Laurea Magistrale in Informatica
Dipartimento di Informatica

UNIVERSITÀ
DI TORINO

Introduction

Course

Neural Networks and Deep Learning

Professors

Roberto Esposito (roberto.esposito@unito.it)
Valentina Gliozzi (valentina.gliozzi@unito.it)

Image dreamed by stable diffusion
Prompt: "Image for the introduction slide to a course on Neural Networks"

Course Overview

- Prof. Esposito
- Prof. Gliozzi
- Both
- Lab session

12h — Introductory Material

- Introduction to the course
- Mathematical background
- Numpy and PyTorch

12h — Neural Networks Basics

- The Perceptron
- Multilayer Perceptron
- Gradient Descent
- Backpropagation

36h — Architectures

- Hopfield, RBM, and Deep Belief Nets
- CNN
- RNN / LSTM / GRU
- Autoencoder
- GANs
- Transformer
- GCN

12h — Advanced Topics

- Representation Learning
- Pruning
- Federated Learning
- Cognitive Assessment

Mathematical Background

The next few lectures, will be dedicated to the mathematical background necessary to understand the course. If you are already familiar with the following topics, you can skip these lectures:

- **19 September**: Linear Algebra (with Prof. Gliozzi);
- **20 September**: Calculus;
- **25 September**: Probability and Information Theory.

Lab sessions

For most of the course we will have lab sessions where we will use the **Python** programming language and the **PyTorch** library to implement and train neural networks.

It is preferable to work on your own laptop, but if you don't have one, you can use the computers in the lab *when available*.

Please make sure to have Python 3.7 or later installed on your machine and a virtual environment with at the following packages installed:

- `numpy`
- `matplotlib`
- `torch`
- `torchvision`
- `jupyterlab`
- `scikit-learn`
- `pandas`

```
python3 -m venv venv

source venv/bin/activate

pip install numpy matplotlib \
    torch torchvision \
    jupyterlab scikit-learn \
    pandas
```

Reference Material

- I. Goodfellow, Y. Bengio, A. Courville, MIT Press, 2016. Deep Learning
- C. Bishop, H. Bishop, Springer, Deep Learning - Foundations and Concepts

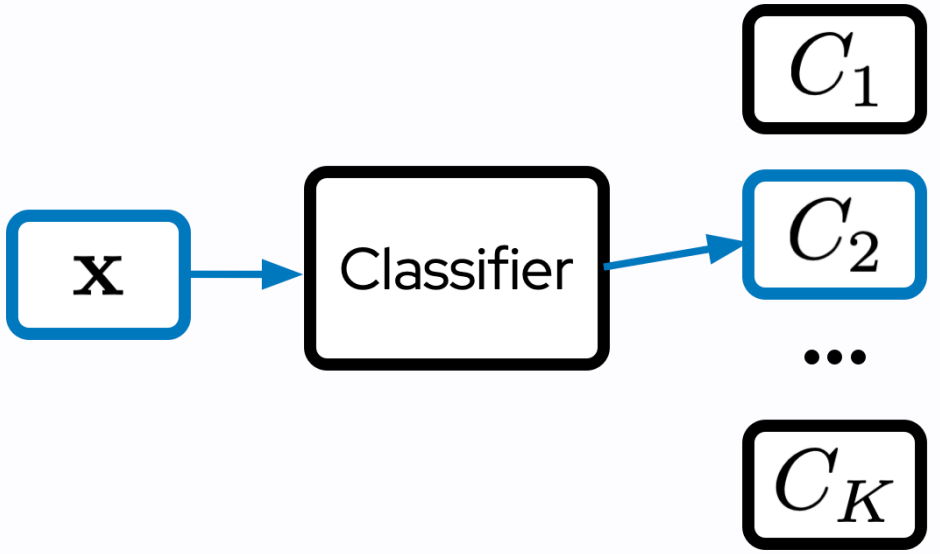
Introduction to supervised learning and induction

Tasks

Tasks in Machine Learning specify the nature of the problem to be solved. There are many different tasks studied in Machine Learning: classification, regression, clustering, association rules extraction, ...

The Classification task

Classification is a **supervised learning task** having the goal to build a model (a.k.a., a classifier) able to classify instances of an underlying concept using labels taken from a set of predefined classes.



In **supervised learning**, the algorithm learns from labeled data, i.e., data that is already labeled with the correct answer.

More formally: given a training set $\mathcal{X} = \{(\mathbf{x}_i, y_i)\}_{i \in [N]}$, the algorithm learns a function $g(\mathbf{x}; \boldsymbol{\theta})$ that approximates the "true" function $f(\mathbf{x})$ that generated the data.

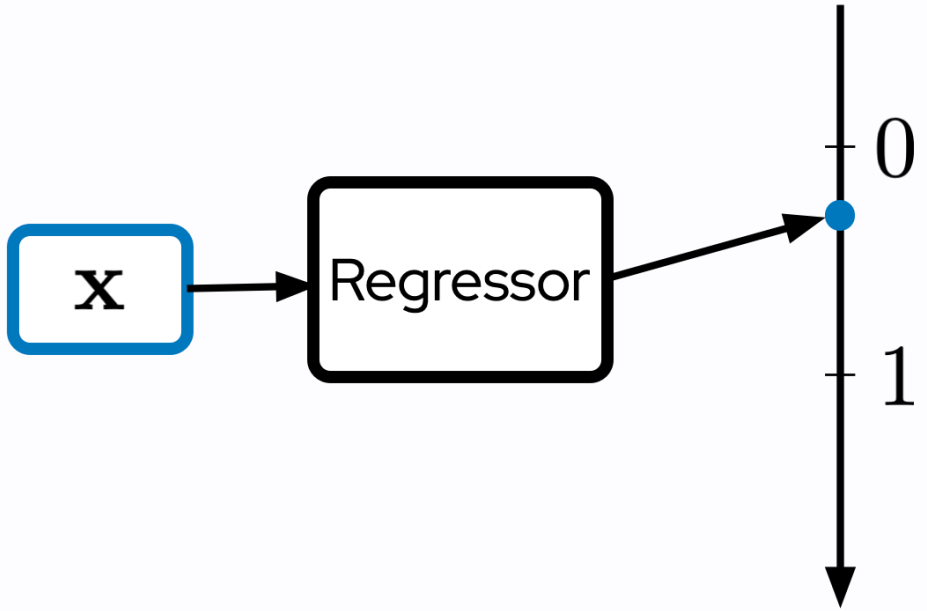
Importantly, the approximating function g is assumed to generalize to unseen data, i.e., to be able to predict the correct label for examples not seen during training.

Examples

- **Spam Detection**: given an email, predict if it is spam or not.
- **Sentiment Analysis**: given a text, predict the sentiment of the author.
- **Medical Diagnosis**: given a set of symptoms, predict the disease.
- **Face Recognition**: given an image, predict the identity of the person.
- **Object Detection**: given an image, predict the location of objects.
- **Large Language Models**: given a text, predict the next word.
- ...

The Regression task

This specification applies also to the "**regression**" task. The difference being that in regression $y \in \mathbb{R}$ while in classification y is a categorical value.



Examples

- **House Price Prediction**: given a set of features, predict the price of a house.
- **Stock Price Prediction**: given a set of features, predict the price of a stock.
- **Credit Scoring**: given a set of features, predict the credit score of a customer.
- **Demand Forecasting**: given a set of features, predict the demand of a product.
- **Weather Forecasting**: given a set of features, predict the weather.
- **Tool for classification**: given a set of features, predict the probability of a class and then classify the example based on the probability.
- **Image generation**: given a set of features, predict the pixel values of an image.
- ...

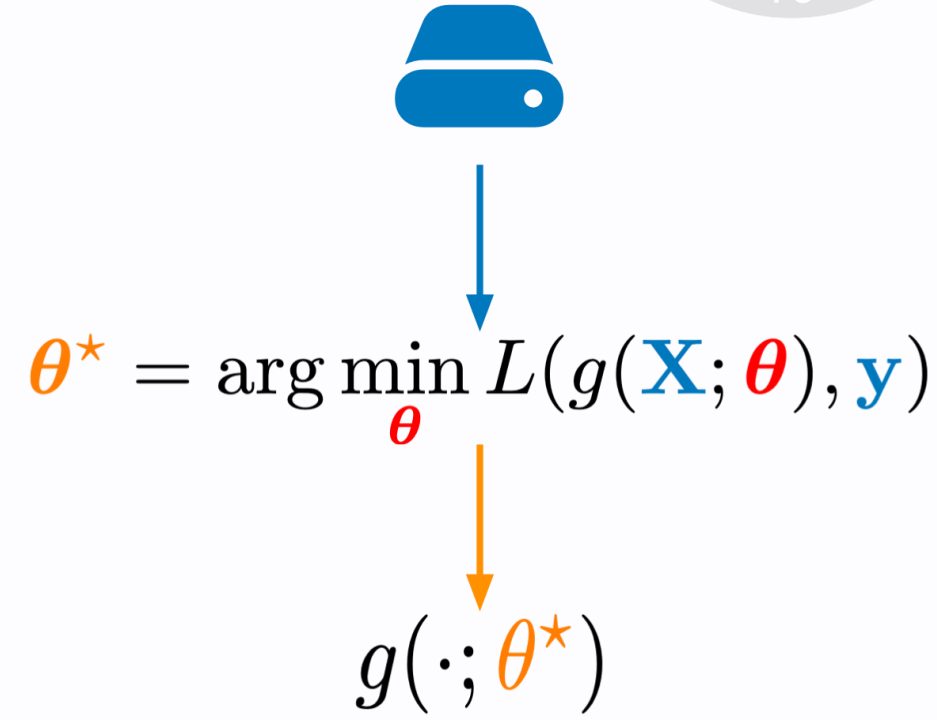
Supervised Learning: an high level overview

One of the most common approaches in machine learning is to assume a model defined up to a set of parameters:

$$\hat{y} = g(\mathbf{x}; \boldsymbol{\theta})$$

where $g(\cdot)$ is the model and $\boldsymbol{\theta}$ are its parameters.

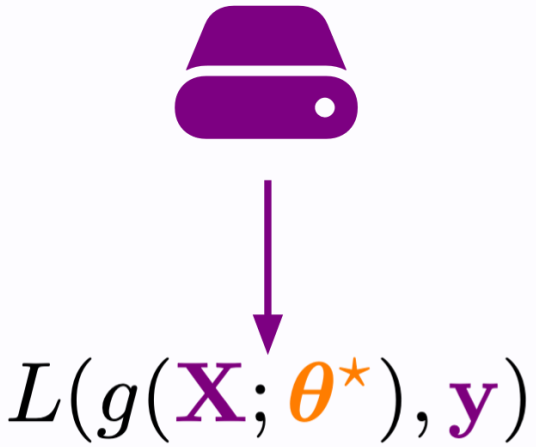
Machine learning algorithms optimizes the parameters, $\boldsymbol{\theta}$, in such a way a loss L (a function of the error) is minimized.



High Level Overview

This approach is only sensible if we are confident that the described **induction** process will hold water in our applicative settings and generalizes to **unseen data**.

This poses a more general question: **what is induction and why should it work?**



The Nature of Induction

According to the Merriam-Webster dictionary, **induction** is defined as:

> ***Induction:*** *(noun) inference of a generalized conclusion from particular instances*

This is precisely what every algorithm aims to achieve when learning a classification function.

It begins with a training set of examples (i.e., particular instances) of an underlying concept and strives to derive a general rule for classifying unseen examples, which are collected in the test set.

Is induction an ill posed problem?

Let us consider a function:

$$f : \{1, 2, 3, 4\} \to \{0, 1, 2, 3, 4, 5, 6, 7, 8, 9\}$$

I tell you that:

$$f(1) = 2$$
$$f(2) = 4$$
$$f(3) = 6$$

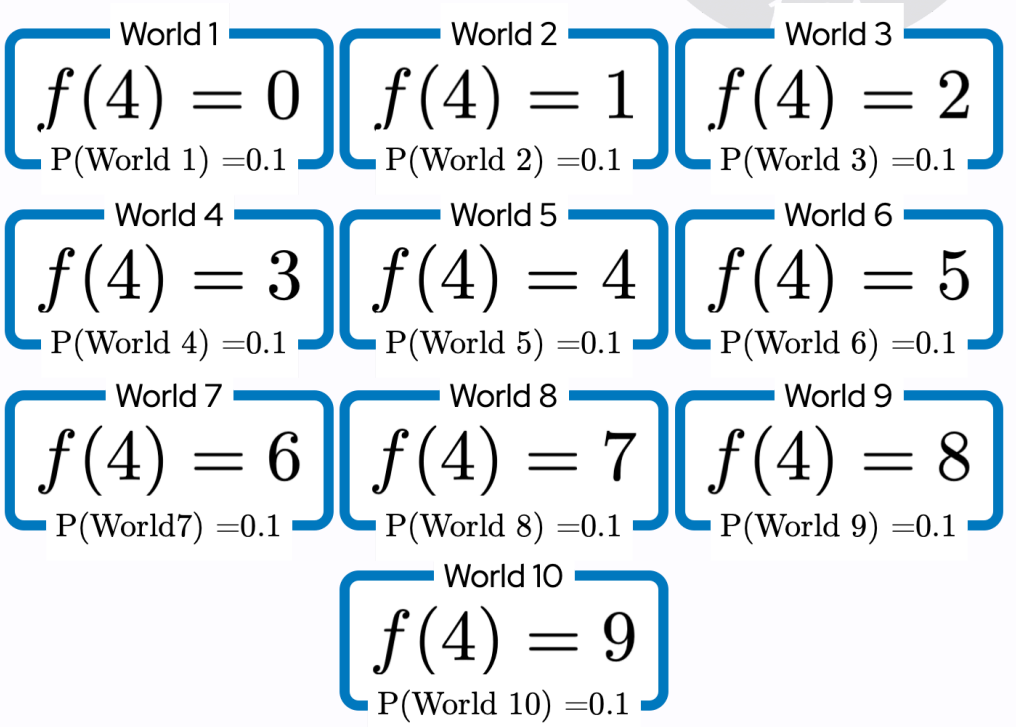
and I ask you to predict the value of $f(4)$.

What's your answer?

Is induction an ill posed problem?

In fact, **without any other assumption** about f **you cannot really answer the question**. *Any* answer is equally good if averaged over all possible "worlds".

In particular, there are 10 possible answers to that questions and *without any additional assumption*, all of them are equally likely, and any answer will be correct only in $1/10$ of these possible worlds.



No Free Lunch Theorem

This is the gist of what is know as the *No Free Lunch Theorem* for machine learning, which states that:

> *[...] all algorithms are equivalent, on average, by any of the following measures of risk [...]".*

David Wolpert from *D. Wolpert. The lack of a priori distinctions between learning algorithms. Neural Computation, 8(7), p.1341--1390.*

Or put in another way:

> *All models are wrong, but some models are useful.*

George Box, *G. E. P. Box and N. R. Draper. Empirical Model-Building and Response Surfaces. Wiley, New York, 1987, p.424.*

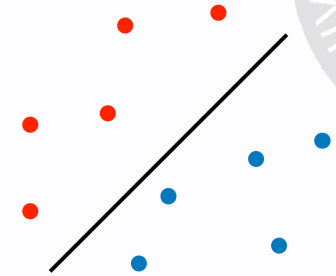
Is induction an ill posed problem?

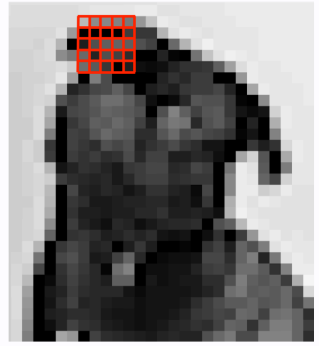
The no free lunch theorem does *not* imply that there is no hope for learning. It only says that **a learning algorithm is only as good as its inductive bias**.

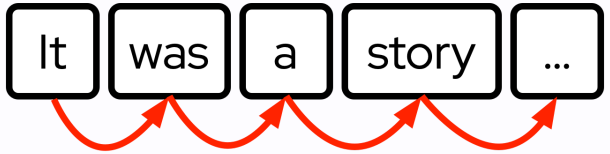
Whenever you consider which is the best learning algorithm for your use case, you should strive to consider (in addition to all other factors) how well the inductive bias of the algorithm fits the problem at hand.

Examples

- a **linear classifier** assumes that examples lie in a metric space and the underlying concept is linear in nature.
- a **convolutional neural network**, assumes that the underlying concept is translation invariant and that the input is an image: i.e., a 2D grid of pixels where adjacent pixels are correlated.
- a **recurrent neural network**, assumes that the underlying concept is sequential in nature.







How to train a neural network model

Neural networks are a incredibly flexible family of models that can be used to approximate any function. This flexibility comes at a cost: they **depend on a vast number of hyperparameters** that need to be set correctly in order to achieve good performance.

Examples:

- the number of layers
- the number of neurons in each layer
- the learning rate
- the optimizer
- ...

It is then important to understand **how to set these hyperparameters correctly**, and how to validate the quality of the inferred model **without overestimating its performance**.

Problem statement

We want to define the correct procedure to:

- find the best possible way to set the hyperparameters
- get an accurate estimation of the generalization error at the end of the process

Generalization Error

The generalization error is the error that the acquired classifier g commit, on average on examples drawn from the same distribution used for sampling the examples in the training set:

$$R = E_{(\mathbf{x},y)\sim p^\star}[L(y, g(\mathbf{x};\boldsymbol{\theta}))]$$

where $p^\star$ is the *true* distribution of the data and $L(y, g(\cdot;\boldsymbol{\theta}))$ is a *loss* function used to measure how bad the error is when y is predicted as $g(\mathbf{x};\boldsymbol{\theta})$.

Note: some authors call the generalization error the difference between R and the error committed on the training set.

Loss Functions

Typical loss functions are: the 0-1 loss

$$L(y, y') = \mathbb{I}_{y \neq y'} = \begin{cases} 1 & \text{if } y \neq y' \\ 0 & \text{otherwise} \end{cases}$$

or the quadratic loss:

$$L(y, y') = (y - y')^2$$

but many others are possible (e.g., hinge, exponential, logistic, ...).

Empirical Error

It should be evident that (in general) R cannot be computed. To overcome this problem in most cases R is approximated with the *empirical error*:

$$\hat{R}_T = \frac{1}{|T|} \sum_{(\mathbf{x},y)\in T} L(y, g(\mathbf{x}; \boldsymbol{\theta}))$$

where T is a finite sample drawn from $p^\star$.

Training and Test Errors

When T is the **training set**, i.e., $T \equiv \mathrm{Tr}$, $\hat{R}_{\mathrm{Tr}}$ is called the **training error** of g.

When T is the **test set**, i.e., $T \equiv \mathrm{Te}$, $\hat{R}_{\mathrm{Te}}$ is called the **test error** of g.

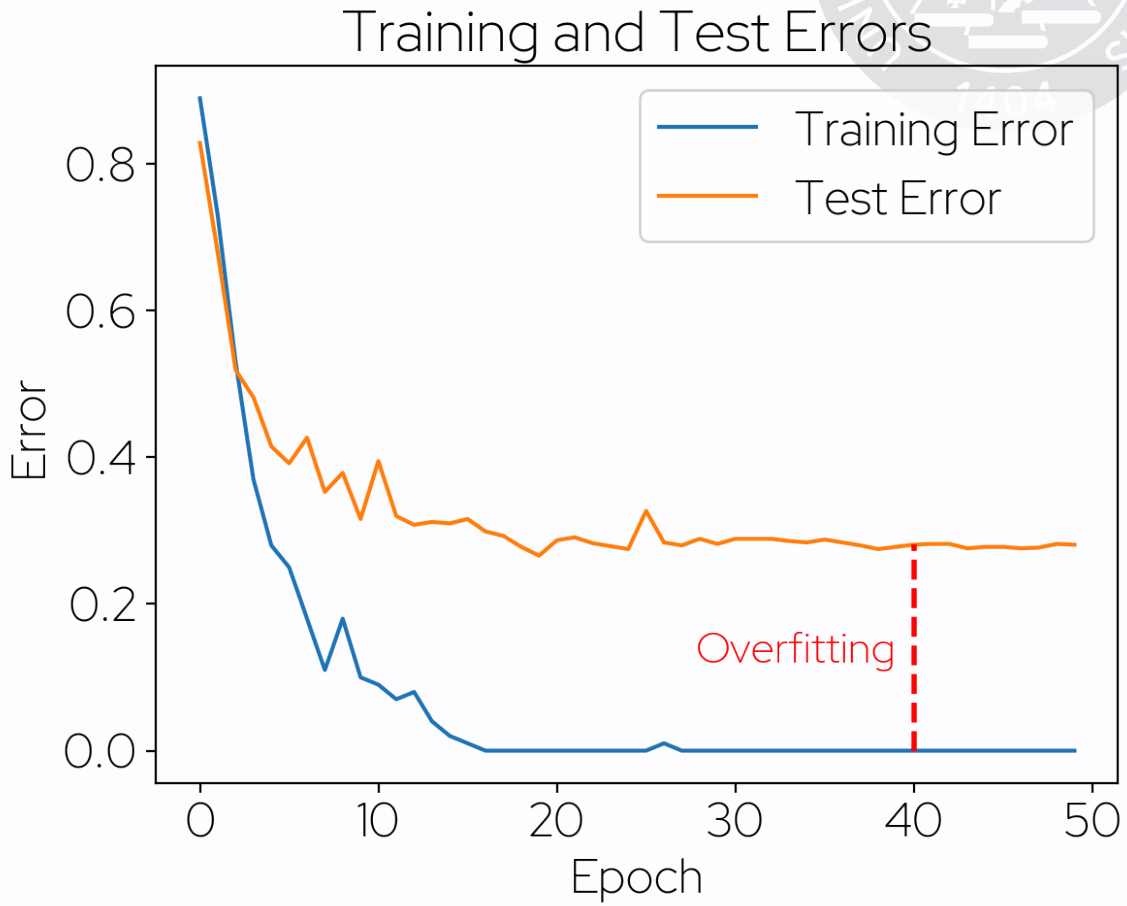
Since the training error is implicitly (sometimes explicitly) optimized by the learning algorithm, it has an **optimistic bias**, i.e.: it tends to be lower than the generalization error.

The test error serves as an unbiased estimator of the generalization error R. However, it may exhibit a **pessimistic bias** because retraining the model on the entire dataset is likely to produce a model with a lower error.

Overfitting

When $\hat{R}_{\mathrm{Te}} - \hat{R}_{\mathrm{Tr}} > 0$ the algorithm is said to be **overfitting** the training set (or simply to be overfitting).

Overfitting is a common problem for learning algorithms and it is usually necessary to counter it by some method.



Estimating Hyperparameters

Coming back to our initial question: **how do we estimate hyperparameters?**

Given what we have seen so far, we might want to implement this procedure:

```
errors = []
parameters = []

while condition not met:
    theta = choose_hyperparameters()
    g = learn(training_set, theta)

    errors.append(evaluate_error(test_set, g))
    parameters.append( parameters )


best_index = errors.index(min(errors))
return parameters[best_index]
```

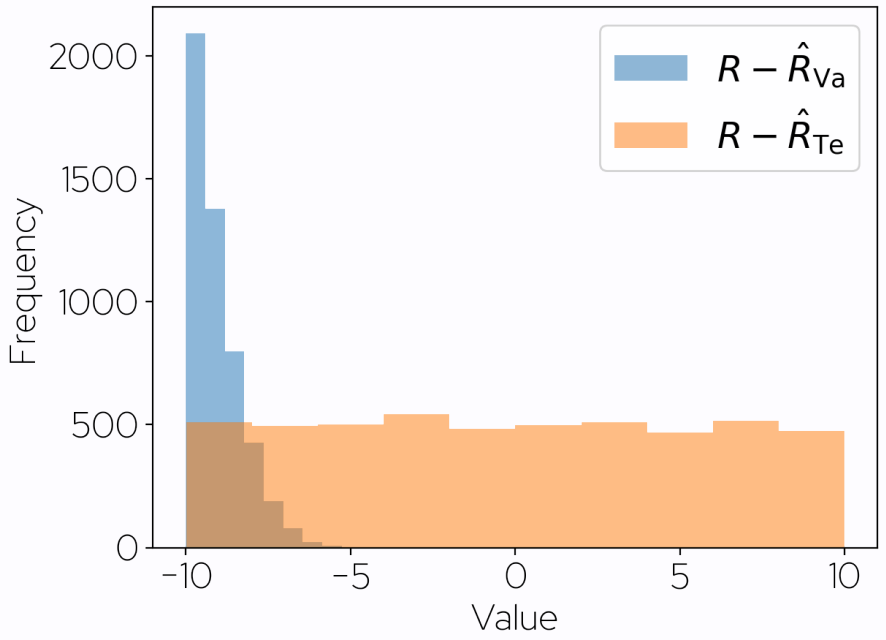
Overfitting the test set

Unfortunately **this will not work**. The problem is that the above procedure, even if carried on by hand is still *optimizing the empirical error*, this time using the test set.

This makes the test set error an **optimistic estimator** of the generalization error and we say that we are **overfitting the test set**.

An illustrative experiment

```python
# 1000 hyper-parameters to set, each yielding an error in (0,100)
means = np.random.uniform(0, 100, 1000)

valid_error_dist = []
test_error_dist = []

N = 5000
for i in range(N):
    # Each iteration simulates one hyperparameter selection
    #  experiment (say by a grid search)  selecting between the
    # 1000 hyper-parameters

    # Evaluate the errors on the validation and test set
    # for each one of the 1000 parametrizations
    samples = np.array(
        [
            [m,                                  # R
             m + np.random.uniform(-10, 10),  # R_val
             m + np.random.uniform(-10, 10)]  # R_test
            for m in means
        ]
    )

    # selects the best hp according to R_val
    amin = np.argmin(samples[:, 1])

    valid_error_dist.append(samples[amin, 1] - samples[amin, 0])
    test_error_dist.append(samples[amin, 2] - samples[amin, 0])
```



The Validation Set

To overcome this problem, we need to make use of an additional set of examples: the **validation set**.

The validation set is used **solely to evaluate the performance of the model under a given choice of hyperparameters**. Once the hyperparameters are set, the **test set is used to assess the final model's quality**.

Note: when data is scarce, the choices have been made, and the quality of the result is satisfactory, one can retrain the classifier using all the data available (i.e., the training and the validation set are merged). Analogously, **when the test set is no longer needed**, it is can be merged with the training set to further improve the classifier.

Estimating hyperparameters

The correct way of estimating the hyperparameters is then:

```
errors = []
parameters = []

while condition not met:
    theta = choose_hyperparameters()
    g = learn(training_set, theta)

    errors.append(evaluate_error(validation_set, c))
    parameters.append( parameters )


best_index = errors.index(min(errors))
return parameters[best_index]
```

Estimating hyperparameters

To summarise:

- **training set**: used during learning;
- **validation set**: used to assess the quality of the current choice of hyperparameters;
- **test set**: used to assess the quality of the final classifier.