussian distribution around the center vector. Its corresponding feature space is of infinite dimension.

$$k(\mathbf{x}, \mathbf{x}') = e^{-\frac{\parallel \mathbf{x}-\mathbf{x}' \parallel^2}{\sigma^2}}$$

σ represents the variance and $\mathbf{x}$ the mean. A large σ will hence induce a smoother boundary

Polynomial kernels

$$k(\mathbf{x}, \mathbf{x}') = 1 + (\mathbf{x}^T\mathbf{x}')^d$$

Remarks

- Validation is essential to determine the best parameters.

1.6 Boosting

The principle is to build a classifier composed of multiple weak classifiers (perceptrons or other simple classifiers). The weak classifiers are trained iteratively and a weighted sum is taken to product the final classifier.

$$\underbrace{y(\mathbf{x})}_{\text{strong classifier}} = \underbrace{\alpha_1}_{\text{weight}} y_1(\mathbf{x}) + \alpha_2 \underbrace{y_2(\mathbf{x})}_{\text{weak classifier}} + \alpha_3 y_3(\mathbf{x}) + \dots$$

The first weak classifier is trained with weighted data whose weights are all equal, then, the subsequent weak classifiers will be trained on data whose weight is greater if it was badly classified before and smaller is it was classified properly.

The weights given to the weak classifiers depend on their accuracy.

Algorithm 2 AdaBoost

function AdaBoost(Training set $\chi = \{\mathbf{x}_n, t_n\}_{n=1}^N$ with $t_i \in \{-1, 1\}$, Number of weak classifiers K)
  Initialise weights $w_n^{(1)} = \frac{1}{N}\ \forall n$
  for $k = 1$ **to** K
    Find classifier $y_k : \chi \to \{-1, 1\}$ that minimises $\sum_{t_n \neq y_k(\mathbf{x}_n)} w_n^{(k)}$
    $\epsilon_k := \frac{\sum_{t_n \neq y_k(\mathbf{x}_n)} w_n^{(k)}}{\sum_{n=1}^N w_n^{(k)}}$ ▷ $\epsilon_k < 0.5$ is y_k is better than chance
    $\alpha_k := \log\left(\frac{1-\epsilon_k}{\epsilon_k}\right)$ ▷ $\alpha_k > 0$ is y_k is better than chance
    $w_n^{(k+1)} = w_n^{(k)} \exp\left(\alpha_k \mathbb{1}(t_n \neq y_k(\mathbf{x}_n))\right)$
  The final classifier is $y(\mathbf{x}) = \text{sign}\left(\sum_{k=1}^K a_k y_k(\mathbf{x})\right)$

Remarks

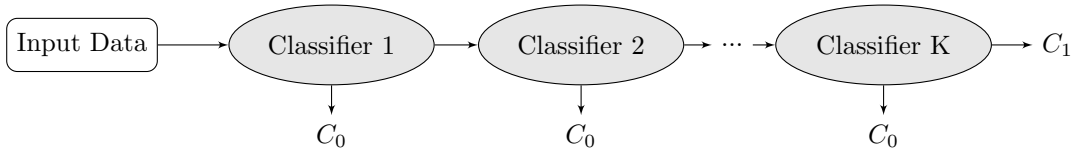
- AdaBoost performs a gradient descent on the Exponential loss
  At iteration k, given $\{y_i\}_{i=1}^{k-1}$ and $\{\alpha_i\}_{i=1}^{k-1}$, AdaBoost minimises

$$\begin{aligned} E_k &= \sum_{n=1}^N \exp\left(\frac{1}{2}\alpha_k y_k(\mathbf{x}_n) - t_n \frac{1}{2}\sum_{l=1}^{k-1}\alpha_s y_s(\mathbf{x}_n)\right) \\ &= \sum_{n=1}^N w_n^{(k)} \exp\left(-\frac{1}{2} t_n \alpha_k y_k(\mathbf{x}_n)\right) \end{aligned}$$

  Minimizing $\sum_{n=1}^N w_n^{(k)} \exp\left(-\frac{1}{2} t_n \alpha_k y_k(\mathbf{x}_n)\right)$ with respect to y_k and a_k yields that y_k must minimise $\sum_{t_n \neq y_k(\mathbf{x}_n)} w_n(k)$ with

$$\begin{aligned} \epsilon_k &= \frac{\sum_{t_n \neq y_k(\mathbf{x}_n)} w_n^{(k)}}{\sum_{n=1}^N w_n^{(k)}} \\ \alpha_k &= \log\left(\frac{1-\epsilon_k}{\epsilon_k}\right) \\ w_n^{(k+1)} &= w_n^{(k)} \exp\left(\alpha_k \mathbb{1}(t_n \neq y_k(\mathbf{x}_n))\right) \end{aligned}$$

- The training error decays exponentially fast if the weak classifiers always perform better than chance
- Overfitting is possible with too many weak classifiers. Thus validation is necessary.
- Different types of classifiers can be used.
- Cascades architectures can be used to improve speed at run-time



1.7 Trees

Trees separate the input space into multiple regions. The tree structure corresponds to the splitting of the input space. Each node represents a decision (i.e. a weak classifier splits the input space into two separate regions). At each leaf there is a separate classifier to predict the target variable (in classification it is possible just to assign a class to each leaf).

Separation Different heuristics can be used as weak learners. With input $\mathbf{x}$, the weak learner $h(\mathbf{x};\boldsymbol{\theta})$ yields only two answers

- Hyperplane $h(\mathbf{x};\boldsymbol{\theta}) = [\tau_1 < \phi(\mathbf{v})\cdot\psi < \tau_2]$, where τ_1 and τ_2 are some parameters, and ψ is some hyperplane
- Sheric selection $h(\mathbf{x};\boldsymbol{\theta}) = [\tau_1 < \psi^T\phi(\mathbf{v})\psi < \tau_2]$, where τ_1 and τ_2 are some parameters, and ψ is some matrix representing a conic.

  In practice, ϕ can map the input space to only a subspace of the feature space.

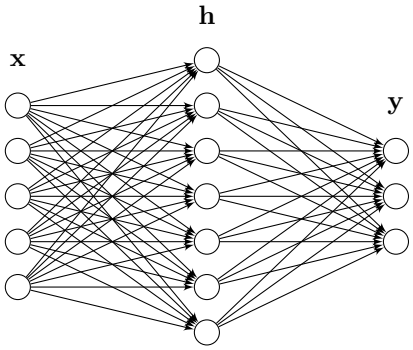
The idea that allows the weak learners to be trained is information maximisation. i.e. we find the $\boldsymbol{\theta}$ such that $I(\mathcal{S},\boldsymbol{\theta})$ when $\mathcal{S}$ is split into $\mathcal{S}^L$ and $\mathcal{S}^R$

$$I(\mathcal{S},\boldsymbol{\theta}) = H(\mathcal{S}) - \frac{|\mathcal{S}^R|}{|\mathcal{S}|}H(\mathcal{S}^R) - \frac{|\mathcal{S}^L|}{|\mathcal{S}|}H(\mathcal{S}^L)$$
$$H(\mathcal{S}) = -\sum_{c\in C} P(c)\log(P(c))$$

1.8 Multi-Layer Perceptron

1.8.1 Single hidden layer

multiple copies of a logistic regression classifier are used each with their weight and bias are used



$$\mathbf{h} = \sigma_1(W_1\mathbf{x} + b_1)$$
$$\mathbf{y} = \sigma_2(W_2\mathbf{h} + b_2)$$

Where W_k is the weights matrix of the k-th hidden layer, $\mathbf{b}_k$ its bias vector, and σ_k its activation function. And the output is a differentiable function of the weights and the biases.

Binary case σ_2 restricts the output to [0,1] and is interpreteed as the probability os the input to belong to a class. The minimisation problem is to minimise $E(\mathbf{W},\mathbf{b})$ given a training set $\{\mathbf{x}_n, t_n\}_{n=1}^N$.

$$\begin{aligned} E(\mathbf{W},\mathbf{b}) &= \sum_{n=1}^N E_n(\mathbf{x}_n;\mathbf{W},\mathbf{b}) \\ &= \sum_{n=1}^N t_n\log(y(\mathbf{x}_n)) + (1-t_n)\log(1-y(\mathbf{x}_n)) \\ y(\mathbf{x}_n) &= \sigma_2(W_2(\sigma_1(W_1\mathbf{x}_n + \mathbf{b}_1)) + \mathbf{b}_2) \end{aligned}$$

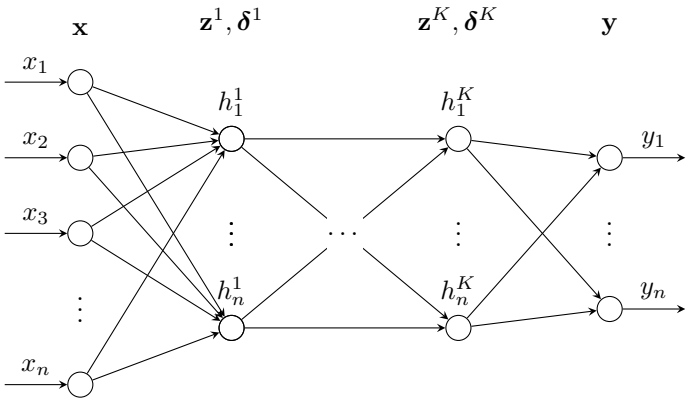
Multiclass case In the multiclass case the $P(\mathbf{x}\in c;\mathbf{W},\mathbf{b}) = \frac{y_c(\mathbf{x};\mathbf{W},\mathbf{b})}{\sum_k y_k(\mathbf{x};\mathbf{W},\mathbf{b})}$. Given $\{\mathbf{x}_n, t_{cn},\dots,t_{Cn}\}_{n=1}^N$, the minimisation problem is

$$\begin{aligned} E(\mathbf{W},\mathbf{b}) &= \sum_{n=1}^N E_n(\mathbf{x}_n;\mathbf{W},\mathbf{b}) \\ &= \sum_{n=1}^N\sum_c t_{cn}\log(P(\mathbf{x}_n\in c;\mathbf{W},\mathbf{b})) \end{aligned}$$

Interpretation Each node defines an hyperplane and the final output is the sum of the distances to all hyperplanes. This is quite similar to AdaBoost, but here all the weak classifiers are trained at the same time, yielding better results

1.8.2 Multiple hidden Layers

The descritptive power of a single layer MLP is linear with the number of nodes in the hidden layer. If more layers are stacked, the descriptive power of the classifier is linear with their product.



1.8.3 Forward pass

$$\mathbf{z}^1 = \sigma_1(W^1\mathbf{x} + \mathbf{b}^1) = \sigma_1(\mathbf{a^1})$$
$$\mathbf{z}^2 = \sigma_2(W^2\mathbf{z}^1 + \mathbf{b}^2) = \sigma_1(\mathbf{a^2})$$
$$\dots$$
$$\mathbf{y} = \sigma_1(W^{K+1}\mathbf{z}^K + \mathbf{b}^{K+1} = \sigma_1(\mathbf{a}^{K+1}))$$

The error to minimise is the same as for the single layer perceptron, but this time ,the weight vector and the biases vector are made of more elements.

1.8.4 Back-propagation

For a deepnet with input size D_0, output side D_{K+1} with K hidden layers of size D_k, each with weights matrices W^k, where biases are included, and activation functions σ_k.

$$a_i^1 = \sum_{\alpha=1}^{D_0} x_\alpha w_{\alpha i}^1 + b_i^1 \qquad z_i^1 = \sigma_1(a_i^1)$$
$$a_i^k = \sum_{\alpha=1}^{D_{k-1}} z_\alpha^{k-1} w_{\alpha i}^k + b_i^k \qquad z_i^k = \sigma_k(a_i^k)$$
$$a_i^{K+1} = \sum_{\alpha=1}^{D_K} z_\alpha^K w_{\alpha i}^{K+1} + b_i^{K+1} \qquad y_i = a_i^{K+1}$$

$$E(\mathbf{W}, \mathbf{b}) = \sum_{n=1}^{N} E_n(\mathbf{x}_n; \mathbf{W}, \mathbf{b})$$

$$\mathbf{w}^{(\tau+1)} = \mathbf{w}^{(\tau)} - \eta \sum_{n \in B_\tau}^{N} \nabla E_n(\mathbf{w}^{(\tau)})$$

at output layer:

$$\begin{aligned} \frac{\partial E_n}{\partial w_{i,j}^{K+1}} &= \frac{\partial E_n}{\partial a_j^{K+1}} \frac{\partial a_j^{K+1}}{\partial w_{i,j}^{K+1}} \\ &= \delta_j^{K+1} z_j^K \\ \delta^{K+1} &= \frac{\partial E_n}{\partial a_j^{K+1}} \end{aligned}$$

at other layers:

$$\begin{aligned} \frac{\partial E_n}{\partial w_{i,j}^{k}} &= \frac{\partial E_n}{\partial z_j^{k}} \frac{\partial z_j^k}{\partial a_j^k} \frac{\partial a_j^k}{\partial w_{i,j}^k} \\ &= \sigma'(a_i^k) \delta_i^k z_j^{k-1} \\ \delta_j^k &= \sigma'(a_j^k) \sum_{i=1}^{D_{k-1}} w_{i,j}^k \delta_i^{k+1} \end{aligned}$$

Remarks

- The output y is non concave, thus there are a lot of local minima, the result hence depends on the initialisation of the weights. Some local-minima escaping techniques are necessary (AdaGrad for instance)
- The error function is either derived from the maximum likelihood of the probabilistic interpretation of the output of the model, which yields the mean square error or is the logistic error.
- Some bypasses can be made to improve convergence in certain cases.
- In very deep networks if $\sigma(a_j)$ approach 0, so does δ_i thus the gradient becomes 0 and no further improvement can be made (Vanishing Gradient). If on the other hand δ_i becomes huge and no improvement can be made either.
- The choice of the different activation functions have different tradeoffs and some might work better for some problems (Validation)

Sigmoid	Vanishing gradient Not zero-centred (gradient updates go too far in each direction → harder optimisation) Saturates and kills gradients Slow convergence
Tanh	Vanishing gradient Zero-centred
ReLu	No vanishing gradient Converges 6 times faster that tanh **Can only be used in hidden layers**, the output should have a softmax of some sort Dying gradients (Leaky ReLu solves that)

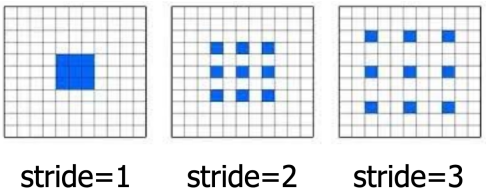
- As MLPs have great descriptive power, they are prone to overfitting. To solve that it is possible to use minibatches (SGD) or to add regularisation to the weights.
- Perceptrons do **not** extrapolate well even if they interpolate well.

1.9 Convolutional Networks

Idea In regular deepnets, the input is squashed into a $1 \times D$ vector, but doing that can remove some useful locality information when treating data whose nature is not intrinsically one dimensional (e.g. images, videos, etc)

Pooling layers Reduce the size of their input (e.g. maximizing, averaging or whatever)

Stride Sparsity of the convolution



ResNet Residual NN can involve Convolutional layers

2 Regression

2.1 Linear Regression

A linear regression solution to a problem which is of the form

$$y(\mathbf{x}; \mathbf{w}) = \mathbf{w}^T \boldsymbol{\phi}(\mathbf{x})$$

Where $\boldsymbol{\phi}(\mathbf{x}) = (1, \phi_1(\mathbf{x}), \dots, \phi_{m-1}(\mathbf{x}))$ is the*feature vector* it is a vector of functions of $\mathbf{x}$ of size m, and w is the parameters vectors to be found. $\boldsymbol{\phi}$'s non-linearities allows the model to fit nonlinear data.

- $\boldsymbol{\phi}(\mathbf{x}) = (1, x_1, x_2)$ fits lines.
- $\boldsymbol{\phi}(\mathbf{x}) = (1, x_1^2, x_1x_2, x_2^2, x_1, x_2)$ fits ellipses.

2.1.1 Maximum likelihood solution

Assuming the data is given by a deterministic function with additive Gaussian noise of mean 0 and variance $\frac{1}{\beta}$

$$\mathbf{t} = \tilde{y}(\mathbf{x}; \mathbf{w}) + \epsilon = W^T \boldsymbol{\phi}(\mathbf{x};) + \epsilon$$

Assuming the data $X, T = \{\mathbf{x}_n, \mathbf{t}_n\}_{n=1}^N$ is iid, maximum likelihood yields the following, which justifies the least squares optimisation method. (NB, the maximisation is achieved by the derivation wrt. $\mathbf{w}$)

$$\begin{aligned}
P\left(T \,\middle|\, X; \mathbf{w}, \beta\right) &= \prod_{n=1}^{N} P\left(\mathbf{t}_n \,\middle|\, \mathbf{x}_n; W, \beta\right) \\
&= \prod_{n=1}^{N} P\left(\mathbf{t}_n \,\middle|\, W^T \boldsymbol{\Phi}(\mathbf{x}_n); W, \beta\right) \\
&= \prod_{n=1}^{N} \mathcal{N}(\mathbf{t}_n \,|\, W^T \boldsymbol{\Phi}(\mathbf{x}_n); W, \beta) \\
\log P\left(T \,\middle|\, W, \beta\right) &= \frac{N}{2} \log \beta - \frac{N}{2} \log 2\pi + \beta E_D(W) \\
E_D(W) &= \frac{1}{2} \sum_{n=1}^{N} \left(\mathbf{t}_n - W^T \boldsymbol{\Phi}(\mathbf{x}_n)\right)^2 \\
\Rightarrow \quad \nabla E_D(W) = \nabla \log P\left(T \,\middle|\, W, \beta\right) &= \beta \sum_{n=1}^{N} \left(\mathbf{t}_n - W^T \boldsymbol{\Phi}(\mathbf{x}_n)\right) \boldsymbol{\Phi}^T(\mathbf{x}_n)
\end{aligned}$$

Setting the gradient to 0 and solving for W yields

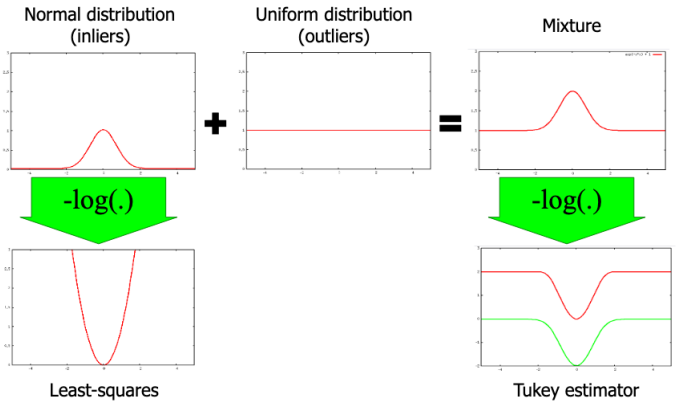
$$\begin{aligned}
0 &= \sum_{n=1}^{N} \mathbf{t}_n \boldsymbol{\Phi}(\mathbf{x}_n)^T - W^T \left(\sum_{n=1}^{N} \boldsymbol{\Phi}(\mathbf{x}_n) \boldsymbol{\Phi}^T(\mathbf{x}_n) \right)^T \\
W_{ML} &= \left(\boldsymbol{\Phi}^T \boldsymbol{\Phi}\right)^{-1} \boldsymbol{\Phi}^T
\end{aligned}$$

Which is the normal equations for the least squares problem. The quantity $\left(\mathbf{\Phi}^T\mathbf{\Phi}\right)^{-1}\mathbf{\Phi}^T$ can be viewed as the generalisation of the inverse for non-square matrices. $\mathbf{\Phi}$ is defined as

$$\mathbf{\Phi} = \begin{pmatrix} \phi_0(\mathbf{x}_1) & \phi_1(\mathbf{x}_1) & \dots & \phi_{M-1}(\mathbf{x}_1) \\ \phi_0(\mathbf{x}_2) & \phi_2(\mathbf{x}_1) & \dots & \phi_{M-1}(\mathbf{x}_2) \\ \vdots & \vdots & \ddots & \vdots \\ \phi_0(\mathbf{x}_N) & \phi_1(\mathbf{x}_N) & \dots & \phi_{M-1}(\mathbf{x}_N) \end{pmatrix}$$

Remarks

- This method is only well adapted to gaussian noise
- This method involved inverting a $M \times M$ matrix which can be really innefficient
- This method is really sensitive to outliers because the metric is the the euclidian distance. There are some solutions to this problem though
  - **Robust estimators** Use another model for the noise that yields a different metric (e.g. tukey which is a mixture between a uniform distribution and a normal)



  - **Reweighted Least Squares**
  - **RanSac** Take a random subset of the dataset, and fit a line to it. If the line is a reasonably good fit, fit the line to the points which are close enough fitted to the line. Otherwise, try again with another subset, a subset which yields a good line will eventually be found. Its parameters are the size of the random subset, the distance threshold for a good fit, and the maximum number of attempts to make.
  - **ProSac** If there exist a confidence level for the measurements, do the same as RanSac but this time the subsets are the subsets of the most reliable measurements in increasing size.
- It is possible to add constraints to the optimisation problem to increase accuracy. The problem can now be solved by Constrained LVM.

2.2 Fitting multiple lines

Method make a parameter space with a dimension for each parameter. Transform this space into an accumulator (i.e. each point i the parameter space sees its value set to the goodness of the fit produced by the corresponding parameters). Then find the local maxima of the accumulators.

NB : the parameters do not need to represent anything linear, thus this method allows to fit ellipses or other shapes to data

2.3 Polynomial Regression

Here we use the assumption is that the underlying function is a polynomial with Gaussian noise

$$t_n = \sum_{i=0}^{M} w_i x_i + \epsilon_n$$

Thus, the optimal weights are

$$\begin{aligned}\mathbf{w}^* &= \underset{\mathbf{w}}{\operatorname{argmin}} \sum_{n=0}^{N} \left\| \sum_{m=0}^{M} w_m x^m - t_n \right\|^2 + \frac{\lambda}{2} \| \mathbf{w} \|^2 \\ &= \underset{\mathbf{w}}{\operatorname{argmin}} \| \Phi\mathbf{w} - \mathbf{t} \|^2 + \frac{\lambda}{2} \| \mathbf{w} \|^2\end{aligned}$$

With

$$\Phi = \begin{pmatrix} 1 & x_1 & x_1^2 & \dots & x_1^M \\ 1 & x_2 & x_2^2 & \dots & x_2^M \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 1 & x_N & x_N^2 & \dots & x_N^M \end{pmatrix} \qquad \mathbf{w} = \begin{pmatrix} w_0 \\ w_1 \\ \vdots \\ w_M \end{pmatrix} \qquad \mathbf{t} = \begin{pmatrix} t_0 \\ t_1 \\ \vdots \\ t_N \end{pmatrix}$$

The solution is given by

$$\mathbf{w} = \left(\Phi^T\Phi + \lambda I\right)^{-1} \Phi^T \mathbf{t}$$

Remarks

- This solution involves weight decay because without it, the model has a high tendency to overfit the noise in the data (if the data is noise-free, this solution performs very well though)
- λ is chosen through validation.
- When there are a lot of data this method is highly impractical.

2.4 Gaussian Process

Like with SVMs a non-linear function of the input is used to introduce nonlinearities.

$$\mathbf{y}(\mathbf{w}^T\phi(\mathbf{x}))$$

The cost (with weight decay) to minimise is

$$E(\mathbf{w}) = \frac{1}{2} \sum_{n=0}^{N} \left\| \mathbf{w}^T\phi(\mathbf{x} - \mathbf{t}_n) \right\|^2 + \frac{\lambda}{2} \| \mathbf{w} \|^2$$

As for SVMs the kernel trick can be performed and

$$\mathbf{y}(\mathbf{x}) = \mathbf{k}(\mathbf{x}) \left(K - \lambda I\right)^{-1} T$$

Where

$$\mathbf{k}(\mathbf{x}) = \begin{pmatrix} k(\mathbf{x}, \mathbf{x}_1) \\ k(\mathbf{x}, \mathbf{x}_2) \\ \vdots \\ k(\mathbf{x}, \mathbf{x}_N) \end{pmatrix} \qquad K = \begin{pmatrix} k(\mathbf{x}_1, \mathbf{x}_1) & k(\mathbf{x}_1, \mathbf{x}_2) & \dots & k(\mathbf{x}_1, \mathbf{x}_N) \\ k(\mathbf{x}_2, \mathbf{x}_1) & k(\mathbf{x}_2, \mathbf{x}_2) & \dots & k(\mathbf{x}_2, \mathbf{x}_N) \\ \vdots & \vdots & \ddots & \vdots \\ k(\mathbf{x}_N, \mathbf{x}_1) & k(\mathbf{x}_N, \mathbf{x}_2) & \dots & k(\mathbf{x}_N, \mathbf{x}_N) \end{pmatrix} \qquad T = \begin{pmatrix} \mathbf{t}_1^T \\ \mathbf{t}_2^T \\ \vdots \\ \mathbf{t}_N^T \end{pmatrix}$$

If we interpret the result as a probability distribution with variance $\frac{1}{\beta}$, $\mathbf{y}$ is the mean of the distribution and its variance can be computed as well :

$$\sigma^2(\mathbf{x}) = k(\mathbf{x}, \mathbf{x}) + \frac{1}{\beta} - \mathbf{k}(\mathbf{x}) \left(K - \lambda I\right)^{-1} \mathbf{k}(\mathbf{x})$$

Remarks

- This method is affected by the curse of dimentionality as the eudclidian distance is involved
- Information about the gradient at some places can be used to improve the accuracy of the predictions
- $O(N^2)$, which is bad

2.5 BoostedTrees

The feature space is subdivided into regions each with its own weak approximation. A collection of K of those trees are trained in order to improve performance (as with AdaBoost)

$$y(\mathbf{x}) = \sum_{k=0}^{K} \alpha_k y_k(\mathbf{x}) + y_0$$

The following approach is used to find those trees

Algorithm 3 BoostedTrees

1: **function** BOOSTEDTREES(Number of trees classifiers K)
2: Find the constant $y_0 = \underset{y}{\text{argmin}} \sum_{n=0}^{N} E(y, t_n)$
3: **for** k = 0 **to** K
4: Train a Tree y s.t. $\forall n\ y(\mathbf{x}_n) \approx \frac{\partial E(y_{k-1}(\mathbf{x}_n), t_n)}{\partial y_{k-1}(\mathbf{x}_n)}$
5: $\alpha_k = \underset{k}{\text{argmin}} \sum_{n=0}^{N} E(y_{k-1}(\mathbf{x}_n) + \alpha y(\mathbf{x}_n), t_n)$
6: $y_k(\mathbf{x}) = y_{k-1}(\mathbf{x}) + \alpha_k y(\mathbf{x})$

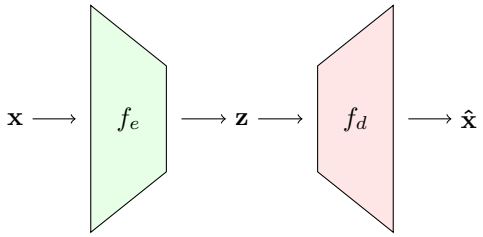
Remarks

- This technique works because the step $y_k(\mathbf{x}) = y_{k-1}(\mathbf{x}) + \alpha_k y(\mathbf{x}) \approx y_{k-1}(\mathbf{x}) - \alpha_k \frac{\partial E(y_{k-1}(\mathbf{x}_n), t_n)}{\partial y_{k-1}(\mathbf{x}_n)}$ which is essentially a gradient descent step.
- This technique can be used for multiclass classification if the labels are os the form $\mathbf{t}$ where $t_k \neq 0 \Rightarrow$ the associated input is of class C_k

2.6 Autoencoders

Before trying to do regression it can be useful to try to remove unnecessary degrees of freedom. i.e. map the inputs to a lower dimension space without losing information. This is achieved by training two functions together $f_e : \mathbb{R}^D \to \mathbb{R}^d$ and $f_d : \mathbb{R}^d \to \mathbb{R}^D$ where $d < D$ such that $f_d \circ f_e$ is nearly the identity. $\mathbb{R}^d$ is called the latent space and $f_e(\mathbf{x})$ the latent vector of $\mathbf{x}$.

2.6.1 Reduction



$$\mathbf{z} = \sigma_e(W_e\mathbf{x} + \mathbf{b}_e)$$
$$\hat{\mathbf{x}} = \sigma_d(W_d\mathbf{z} + \mathbf{b}_d)$$

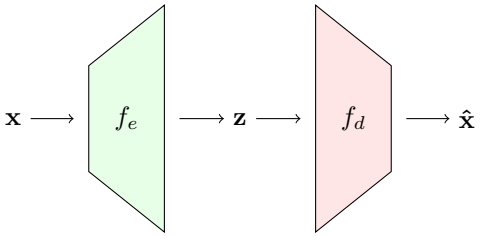
And the goal is to minimise $\sum_{n=1}^{N} \| \hat{\mathbf{x}}_n - \mathbf{x}_n \|^2$

Remarks

- A convolution with stride is necessary to reduce the dimensionality.
- Autoencoders can be used to achieve compression, reducing the noise, make correlations more apparent, remove degrees of freedom, and interpolation.

2.6.2 Augmentation

The idea is the same but $d > D$

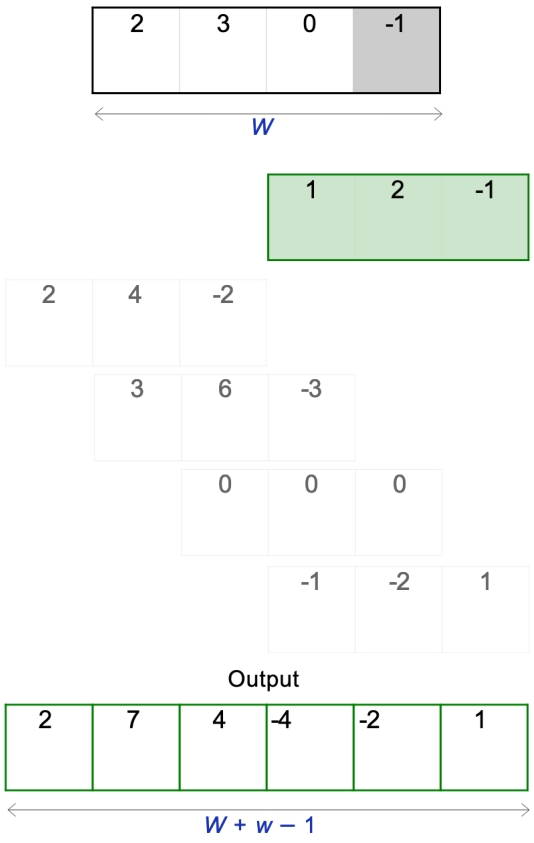


To avoid degenerate solutions and trivialities the problem is now to minimise

$$R(\mathbf{w}) = \sum_{n=1}^{N} \left[\| \hat{\mathbf{x}}_n - \mathbf{x}_n \|^2 + \lambda \sum_{i,j} \frac{\partial f_e(\mathbf{x}_n; \mathbf{w})_j}{\partial x_i} \right]$$

Remarks

- If Higher dimension can sometimes help, degenerate solutions can appear, hence the regularisation term.
- If $\lambda >> 0$ f_e becomes the identity
- If $\lambda \approx 0$ f_e becomes a constant
- To achieve reduction, a transpose convolution with stride is used.



3 Numerical optimization

3.1 Steepest Gradient

Idea Follows the direction of the steepest slope. The first order Taylor expansion yields :

$$f(\mathbf{x}+\mathbf{dx}) \approx f(\mathbf{x}) + \nabla f(\mathbf{x})^T \mathbf{dx}$$
$$f(\mathbf{x}-\lambda\nabla f(\mathbf{x})) \approx f(\mathbf{x}) - \lambda\|\nabla f(\mathbf{x})\|^2$$

$$\mathbf{x}^{(\tau+1)} = \mathbf{x}^{(\tau)} - \lambda\nabla f(\mathbf{x}^{(\tau)})$$

Remarks

- The optimal λ can be chosen at each step by linear search.
- This method is slow.

3.2 Stochastic Gradient Descent

Idea Only compute the gradient on some randomly chosen subset (change at each iteration) of the dataset. This methods has higher chance of avoiding overfitting and local minima, and enables the computation on GPU when dealing with large datasets.

$$\text{Gradient Descent:} \quad \mathbf{w}^{(\tau+1)} = \mathbf{w}^{(\tau)} - \eta \sum_{n=1}^{N} \nabla E_n(\mathbf{w}^{(\tau)})$$
$$\text{Stochastic Gradient Descent:} \quad \mathbf{w}^{(\tau+1)} = \mathbf{w}^{(\tau)} - \eta \sum_{n\in B_\tau}^{N} \nabla E_n(\mathbf{w}^{(\tau)})$$

3.3 AdaGrad

Idea Visualise the gradient descent step as a mass rolling on a hill, the slope of the function only affects the acceleration of the mass, and its speed (direction and step size) is in turn affected by this acceleration.

$$\text{Minibatch gradient} \quad \mathbf{g}_{(\tau+1)} = \sum_{n\in B_{(\tau)}} \nabla E_n(\mathbf{w}_{(\tau)})$$
$$\text{Mean gradient} \quad \mathbf{m}_{(\tau+1)} = \beta_1 \mathbf{m}_{(\tau)} + (1-\beta_1)\mathbf{g}_{(\tau+1)}$$
$$\text{Mean gradient squared} \quad \mathbf{v}_{(\tau+1)} = \beta_2 \mathbf{v}_{(\tau)} + (1-\beta_2)\mathbf{g}^2_{(\tau+1)}$$
$$\text{Corrective factor} \quad \hat{\mathbf{m}}_{(\tau+1)} = \frac{\mathbf{m}_{(\tau+1)}}{1-\beta_1}$$
$$\text{Corrective factor} \quad \hat{\mathbf{v}}_{(\tau+1)} = \frac{\mathbf{v}_{(\tau+1)}}{1-\beta_2}$$
$$\text{Gradient step} \quad \hat{\mathbf{w}}_{(\tau+1)} = \mathbf{w}_\tau - \alpha \frac{\hat{\mathbf{v}}_{(\tau+1)}}{\sqrt{\hat{\mathbf{v}}_{(\tau+1)} + \epsilon}}$$

3.4 Conjugate Gradient

Idea Go along the direction of the weighted average of the previous directions taken and of the direction of the steepest slope.

$$\alpha^{(\tau)} = \underset{a^{(\tau)}}{\operatorname{argmin}} f(\mathbf{x} + \alpha^{(\tau)}\mathbf{g}^{(\tau)})$$
$$\mathbf{x}^{(\tau+1)} = \mathbf{x}^{(\tau)} + \alpha^{(\tau)}\mathbf{g}^{(\tau)}$$
$$\beta^{(\tau)} = \frac{\|f(\mathbf{x}^{(\tau+1)})\|^2}{\|f(\mathbf{x}^{(\tau)})\|^2}$$
$$\mathbf{g}^{(\tau+1)} = -\nabla f(\mathbf{x}^{(\tau+1)}) + \beta^{(\tau)}\mathbf{g}^{(\tau)}$$

$\mathbf{g}$ is reset to the value of the gradient every n iterations.

3.5 Newton

Idea Follow the direction of the steepest descent and of the steepest rate of curvature. The second order Taylor expansion yields :

$$f(\mathbf{x}+\mathbf{dx}) = f(\mathbf{x}) + \nabla f(\mathbf{x})^T \mathbf{dx} + \frac{1}{2}\mathbf{dx}^T H_f(\mathbf{x})\mathbf{dx} \tag{2}$$

$$\nabla f(\mathbf{x}+\mathbf{dx}) = \nabla f(\mathbf{x}) + H_f(\mathbf{x})\mathbf{dx} \tag{3}$$

$$H_f(\mathbf{x})\mathbf{dx} = -\nabla f(\mathbf{x}) \tag{4}$$

$$\Rightarrow \mathbf{dx} = -H_f^{-1}(\mathbf{x})\nabla f(\mathbf{x}) \tag{5}$$

(5) in (3) yields

$$\nabla f(\mathbf{x}+\mathbf{dx}) \approx 0$$

(5) in (2) yields

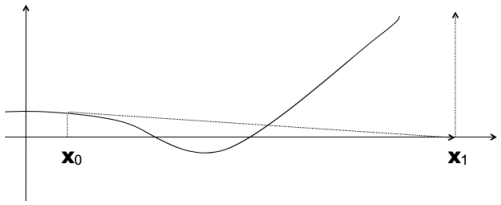
$$\begin{aligned} f(\mathbf{x}+\mathbf{dx}) &\approx f(\mathbf{x}) - \nabla f^T(\mathbf{x}) H_f^{-1}(\mathbf{x})\nabla f(\mathbf{x}) + \\ &\quad \frac{1}{2}\nabla f^T(\mathbf{x}) H_f^{-1}(\mathbf{x}) H_f(\mathbf{x}) H_f^{-1}(\mathbf{x})\nabla f(\mathbf{x}) \\ &\approx f(\mathbf{x}) - \frac{1}{2}\nabla f^T(\mathbf{x}) H_f^{-1}(\mathbf{x})\nabla f(\mathbf{x}) \end{aligned}$$

Hence, one step is performed the following way

$$\mathbf{x}^{(\tau+1)} = \mathbf{x}^{(\tau)} - H_f^{-1}(\mathbf{x}^{(\tau)})\nabla f(\mathbf{x}^{(\tau)})$$

Remarks

- Fast Convergence.
- Potential instability when $\mathbf{x}^{(\tau)}$ ends up at a relatively flat point.



3.6 Damped Newton

Idea Basically a mix between gradient descent and Newton method. With damping factor in $\mathbf{dx}$.

$$\begin{aligned} (H_f(\mathbf{x}) + \lambda I)\mathbf{dx} &= -\nabla f(\mathbf{x}) \\ \Rightarrow \mathbf{dx} &= (H_f(\mathbf{x}) + \lambda I)^{-1}\nabla f(\mathbf{x}) \end{aligned}$$

$$\mathbf{x}^{(\tau+1)} = \mathbf{x}^{(\tau)} - (H_f(\mathbf{x}^{(\tau)}) + \lambda I)^{-1}\nabla f(\mathbf{x}^{(\tau)})$$

Remarks

- Fast Convergence
- $\lambda = 0$, Regular Newton
- $\lambda >> 0$, Gradient Descent

3.7 Gauss-Newton

Gauss -Newton method is the same as the Gradient descent method, but with $f : \mathbb{R}^n \to \mathbb{R}^m$ functions

$$\begin{aligned}
\| f(\mathbf{x}+\mathbf{dx}) \|^2 &= f(\mathbf{x}+\mathbf{dx})^T f(\mathbf{x}+\mathbf{dx}) \\
\Rightarrow \quad 0 &= \frac{\partial f(\mathbf{x}+\mathbf{dx})^T f(\mathbf{x}+\mathbf{dx})}{\partial \mathbf{dx}} \\
&= \frac{\partial (f(\mathbf{x}) + J_f \mathbf{dx})^T (f(\mathbf{x}) + J_f \mathbf{dx})}{\partial \mathbf{dx}} \qquad \text{First order Taylor expansion} \\
&= J_f^T (f(\mathbf{x}) + J_f \mathbf{dx}) \\
\Rightarrow \quad J_f^T J_f \mathbf{dx} &= -J_f^T f(\mathbf{x}) \\
\Rightarrow \quad \mathbf{dx} &= -(J_f^T J_f)^{-1} J_f^T f(\mathbf{x})
\end{aligned}$$

$$\mathbf{x}^{(\tau+1)} = \mathbf{x}^{(\tau)} - (J_f^T J_f)^{-1} J_f^T f(\mathbf{x}^{(\tau)})$$

Remarks

- If the elements of f are linear in $\mathbf{x}$, the convergence is achieved in a single step.
- If it is not, the steps can be too large and the error can thus increase

3.8 Levenberg-Marquardt

Instead of using $\mathbf{dx} = -(J_f^T J_f) - 1 J_f^T f(\mathbf{x})$, we use

$$\mathbf{dx} = -(J_f^T J_f + \lambda I)^{-1} J_f^T f(\mathbf{x})$$

The value of lambda depend on the previous steps

Algorithm 4 Levenberg-Marquardt

```
function LEVENBERG-MARQUARDT(f(x))
    Initialise λ (e.g. λ = 10^-3)
    while The optimum is not reached do
        dx = -(J_f^T J_f + λI)^-1 J_f^T f(x)
        if ‖ f(x + dx) ‖ > ‖ f(x) ‖ then          ▷ Went too far
            λ = 10λ
        else
            λ = λ/10
            x = x + dx
```

Remarks

- When $\lambda >> 0$, regular gradient with a small step is performed, When $\lambda \approx 0$, A Gauss-Newton step is made
- Damped-Newton and Levenberg-Marquardt perform similarly in practice when the non-linearities are not too extreme and the initialisation point is not too bad, but Levenberg-Marquardt does not need the second order derivatives, which is better