Neural Networks - Experimental Issues

Enrique Romero

Computational Intelligence
Master in Artificial Intelligence

Soft Computing Group
Computer Science Department
Universitat Politècnica de Catalunya, Barcelona, Spain

Outline

Outline

Outline

Outline

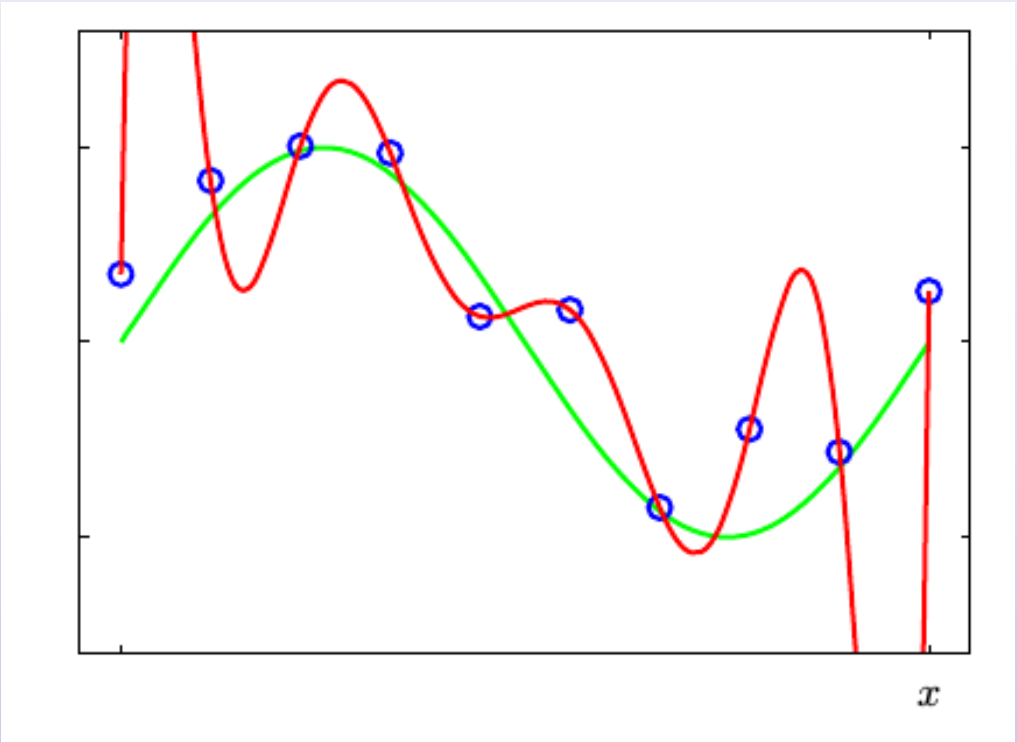
The starting point of a Machine Learning system is usually:

- A data set (labeled to some extent)
  $\boldsymbol{D} = \{(\boldsymbol{x}^{(1)}, y^{(1)}), \ldots, (\boldsymbol{x}^{(N)}, y^{(N)})\}$
- Some prior knowledge of the domain (hopefully, but not always)

The ending point of a Machine Learning system is a to construct a computational model, using the data set D, which can guarantee **good GENERALIZATION: good predictions on ANY example x of the domain**

The Main Goal of Machine Learning: Generalization

Recall that learning **FROM** data is different that learning **THE** data (discover regularities vs memorize examples): **just learning the data as much as possible usually is not a good idea**



The Main Goal of Machine Learning: Generalization

Mathematically, generalization can be described as follows.

Given:

- A "generator" of input vectors $\boldsymbol{x}$, drawn independently from a fixed but unknown distribution $P(\boldsymbol{x})$
- A "supervisor", that gives a target y for every input $\boldsymbol{x}$, drawn from a conditional distribution $P(y|\boldsymbol{x})$, also fixed but unknown
- A data set D, drawn from $P(\boldsymbol{x})$ and $P(y|\boldsymbol{x})$
- A learning algorithm, that selects a function $f_D^o(\boldsymbol{x}, \boldsymbol{\theta})$ with parameters $\boldsymbol{\theta} \in \Theta$ after being trained with D

choose the function (i.e, the set of parameters) which predicts the supervisor response as best as possible

The Main Goal of Machine Learning: Generalization

The optimality of the prediction is measured by a loss function $\mathcal{L}$, and averaged over the whole space to give the **Risk functional**:

$$R(\mathcal{L}, f_D^o, \boldsymbol{\theta}) = \int \mathcal{L}(y, f_D^o(\boldsymbol{x}, \boldsymbol{\theta}))\ dP(\boldsymbol{x}, y)$$

For example, for the the sum-of-squares loss function, we have

$$R(f_D^o, \boldsymbol{\theta}) = \int ||y - f_D^o(\boldsymbol{x}, \boldsymbol{\theta})||^2\ dP(\boldsymbol{x}, y)$$

Since we cannot compute analytically the risk, we will typically check the generalization of the model on some (maybe extra) data

Note that, **after selecting the model representation** (neural networks in our case), **the risk functional (generalization error) is only a function of the parameters θ**, and **the main goal of machine learning is selecting the best set of parameters**

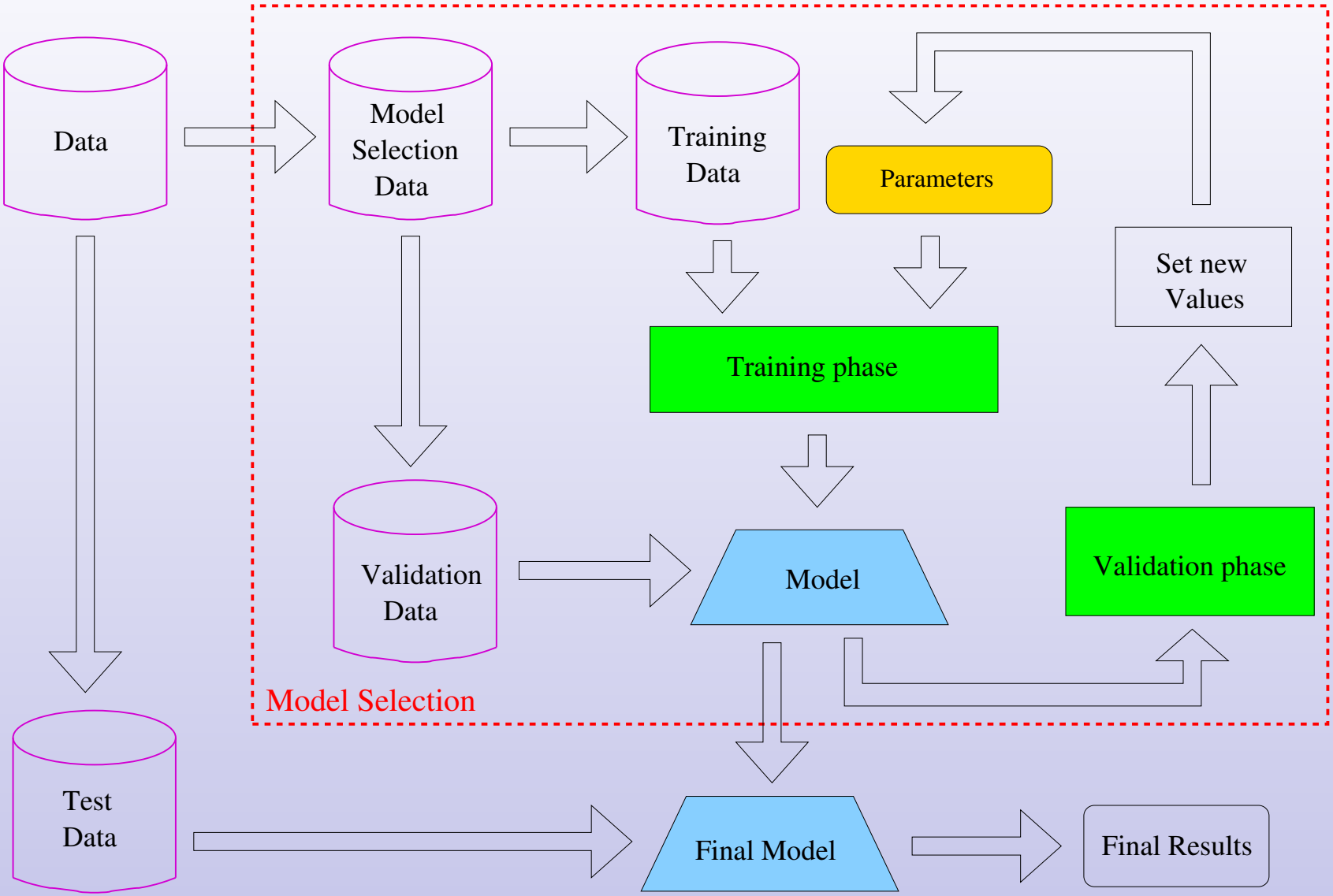
Model Selection

The process of choosing the Machine Learning algorithm (neural networks in our case) and the best parameters of the algorithm (architecture, activation functions, learning rate, etc) is called **model selection**

In the **model selection phase**, we will obtain and check a different model for every configuration of parameters

- The process of obtaining a model from a particular set of parameters is called **training phase**
- The process of checking the model obtained in the training phase is called **validation phase**
- Typically, different configuration of parameters are trained and validated with different data sets

Model Selection (Typical Scheme)



Outline

Underfitting and Overfitting

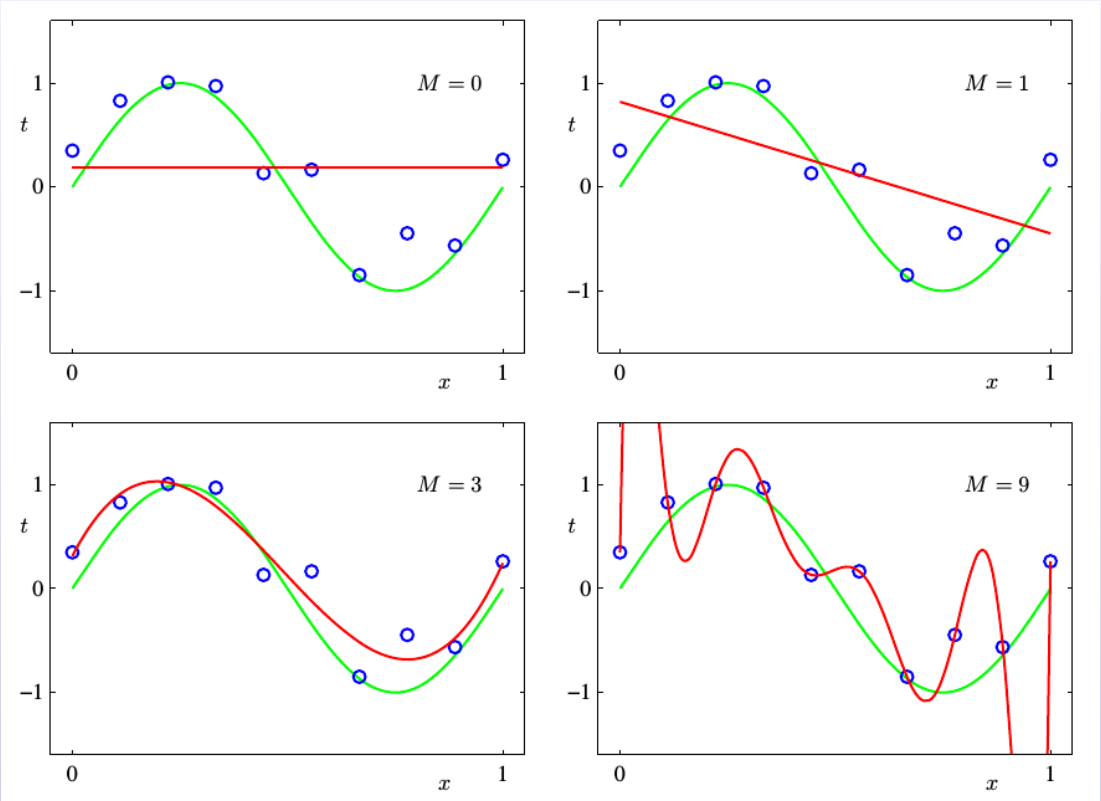
In order to obtain the best generalization, we have to search for the best possible model and parameters (model selection)

Bad models typically have

- **Underfitting**, when they fit poorly the data, so that they have "relevant things to learn yet"
- **Overfitting**, when they fit too much the data, so that they have "learned irrelevant or noisy things"

Underfitting and Overfitting

An example of underfitting ($M = 0$ and $M = 1$) and overfitting ($M = 9$) for polynomial approximation with different orders M:

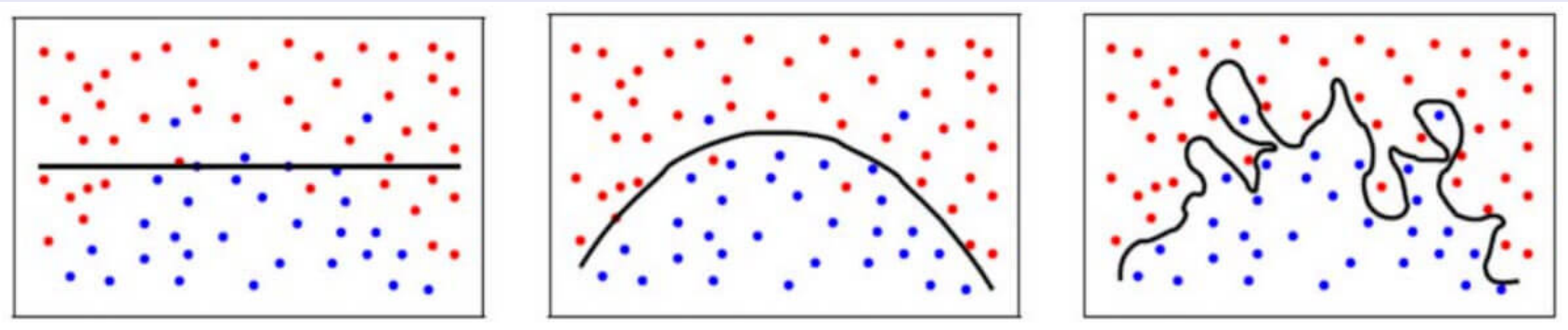


Data (blue circles) generated adding noise to several points in the green curve

The best fit corresponds to $M = 3$

Underfitting and Overfitting

An example of underfitting (left) and overfitting (right) for classification problems:



The best fit corresponds to the middle image

Issues Related to Underfitting and Overfitting

There are several issues related to the problems of underfitting and overfitting:

- The **model complexity**, related to the **bias-variance decomposition**
- The **amount of training data**, related to the **curse of dimensionality**
- The amount of **noise** in the data

These issues will be discussed in the next sections

Outline

The Bias-Variance Decomposition

The Bias-Variance decomposition for the sum-of-squares loss function [Geman et al., 1992]

$$E_D\left[\, R(\mathcal{L}^2, f_D^o) \,\right] = E_X\left[\, E_Y\left[\, (\, y - E_Y[\, y|x \,]\,)^2 \,\right]\,\right] + E_X\left[\, (\, E_D[\, f_D^o(x) \,] - E_Y[\, y|x \,]\,)^2 \,\right] + E_X\left[\, E_D\left[\, (\, f_D^o(x) - E_D[\, f_D^o(x) \,]\,)^2 \,\right]\,\right].$$

- Model: f_D^o (output function trained with a data set D)
- $R(\mathcal{L}^2, f_D^o) = E_{X\times Y}\left[\, (y - f_D^o(x))^2 \,\right]$ (expected risk)
- First term: **independent of the model**
- Second term: **bias**, related to the approximation capability of the model: high approximation capability may have small bias
- Third term: **variance**, related to the complexity of the model: high complexity may have large variance
- **Bias and variance go in opposite directions**

The Bias-Variance Decomposition

Therefore,

- **Models with (too) low complexity, that are prone to have high bias, tend to underfit**
- **Models with (too) high complexity, that are prone to have high variance, tend to overfit**

How can we measure the complexity of Neural Networks? In general (although not independently),

- Models with more hidden units are more complex
- Models with more layers are more complex
- Models with highly non-linear functions are more complex
- Models with larger weights are more complex

Controlling the Complexity of the Model

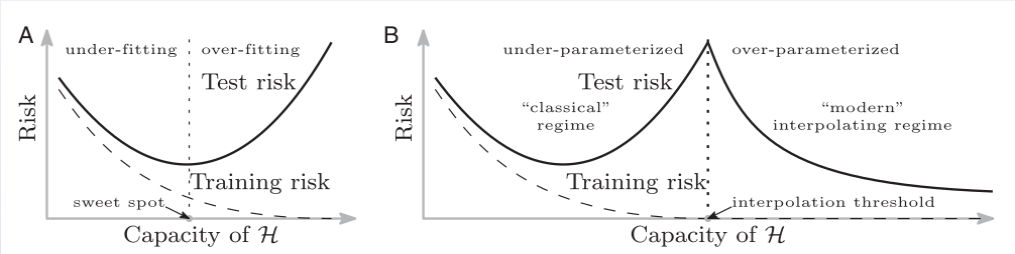
There are many ways to control the complexity of the model (it is still an active area of research)

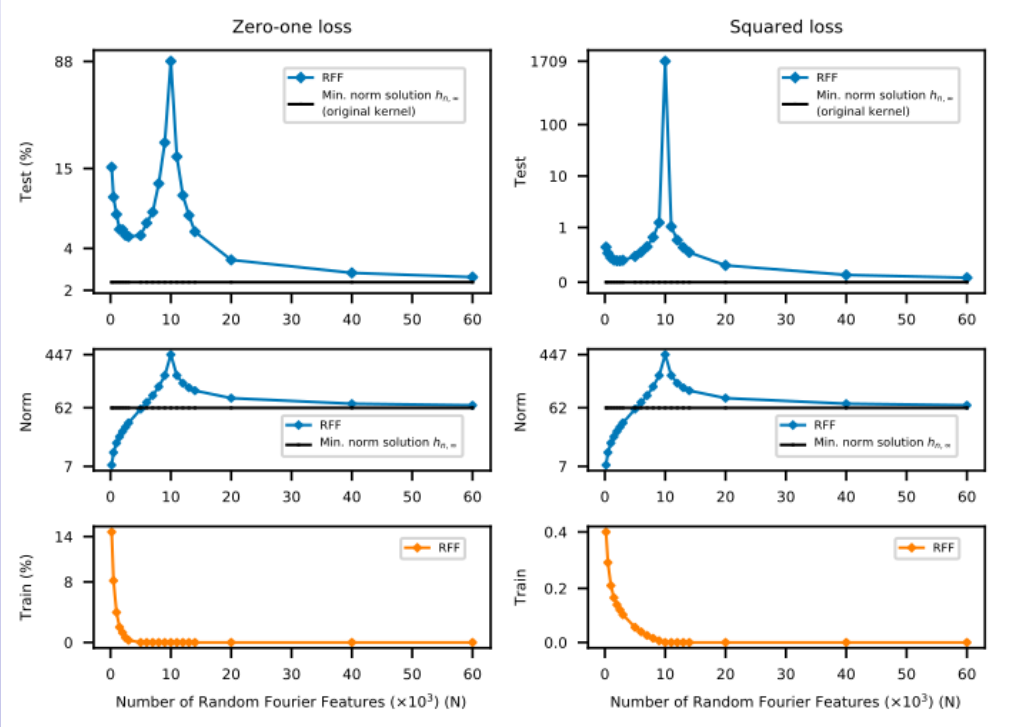
For Neural Networks, we have several forms of **regularization**:

- Weight constraints (limiting the size,...)
- Weight sharing
- Adding noise (to the weights, activities,...)
- L_2 regularization
- L_1 regularization (not differenciable)
- Early stopping
- Dropout
- Batch normalization
- etc

The Double Descent Curve

It has been shown that controlling the complexity of Neural Networks (and other models) plays a crucial role in generalization, although in a non-trivial way [Belkin et al., 2019]:





Outline

The Curse of Dimensionality

The curse of dimensionality [Bellman, 1961] is related to the fact that the number of examples needed to reasonably "cover" the space grows **exponentially** with the input dimension

Therefore, **for high-dimensional spaces**, it is impossible to manage (or even store) such a huge quantity of data

For example, imagine that we are working in $[0, 1]^d$ (gray levels of an image) and we want to have 10 levels of gray in each dimension

In general, we need 10^d examples

- For $d = 2$, we only need $10^2 = 100$ examples
- ...
- But for $d = 100$ we need 10^{100} examples (more than the number of atoms in the universe!)

The Curse of Dimensionality

Usually, models based on few examples tend to overfit, since the input space is not properly represented

In general, generalization will improve as the amount of data increases, but only up to a certain point:

- As previously mentioned, in high-dimensional spaces it is usually impossible to obtain enough data
- The complexity of the model may be too high for any reasonable amount of data
- Data may be noisy, so that there will be errors that can not be avoided

Most times, however, we can not choose the amount of data

Outline

Noise in the Data

Noise can be present in the data in many different forms:

- Noisy input values (for example, bad measurements or large tolerances of measure systems)
- Missing values
- Mislabeled examples (for example, labels assigned automatically to large data sets)
- Variables (features) irrelevant for the problem at hand (which is also related to the curse of dimensionality)

Additionally, it is very difficult to know if there are missing relevant features

Noise in the Data

The noise in the data may cause underfitting or overfitting, depending on the type of noise, the complexity of the model and the amount of available data

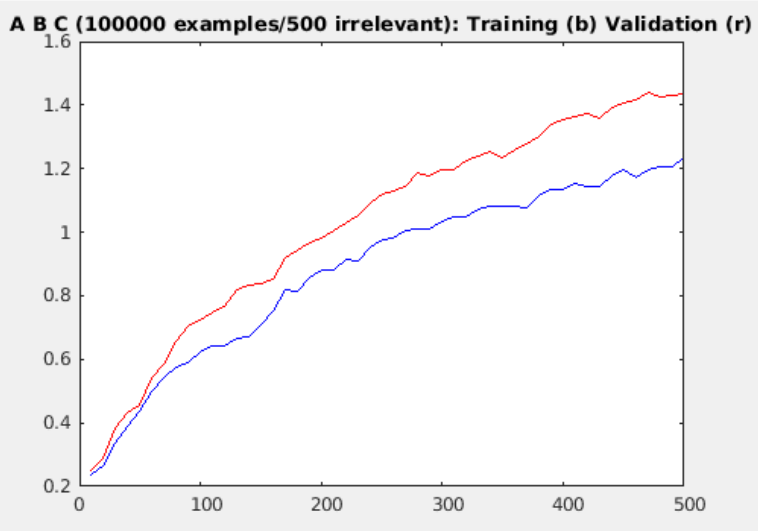
In general, generalization will improve as the amount of noise decreases

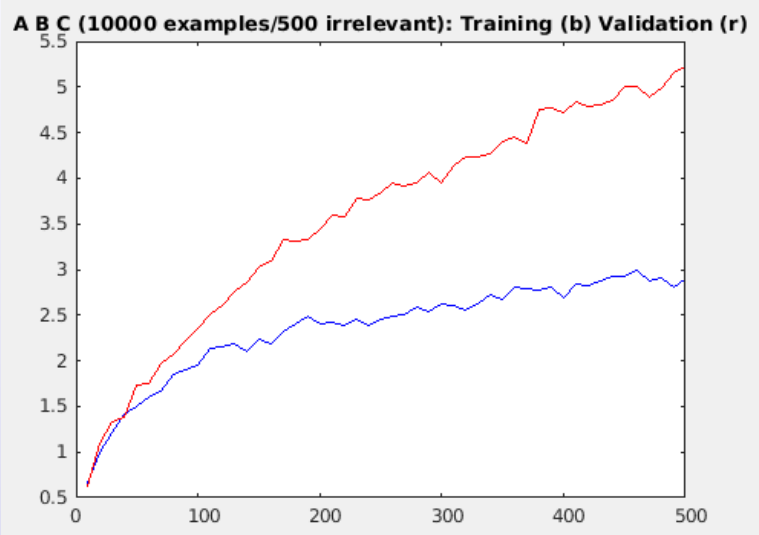
It is very difficult to remove the noise in the data:

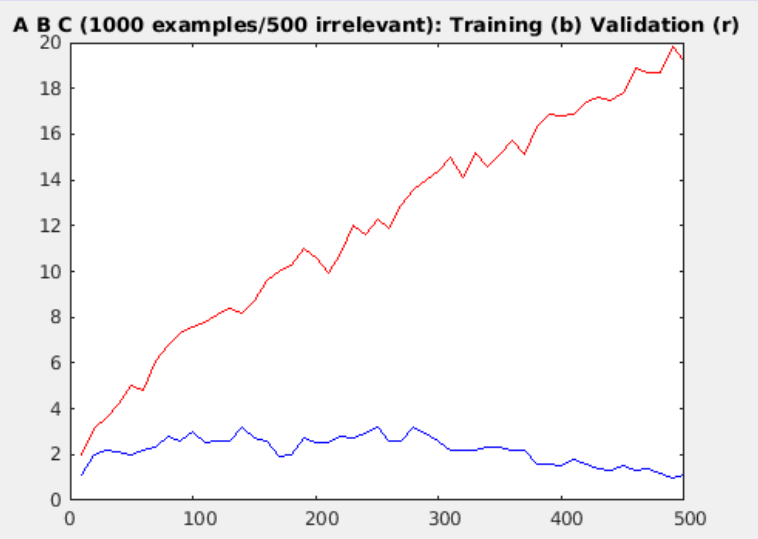
- A standard analysis of the data (means, variances, etc) may be useful to detect very noisy input values
- Techniques of outliers detection can be used to look for mislabeled or out-of-range examples
- Feature Selection aims to select the best subset of features for the task (hopefully removing irrelevant features)

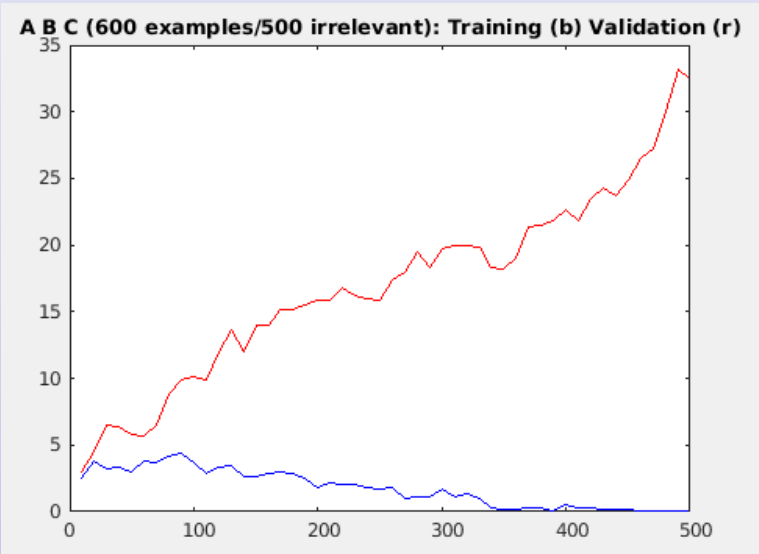
The Effect of Irrelevant Features

Error evolution when adding 500 irrelevant variables:









The Effect of Irrelevant Features

Typically we observe that:

- When **many examples** are available, adding irrelevant features "only" **increases both training and test error**
- When **few examples** are available, adding irrelevant features **decreases the training error but increases the test error**

Note that

- It is impossible to know how many irrelevant variables we have
- It is impossible to know if we have enough training examples

Outline

Model Selection (revisited)

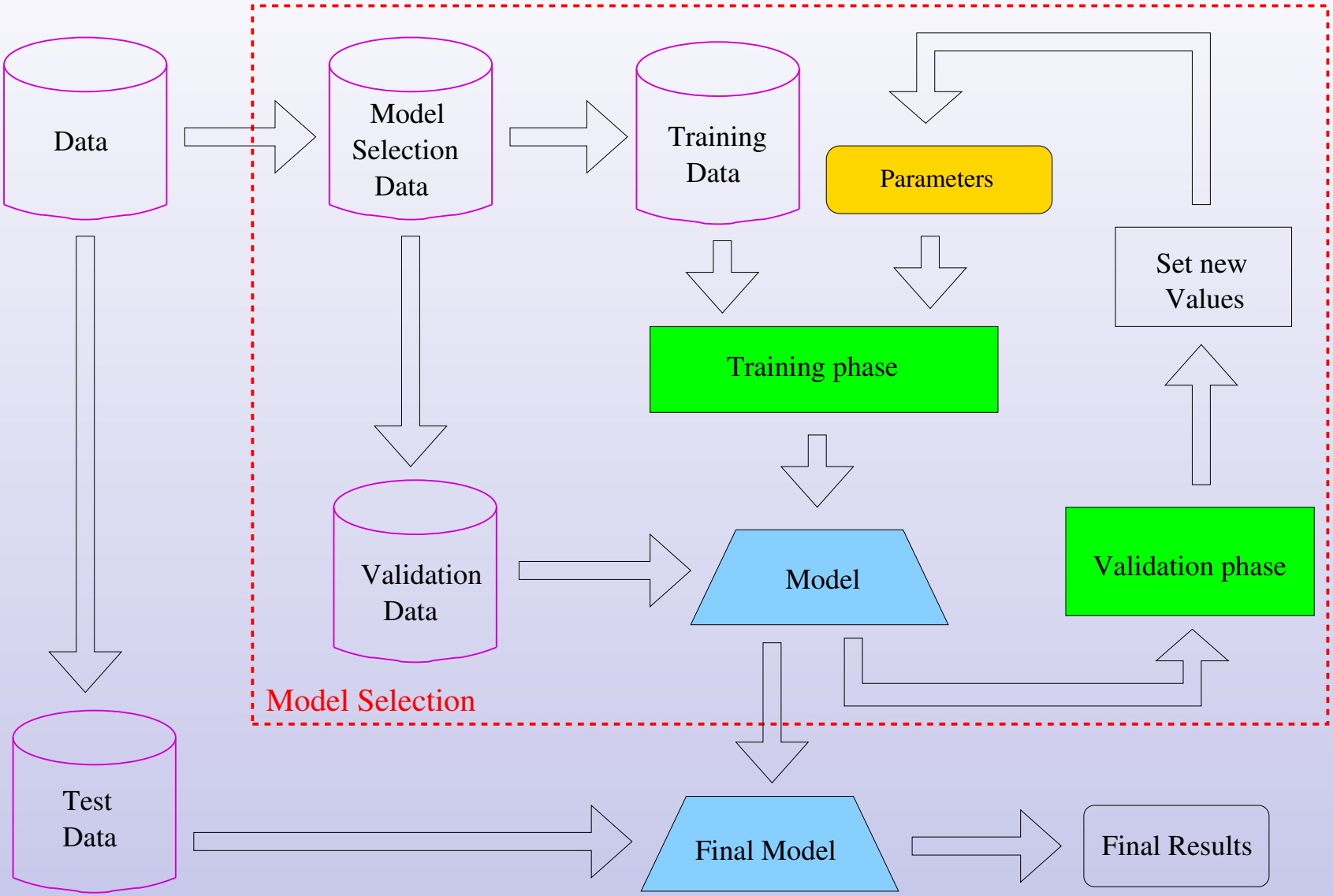
Recall that **model selection** is the process of choosing the Machine Learning algorithm (neural networks in our case) and the best parameters of the algorithm (architecture, activation functions, learning rate, etc)

In the model selection phase, we will have to

- Train a network with a particular set of parameters (**training phase**)
- Check the model obtained after the training phase (**validation phase**)

After the model has been selected, we have to categorize correctly new examples (**test phase**) different from those used in the model selection phase, in order to estimate its generalization

Model Selection (Typical Scheme)



Data Sets for Model Selection

Each of the phases of the model selection needs a data set:

- A **training set**, used to train the model (training phase)
- A **validation set**, used to check the model (validation phase)

Subsequently, a **test set** is used to check the model (test phase)

Obviously, **data sets in the training and validation phases must be disjoint**

Ideally, **test data can not be used in the model selection phase**

Typically, **different configuration of parameters are trained and validated with different data sets (shuffling)**

How can we fulfill all these requirements and obtain reliable (statistically valid) results?

Typical Techniques for Model Selection

In the ideal case, when we have a very large amount of data, we simply can do the following:

- Split the whole data into two data sets: model selection and test (these two data sets will still be large)
- In the model selection process, for every configuration of the parameters, split the model selection data set into different training and validation data sets (that will still be large)
- After the model is selected, train with the model selection data set and check with the test set

This technique is known as **holdout cross-validation** (or simply holdout) [Highleyman, 1962]

Typical Techniques for Model Selection

Several comments on the holdout method:

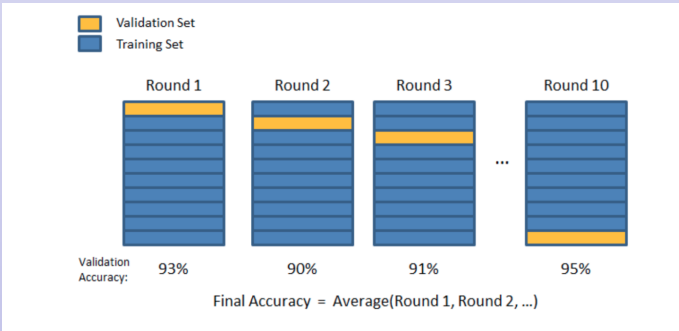
- The split of the data sets can be completely random or include some form of stratification
- Typical percentages for splitting (arguably): $70\%, 15\%, 15\%$
- **Obviously, there will be variations in the results due to the randomness of the examples in the data sets**
- We can repeat the process several times (with different splits both in the model selection phase and in the test phase) and take the means

Typical Techniques for Model Selection

When we do not have a very large amount of data, things are not so easy, because we would like to use as much of the available data as possible for training (keeping aside part of the data set can be considered a luxury that we may not be able to afford)

In this cases, we can use **k-fold cross-validation** [Cover, 1969, Stone, 1974] for the model selection phase:

- The training set is divided into k distinct folds
- We then train a network using data from $k-1$ of the folds and validate it with the remaining fold
- This process is repeated for each of the k possible folds
- Take the mean over all k folds



Typical Techniques for Model Selection

Several comments on the k-fold cross-validation method:

- Again, the split of the data in folds can be completely random or include some form of stratification
- Typical choices for k: 5, 10
- Advantages: allows to use a high proportion of the available data (a fraction $1 - 1/k$) to train the model, while also making use of all data points for validation
- Drawbacks: it requires the training process to be repeated k times **for every combination of parameters** (and we may have many combinations!)

Typical Techniques for Model Selection

Several variations of k-fold cross-validation:

- If data is very scarce we could go to the extreme limit of $k = N$ for a data set with N examples, which involves N separate training phases, each using $N - 1$ examples (**leave-one-out cross-validation)**
- If we want to include a test phase in the whole procedure, we can perform a **double k-m-fold cross-validation** [Ripley, 1995, Ripley, 1996]:
  - Perform a k-fold cross-validation to obtain k folds ($k - 1$ folds for model selection and and 1 fold to test)
  - Then, perform a m-fold cross-validation on each of the k folds ($m - 1$ for training and 1 fold for validation)
  - Take the means over all folds

Alternatives to cross-validation: **bootstrapping** [Ripley, 1996]

Outline

Data Pre-processing

In principle, the **raw** input data can be used to obtain the required final target values

In practice, except for very simple problems, this will generally give poor results

For most data sets, it is necessary first to transform the data into some new representation before training

This is specially important for neural networks: in many practical applications the choice of pre-processing will be one of the most significant factors in determining the performance of the final system

Data Pre-processing

There are many ways to pre-process the data

- Linear transformations (often useful if different variables have values which differ significantly)
  - Transformed each variable so as to have zero mean and unit standard deviation (**z-scores**): $x_{\widetilde{v}}^{(i)} = (x_v^{(i)} - \overline{x_v})/\sigma_v$
  - **Whitening**: similar to z-score but taking into account the covariance matrix so as to obtain uncorrelated features
  - **Linear scaling** to $[0, 1]$:
    $$x_{\widetilde{v}}^{(i)} = (x_v^{(i)} - \min_i (x_v^{(i)}))/(\max_i (x_v^{(i)}) - \min_i (x_v^{(i)}))$$
- Non-linear transformations (Feature extraction):
  - Polynomial features: $x_{v_n}^{(i)} x_{v_m}^{(i)}$, etc
  - Non-linear filters, etc
- Dimensionality reduction:
  - Principal Component Analysis
  - Feature selection

It will generally be convenient to pre-process the whole data set, and then use this transformed data set to train, validate and test the model

Outline

Error Measures

How is generalization measured? With an error measure

It depends on the type problem (regression, classification, ...) and the particular setting of the problem:

- For regression problems, the most usual error measure is the sum-of-squares error and variations (normalized, relative, etc)
- For classification problems, there are many error measures, most derived from the **confusion matrix** (`https://en.wikipedia.org/wiki/Confusion_matrix`)
  - Accuracy and Balanced Accuracy
  - Specificity (True Negative Rate)
  - Sensitivity (Recall, True Positive Rate)
  - Precision
  - $F1 = 2 \cdot \text{Precision} \cdot \text{Recall} / (\text{Precision} + \text{Recall})$
  - False Negative Rate and False Positive Rate
  - AUC

Some of them are very useful when we have **unbalanced classes**

Outline

Notebook Example Multilayer Perceptrons

Run:

- BasicExample_SLP-MLP_Classification-Regression.ipynb

Things that you can test:

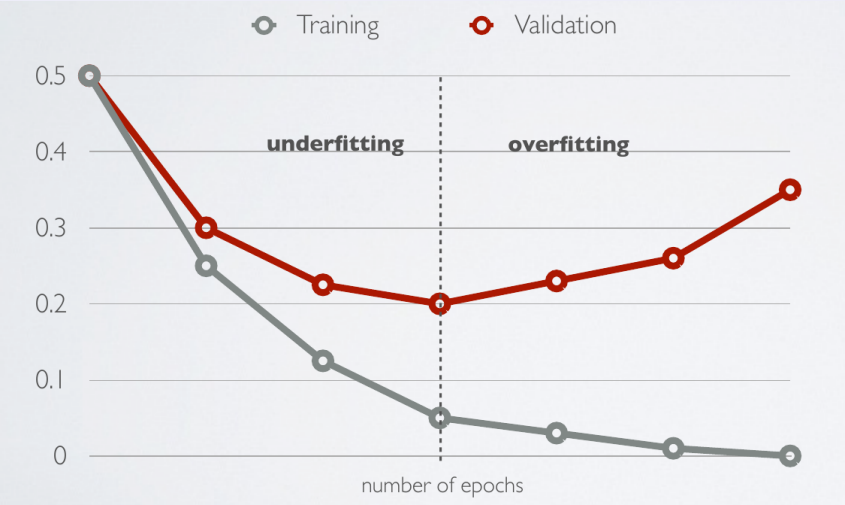
- Change the complexity of the model (SLP vs MLP)
- Change the percentage of data for training
- Change the pre-processing of the data

Outline

Outline

Diagnosing Underfitting and Overfitting

In Neural Networks, the best way to detect the presence of underfitting or overfitting is by plotting the error in the training and validation sets as a function of the number of updates



Typically,

- When we have underfitting, training errors are large and validation errors are similar to training errors
- When we have overfitting, training errors are small and there is a large gap between validation and training errors

Diagnosing Underfitting and Overfitting

In order to reduce the underfitting you can increase the complexity of the model

- Increasing the complexity of the architecture
- Performing more training updates
- ...

In order to reduce the overfitting you can decrease the complexity of the model

- Reducing the complexity of the architecture
- Reducing the number of features (Feature Selection, see above)
- Using early stopping or some other form of regularization (see below)

Outline

Stochastic Gradient Descent

The best estimation of the gradient is obtained by averaging the gradient of all examples

However, we have to take into account that

- We know that the gradient direction may not be (and usually is not) the best direction of descent, so that there is no reason to spend a lot of time to estimate it precisely
- It may be better to explore in a different way the parameter space by changing the parameters more frequently

Stochastic Gradient Descent

In practice, therefore, we use **mini-batch updates**, based on an average of the gradient in a **mini-batch** of B examples:

$$g = \frac{1}{B} \sum_{(\boldsymbol{x},y)\in \text{Batch}} \frac{\partial \mathcal{L}(y, f_D^o(\boldsymbol{x}, \boldsymbol{\theta}))}{\partial \boldsymbol{\theta}}$$

Several comments:

- When $B = 1$, we have **Online Gradient Descent**
- When B is the training set size, we have **Batch (Standard) Gradient Descent**
- For small values of B, updates are done more frequently than for larger values, but with less precision

This technique is called **Stochastic Gradient Descent**

Stochastic Gradient Descent

A general (simplified) algorithm for SGD looks like

Input: (data set $\boldsymbol{D}$, learning rate $\lambda > 0$)
Initialize $\boldsymbol{\theta}_0$
$t = 0$
Repeat until convergence:
  Select the minibatches $\boldsymbol{B}_i \subseteq \boldsymbol{D}$
  For every batch $\boldsymbol{B}_i$
    Compute the gradient $g_t = \partial J(\boldsymbol{\theta}_t, \boldsymbol{B}_i)/\partial \boldsymbol{\theta}$
    $\boldsymbol{\theta}_{t+1} = \boldsymbol{\theta}_t - \lambda\, g_t$
    $t = t + 1$
Return $\boldsymbol{\theta}_t$

Stochastic Gradient Descent

In practice, **is really important to use Stochastic Gradient Descent (SDG) in order to get reasonable convergence speeds for large data sets**:

- Experience demonstrates that SGD converges much faster than Batch Gradient Descent for large data sets, and it does not usually depend too much on the size of the batch
- For very large data sets, updating the parameters after seeing all the examples is hopeless and computationally expensive
- Small values of B benefit from more exploration of the parameter space (maybe we have a form of regularization due to the "noise" injected in the gradient estimator)

Typical values for B range between 16 and 128 (no general rule), and it is better to change the order in which the mini-batches are visited in different epochs

Outline

Parameters

Training of Neural Networks involves selecting a (usually large) number of parameters for the different issues involved:

- The model representation (architecture): number of layers, number of hidden units, connections, non-linearities, etc
- The cost function: squared error, cross-entropy, etc
- The regularization scheme: early stopping, regularization coefficients, dropout, etc
- The optimization technique (usually SGD): mini-batch size, number of iterations, learning rates, momentum, weight initialization, etc

A proper selection of these parameters will strongly affect the final performance of the model

Outline

Parameters for the Architecture

Regarding the **number of hidden layers** and **number of hidden units** in every hidden layer

- They are related with the complexity of the network
- The optimal configuration is problem-dependent, and is not independent of other parameters (connections, non-linearities, regularization)
- For a long time, it was thought that the optimal solution should have the minimum number of layers and units
- However, **recent results suggest that it is better to choose a large architecture (deep, wide or both) and apply strong regularization techniques**

Regarding the **connections**, typical MLPs architectures are fully connected, sometimes with direct connections from the input units (Convolutional Neural Networks are not fully connected)

Parameters for the Architecture

Regarding the **non-linearity of the hidden units**

- Typical non-linear functions are sigmoid ones, such as the logistic function

$$f(z) = \frac{1}{1 + e^{-z}}$$

and the hyperbolic tangent function

$$f(z) = \frac{e^{z} - e^{-z}}{e^{z} + e^{-z}} = \frac{2}{1 + e^{-2z}} - 1$$

- Recently, rectifier non-linearities, such as the rectified linear

$$f(z) = \max(0, z)$$

or the softplus function

$$f(z) = \log(1 + e^{z})$$

have been very successful in several applications, since they avoid the vanishing gradient problem in some cases

Parameters for the Architecture

The **activation function of the output units** is typically associated to the cost function

- Squared error and linear output units (regression problems)
- Cross-entropy and softmax output units (standard classification problems)
- etc

Sigmoidal functions (together with target values normalized to their range) are useful in some cases, but show optimization difficulties in other ones

Additionally, the error propagation must be taken into account in order to choose the activation function of the output units (for example, rectified linear units can not propagate the gradient when the unit is saturated, i.e., there is no chance to correct the error)

Outline

Parameters for Regularization

Clearly, it depends on the regularization scheme chosen:

- **Early stopping** estimates the generalization error on a validation data set (see above), and decides to stop the training when either overfitting or no progress is observed (which may also depend on other parameters)
- L_2 **regularization** adds a term $\lambda \sum \theta_i^2$ to the cost function, penalizing large values of the parameters (depending on λ)
- L_1 **regularization** (not differenciable) adds a term $\lambda \sum |\theta_i|$ to the cost function, encouraging sparsity of the parameters
- The probability parameter for **Dropout** (see below)
- **Batch normalization** does not have any parameter (see below)

Parameters for Regularization

Dropout [Srivastava et al., 2014] is based on the following idea:

- For every training case, **hidden units are randomly omitted from the network with probability p (typically 0.5)**
- At test time, the "mean network" that contains all of the hidden units is used but **with their outgoing weights scaled by $1 - p$** to compensate for the training dropout

An interpretation of the dropout procedure is that it is a very efficient way of performing **model averaging with neural networks**: There is almost certainly a different network for each presentation of each training case but all of these networks share the same weights for the same hidden units

The winner of the ImageNet Large Scale Visual Recognition Challenge 2012 used Dropout [Krizhevsky et al., 2012]

Parameters for Regularization

The idea behind **Batch normalization** [Ioffe and Szegedy, 2015] is to normalize the inputs to every unit as follows:

- Each dimension is linearly transformed so as to have zero mean and unit standard deviation (z-score):

$$\widetilde{\boldsymbol{x}}_j = (\boldsymbol{x}_j - \overline{\boldsymbol{x}_j})/\sigma_j$$

- Linear weights are added so that the original values $\boldsymbol{x}_j$ can be restored

$$\widetilde{y}_j = \gamma_j \, \widetilde{\boldsymbol{x}}_j + \beta_j$$

(these new weights are learned along with the rest of weights)

After a few simplifications and readjustments, the work in [Ioffe and Szegedy, 2015] obtained new state-of-the-art results on the ImageNet Large Scale Visual Recognition Challenge 2014

Outline

Parameters for Stochastic Gradient Descent

A general (simplified) algorithm for SGD looks like

Input: (data set $\boldsymbol{D}$, learning rate $\lambda > 0$)
Initialize $\boldsymbol{\theta}_0$
$t = 0$
Repeat until convergence:
  Select the minibatches $\boldsymbol{B}_i \subseteq \boldsymbol{D}$
  For every batch $\boldsymbol{B}_i$
    Compute the gradient $g_t = \partial J(\boldsymbol{\theta}_t, \boldsymbol{B}_i)/\partial \boldsymbol{\theta}$
    $\boldsymbol{\theta}_{t+1} = \boldsymbol{\theta}_t - \lambda\, g_t$
    $t = t + 1$
Return $\boldsymbol{\theta}_t$

Parameters for Stochastic Gradient Descent

Regarding the **mini-batch size**

- As already mentioned, typical values range between 16 and 128, but there is no general rule
- The impact of this parameter is mostly computational: it should impact training time and not so much generalization

The **number of iterations**

- Can be measured as the number of epochs (an epoch is a complete iteration through the training set) or as the number of (mini-batch) updates
- Is strongly related to early stopping
- In practice, however, it could be convenient to turn early stopping off when analyzing the effect of individual parameters (and therefore train more than necessary) so as to have a clearer idea of the error curves

Parameters for Stochastic Gradient Descent

The **learning rate is usually the most important parameter to tune**

- Typical values for a network with standardized inputs are less than 1 and larger than 10^{-6}
- The optimal learning rate is usually close to (by a factor of 2) the largest learning rate that does not makes training diverge [Bengio, 2012]
- It may change during training with a **learning rate schedule**: linear decaying, exponential decaying, etc

Parameters for Stochastic Gradient Descent

The **momentum method** [Polyak, 1964] is a technique for accelerating and smoothing out the gradient descent procedure

It **accumulates** a "velocity vector" with **the updates of the previous iterations**, and determines the next update as a linear combination of the actual gradient and this velocity vector:

In the previous algorithm, change

$$\boldsymbol{\theta}_{t+1} = \boldsymbol{\theta}_t - \lambda \; g_t$$

by (initializing $\boldsymbol{v}_0 = 0$)

$$\boldsymbol{v}_{t+1} = \mu \; \boldsymbol{v}_t + (1 - \mu) \; g_t$$
$$\boldsymbol{\theta}_{t+1} = \boldsymbol{\theta}_t - \lambda \; \boldsymbol{v}_{t+1}$$

or, equivalently ($\mu' = \mu$ and $\lambda' = \lambda(1 - \mu)$)

$$\boldsymbol{v}_{t+1} = \mu' \; \boldsymbol{v}_t - \lambda' \; g_t$$
$$\boldsymbol{\theta}_{t+1} = \boldsymbol{\theta}_t + \boldsymbol{v}_{t+1}$$

Parameters for Stochastic Gradient Descent

Under certain hypothesis, it can be proved that the momentum method can accelerate the convergence of gradient descent [Polyak, 1964]

The momentum method needs a parameter μ that determines the weight that is given to the accumulated updates

- Its value must be in $[0, 1]$
- Similar to the learning rate, its value may change during training

There are also other variants for the momentum method, such as Nesterov momentum, etc

Parameters for Stochastic Gradient Descent

Biases and output weights can generally be initialized to 0, but the rest of the weights need to be carefully initialized

- To break symmetries between hidden units of the same layer
- To be small enough so as to not saturate neither the forward nor the backpropagation phase

A quite popular scheme for **weights initialization** is known as **Xavier initialization** [Glorot and Bengio, 2010], which initializes the weights randomly so as to have variance proportional to $1/\sqrt{\text{fan-in} + \text{fan-out}}$

A completely different approach is to use **unsupervised pre-training** to initialize the weights: an unsupervised model (such as a Restricted Boltzmann Machine or an Auto-encoder) is trained with the data, and the final weights of the unsupervised procedure are used as the initial weights of the supervised one

Outline

Other Variants for Gradient Descent

There are many variants to the standard Gradient Descent procedure

Several recent approaches are

- RMSProp
- Adagrad
- Adadelta
- Adam
- etc

which are basically based on using an **individual and adaptive learning rate** for every parameter

More info: `http://ruder.io/optimizing-gradient-descent`

Outline

Notebook Example Multilayer Perceptrons

- WorkingExample_SLP-MLP_Classification.ipynb

Outline

Alternatives to Gradient Descent

There are a number of alternatives to Gradient Descent as the optimization technique for Neural Networks:

- Conjugate Gradients
- Broyden-Fletcher-Goldfarb-Shanno (BFGS) and limited-memory BFGS (L-BFGS)
- Davidon-Fletcher-Powell
- Levenberg-Marquardt
- Powell's dogleg
- etc

Anyway, all these methods need to compute the gradient of the cost function, that will be most likely computed with the Backpropagation algorithm

Outline

Strategies for Searching the Parameters

As we have seen, the training of a Neural Network usually involves the choice of **many (hyper) parameters**, which:

- May have many possible values
- Cannot be adjusted with a gradient descent procedure (not to be confused with the weights of the network, which are the parameters of the output function computed by the network)

This leads to an **exponential and complex search process**, which is computationally unfeasible

There are, however, several guidelines and strategies that can be used to alleviate and facilitate the selection of parameters

General Guidelines for Searching the Parameters

When searching for the value of a **single parameter** (being the rest fixed)

- It is recommended to set a **starting interval**, and choosing the **intermediate values linearly in the log-domain** (instead of linearly in the interval): $10^{-1}, 10^{-2}, 10^{-3}, ...$
- We expect a kind of a U-shape curve, although there may be noisy variations
- If the best value is near the border, it indicates that we should explore beyond this border
- There is no point in exploring the effect of small changes until a reasonable good setting has been found: we can start with a few values in the interval, and then refine the search around the best value found so far (**multi-resolution search**)

Grid Search and Coordinate Descent

After the intervals for each parameter have been set, a **grid search** exhaustively explores all the combinations of these values

- Advantages: it can be done in parallel
- Disadvantages: it is exponential in the number of parameters

When performing a manual search with access to a single computer, a reasonable strategy is **coordinate descent**: change only one parameter at a time, always making a change from the best configuration found so far

Typically one needs to perform several refinements, and subsequent refinements may fix several parameters and only search for a subset (maybe with a different strategy than the previous search)

Random Sampling of Parameters

The idea of **random sampling of parameters** is to replace manual or grid search by a random sampling [Bergstra and Bengio, 2012]

- The easiest way is to perform a uniform sampling in the interval of each parameter
- More complex sampling can be performed by including our prior beliefs on the likely good values, so that different values for the same parameter may have different probabilities, using Bayesian optimization with Gaussian processes [Snoek et al., 2012]

Uniform Random Sampling of Parameters

Several situations where using a uniform random search may be efficient

- When there is a *large subset of parameters* close to the optimal one
- When only a small subset of parameters matter

Additionally, if we plot the curves of validation error as a function of the number of trials, we can early stop the search when there are no improvements

Bayesian Optimization of the Parameters

Bayesian optimization is an iterative algorithm for global optimization of black-box functions [Mockus, 1974]:

$$\omega^* = arg\ max_{\omega \in \Omega}\ f(\omega)$$

where Ω is a certain space of interest

Since the objective function is unknown, the Bayesian strategy is to treat it as a random function

The main ingredients of Bayesian optimization are

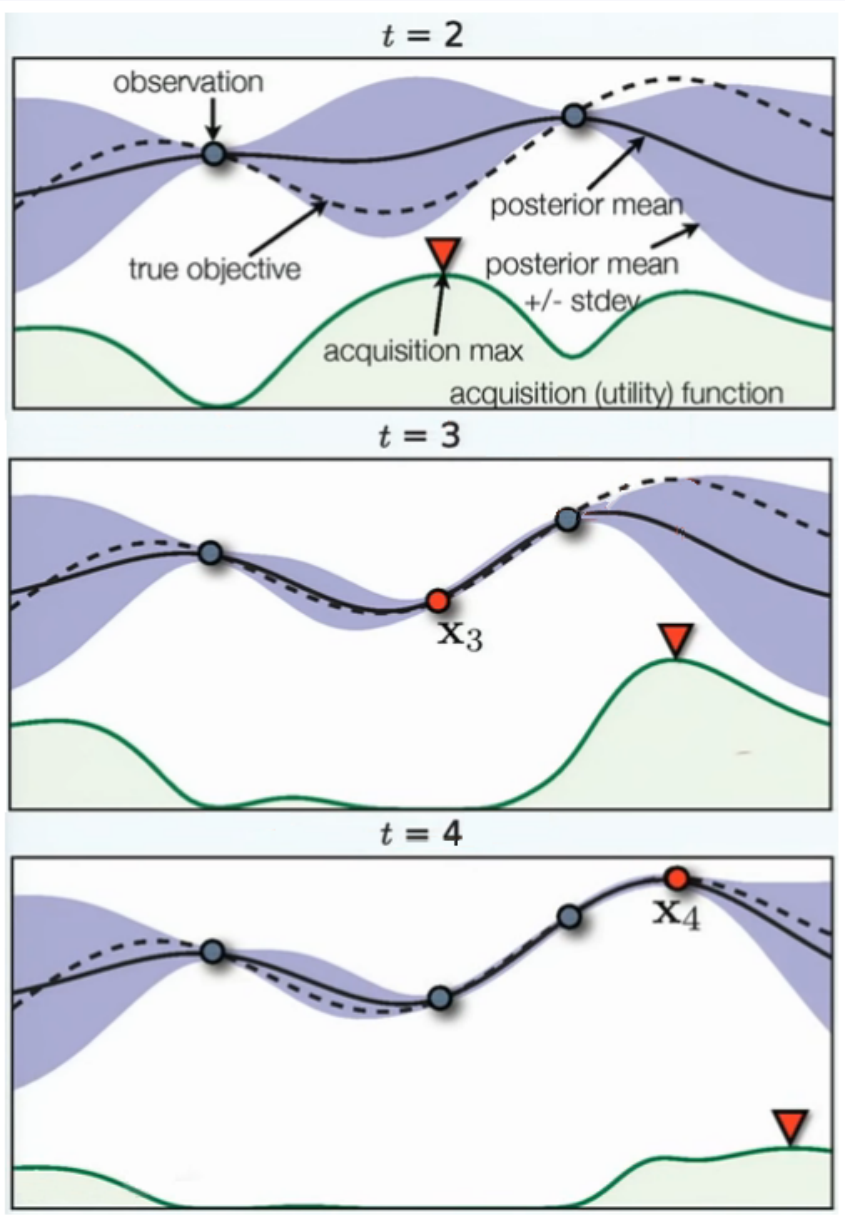
- The hypothesis that f can be evaluated at any point $\omega \in \Omega$
- A prior belief $p(\omega; D)$ about the function f, which is refined (as new data D are observed) via Bayesian updating to obtain a posterior distribution (updated belief)
- An acquisition function $\alpha(\omega; p, D)$ that, based on the posterior distribution and the data, is optimized in order to obtain the next candidate ω_n at iteration n

Bayesian Optimization of the Parameters

A general algorithm for Bayesian optimization looks like

Define a prior distribution $p(\omega; D)$ over Ω
Define an acquisition function $\alpha(\omega; p, D)$
$D_1 = \emptyset$
For $n = 1, 2, \ldots$ **do**:
  Select ω_n by optimizing the acquisition function $\alpha(\omega; p, D_n)$
    // After having seen D_n, the most promising candidate
    // (according to α) to maximize f is ω_n
  Compute $f_n = f(\omega_n)$ // real value of the function in ω_n
  Augment the data set $D_{n+1} = D_n \cup \{\omega_n, f_n\}$
  Update distribution $p(\omega; D)$ with D_{n+1}
Return the best ω^* found in the loop

Bayesian Optimization of the Parameters



Outline

Further Information

See [Bengio, 2012] for further information

That's it!

Bibliography

- Belkin, M., Hsu, D., Ma, S., and Mandal, S. (2019). Reconciling Modern Machine-Learning Practice and the Classical BiasVariance Trade-off. *Proceedings of the National Academy of Sciences*, 116(32):15849–15854.
- Bellman, R. (1961). *Adaptive Control Processes: A Guided Tour*. Princeton University Press, NJ.
- Bengio, Y. (2012). Practical Recommendations for Gradient-Based Training of Deep Architectures. In Montavon, G., Orr, G. B., and Müller, K. R., editors, *Neural Networks: Tricks of the Trade*, pages 437–478. Springer.
- Bergstra, J. and Bengio, Y. (2012). Random Search for Hyper-Parameter Optimization. *Journal of Machine Learning Research*, 13:281–305.
- Cover, T. M. (1969). Learning in Pattern Recognition. In Watanabe, S., editor, *Methodologies of Pattern Recognition*, pages 111–132. New York: Academic Press.
- Geman, S., Bienenstock, E., and Doursat, R. (1992). Neural Networks and the Bias/Variance Dilemma. *Neural Computation*, 4(1):1–58.
- Glorot, X. and Bengio, Y. (2010). Understanding the Difficulty of Training Deep Feedforward Neural Networks. In *International Conference on Artificial Intelligence and Statistics*, pages 249–256.
- Highleyman, W. H. (1962). The Design and Analysis of Pattern Recognition Experiments. *Bell Systems Technical Journal*, 41:723–744.
- Ioffe, S. and Szegedy, C. (2015). Batch Normalization: Accelerating Deep Network Training by Reducing Internal Covariate Shift. In *International Conference on Machine Learning*, pages 448–456.
- Krizhevsky, A., Sutskever, I., and Hinton, G. E. (2012). ImageNet Classification with Deep Convolutional Neural Networks. In *Advances in Neural Information Processing Systems*, volume 24, pages 1106–1114.
- Mockus, J. (1974). On Bayesian Methods for Seeking the Extremum. In *Proceedings of the IFIP Technical Conference*, pages 400–404.
- Polyak, B. T. (1964). Some Methods of Speeding up the Convergence of Iteration Methods. *USSR Computational Mathematics and Mathematical Physics*, 4(5):1–17.

Bibliography

- Rasmussen, C. E. and Williams, C. K. I. (2006). *Gaussian Processes for Machine Learning*. MIT Press.
- Ripley, B. D. (1995). Statistical Ideas for Selecting Network Architectures. In Kappen, B. and Gielen, S., editors, *Neural Networks: Artificial Intelligence and Industrial Applications*, pages 183–190. Springer-Verlag, London.
- Ripley, B. D. (1996). *Pattern Recognition and Neural Networks*. Cambridge University Press.
- Snoek, J., Larochelle, H., and Adams, R. P. (2012). Practical Bayesian Optimization of Machine Learning Algorithms. In *Advances in Neural Information Processing Systems*, volume 24, pages 2951–2959.
- Srivastava, N., Hinton, G. E., Krizhevsky, A., Sutskever, I., and Salakhutdinov, R. (2014). Dropout: A Simple Way to Prevent Neural Networks from Overfitting. *Journal of Machine Learning Research*, 15:1929–1958.
- Stone, M. (1974). Cross-validatory Choice and Assessment of Statistical Predictions. *Journal of the Royal Statistical Society*, B 36(1):111–147.