Deep Learning in Omics

Sessión 1: Fundamentals of Machine Learning

FRC-EVL

Curso Extensión Universitaria - UB, 2021

Table of Contents

Program

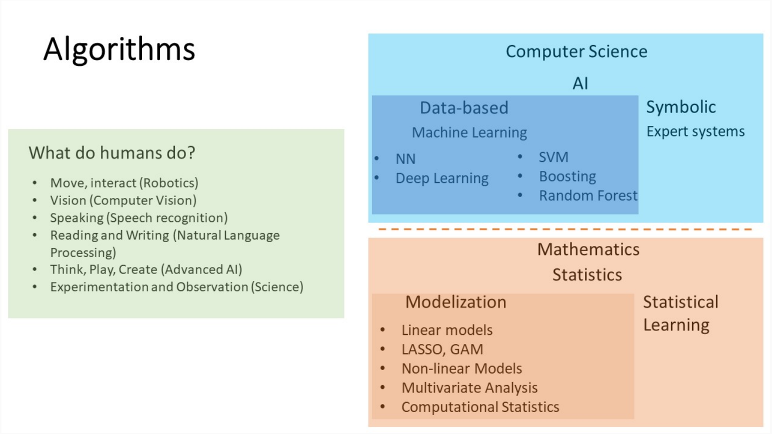
Program

- 01/02 ML+install Keras+Example+ML+NN
- 02/02 Optimization + CNN + Practices1
- 03/02 CNN + Monitoring + Practices2
- 04/02 Autoencoders + Practices3
- 05/02 Deep Learning data integration

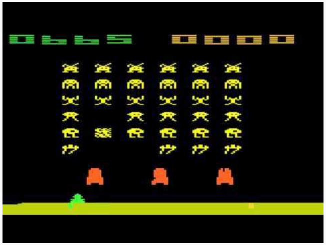
1. Introduction

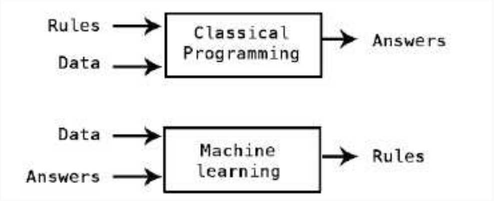
Machine learning: a new programming paradigm

A machine-learning system is trained rather than explicitly programmed. It's presented with many examples relevant to a task, and it finds **statistical structure** in these examples that eventually allows the system to come up with rules for automating the task.



Learning rules from data





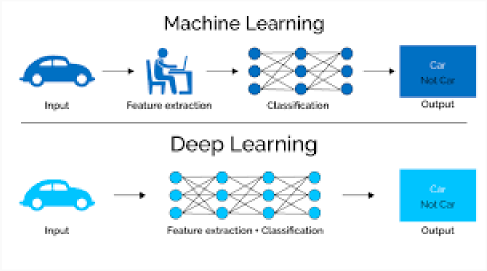
What makes deep learning different: feature learning

Hand-made features

- Neural networks, Support Vector Machines, Random Forest and Gradient Boosting.

Back to Neural networks

- Better performance on many problems.
- Completely automates the most crucial step in a machine-learning workflow: feature engineering.

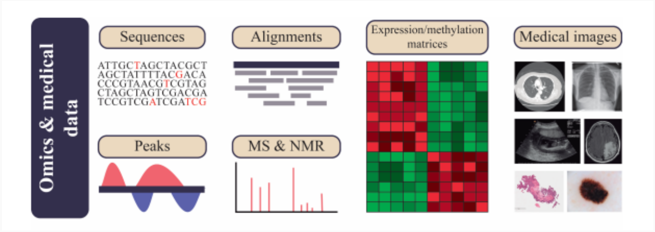


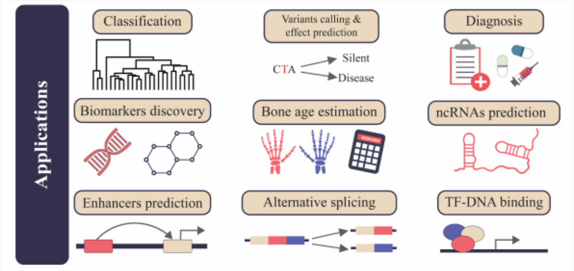
What deep learning has achieved so far

Deep learning has achieved the following breakthroughs, all in historically difficult areas of machine learning:

- Near-human-level image classification
- Near-human-level speech recognition
- Near-human-level handwriting transcription
- Improved machine translation
- Improved text-to-speech conversion
- Digital assistants such as Google Now and Amazon Alexa
- Near-human-level autonomous driving
- Improved ad targeting, as used by Google, Baidu, and Bing
- Improved search results on the Web
- Ability to answer natural-language questions
- Superhuman Go playing

Deep Learning in Omics





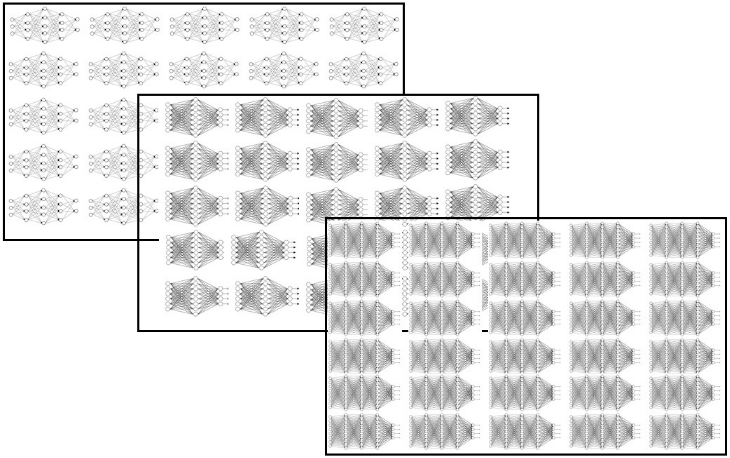
Martorell-Marugan, J. et all.(2019). Deep Learning in Omics Data Analysis and Precision Medicine. 10.15586/computationalbiology.2019.ch3.

2. Statistical Learning

Statistical learning

- Input space and the Output space: $\mathcal{X}$ and $\mathcal{Y}$.
- Input space: $\mathcal{X} = \mathbb{R}^p$, tensor arrays $\mathcal{X} = \mathbb{R}^p \times \mathbb{R}^p \times 3$.
- Output space: $\mathcal{Y} = \mathbb{R}$ (regression); $\mathcal{Y} = \{-1, 1\}$ (classification).
- $P_{X,Y}$ joint probability distribution on $\mathcal{X} \times \mathcal{Y}$ (**Unknown**).
- Learning set: $\mathcal{D}_n = \{(x_1, y_1), ..., (x_n, y_n)\}$ be an i.i.d. random sample from $P_{X,Y}$.
- **Goal**: Find a mapping $f : \mathcal{X} \to \mathcal{Y}$ based on $\mathcal{D}_n$ so that $f(X)$ is a good approximation of Y.
- A learning algorithm is a procedure $\mathcal{A}$ which takes the training set $\mathcal{D}_n$ and produces a model $f_n = \mathcal{A}(\mathcal{D}_n)$ as the output.
- Learning algorithm will search over a space of functions $\mathcal{F}$, which we call the **Hypothesis space**.

Hypothesis space



Loss function

- A *loss function* is a mapping $\ell : \mathcal{Y} \times \mathcal{Y} \to \mathbb{R}^+$
- For example, in binary classification the 0-1 loss function $\ell(y,z) = \mathbf{1}[y \neq z]$ and in regression the squared error loss $\ell(y,z) = (y-z)^2$.
- The performance of a predictor $f : \mathcal{X} \to \mathcal{Y}$ is measured by the expected loss, that is, the risk or generalization error:

$$R(f) = E_{X,Y}\big(\ell(Y, f(X))\big)$$

  where the expectation is taken with respect to the distribution $P_{X,Y}$.
- Can we calculate $R(f)$? Given a loss function $\ell(\cdot,\cdot)$, the risk $R(f)$ is not computable as $P_{X,Y}$ is unknown.
- Thus we may not able to directly minimize $R(f)$ to obtain some predictor.

Empirical Risk Minimization

- We are provided with the training data $\mathcal{D}_n = \{(x_1, y_1), ..., (x_n, y_n)\}$ which represents the underlying distribution $P_{X,Y}$.
- Instead of minimizing $R(f) = E_{X,Y}[\ell(Y, f(X))]$, one may replace $P_{X,Y}$ by its empirical distribution and define the *empirical risk* $\hat{R}_n(f)$ as

$$\hat{R}_n(f) = \frac{1}{n}\sum_{i=1}^{n} \ell(y_i, f(x_i))$$

- Thus obtain the following minimization problem:

$$\hat{f}_n = \underset{f \in \mathcal{F}}{\operatorname{argmin}} \frac{1}{n}\sum_{i=1}^{n} \ell(y_i, f(x_i))$$

which we call *empirical risk minimization* (ERM).

Overfitting

- ERM works by minimizing the empirical risk $\hat{R}_n(f)$, while the goal of learning is to obtain a predictor with a small true risk $R(f)$.
- Although under certain conditions $\hat{R}_n(f)$ will converge to $R(f)$ as $n \to \infty$, in practice we always have a finite sample and as a result, there might be a large discrepancy between those two targets, especially when $\mathcal{F}$ is large and n is small.
- Overfitting refers to the situation where we have a small empirical risk but still a relatively large true risk.

3. Overfitting

Tackling Overfitting

- The fundamental issue in machine learning is the tension between **Optimization** and **Generalization**.
- Optimization refers to the process of adjusting a model to get the best performance possible on the training data (the learning in machine learning), whereas
- Generalization refers to how well the trained model performs on data it has never seen before.

Statistical Learning Theory

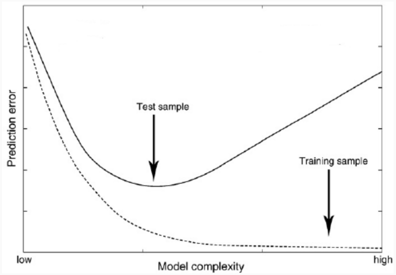
- Statistical Learning Theory provides probabilistic bounds on the distance between the empirical and true risk of any function that can be used to control overfitting.
- The bounds involve the number of examples and the capacity h of the hypothesis space, a quantity measuring the "complexity" of that space.
- Appropriate capacity quantities are defined in the theory, the most popular one being the VC-dimension (Vapnik and Chervonenkis, 1971).

Statistical Learning Theory

The bounds have the following general form: with probability at least η

$$R(f) < \hat{R}_n(f) + \varphi\left(\sqrt{\frac{h}{n}}, \eta\right)$$

where h is the capacity and φ an increasing function of h/n and η.



If the capacity of the function space (in which we perform empirical risk minimization) is very large and the number of examples is small, then the distance between the empirical and expected risk can be large and overfitting is very likely to occur.

Prevent overfitting in neural networks

- Reduce the capacity of the network.
- Add weight regularization. The principle of regularization consists in choosing the model that minimizes the penalized risk:
  $$\frac{1}{n}\sum_{i=1}^{n}\ell(y_i, f(x_i)) + \lambda\Omega(f)$$
  $\Omega(f)$ such that it is small for functions which vary slowly, and large for functions which fluctuate a lot. For instance, $\Omega(f) = ||f||^2$. The λ is a trade-off constant. It "negotiates" between the importance of $\hat{R}_n(f)$ and of $\Omega(f)$.
- Add dropout.
- Get more training data.

4. Evaluating machine learning models

Training, validation and test sets

- In machine learning, the goal is to achieve models that generalize.
- It's crucial to be able to reliably measure the generalization power of your model.
- Evaluating a model always boils down to splitting the available data into three sets: training, validation, and test.
- When little data is available there are alternative recipes. K-fold validation on training set.
- You train on the training data and evaluate your model on the validation data.
- Once your model is ready for prime time, you test it one final time on the test data.

Hyperparameters

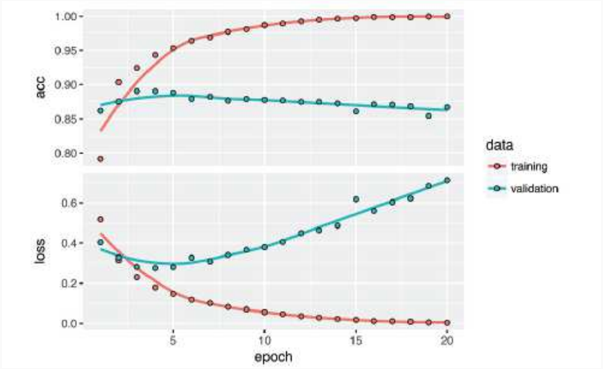
- You may ask, why not have two sets: a training set and a test set? Much simpler!
- The reason is that developing a model always involves tuning its configuration: for example, choosing the number of layers or the size of the layers (called the **hyperparameters** of the model, to distinguish them from the parameters, which are the network's weights).
- We have to minimize the empirical risk for a fixed value of the hyperparameters (hp)

$$\underset{\theta}{\operatorname{argmin}} \frac{1}{n} \sum_{i=1}^{n} \ell\big(y_i, f_{\mathrm{hp}}(x_i;\theta)\big)$$

- Hyperparameter selection. Find the values of the hyperparameters that determine models with lower empirical risk.

Tuning

- You'll do this **tuning** by using as a feedback signal the performance of the model on the validation data (here, the hyperparameter is the number of epochs).
- A model that performs better on the training data isn't necessarily a model that will do better on data it has never seen before. What you are seeing is overfitting.

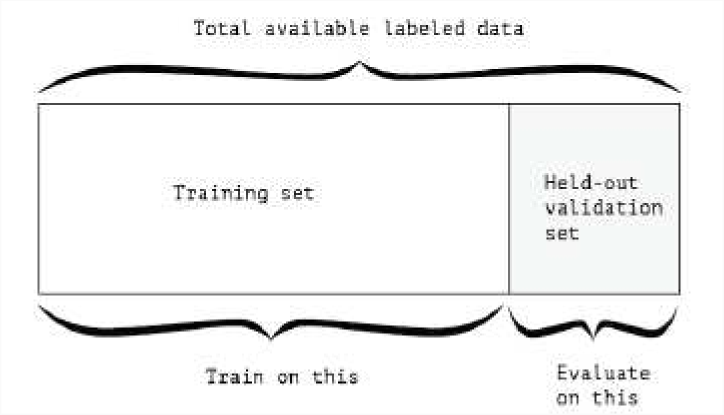


Validation recipes

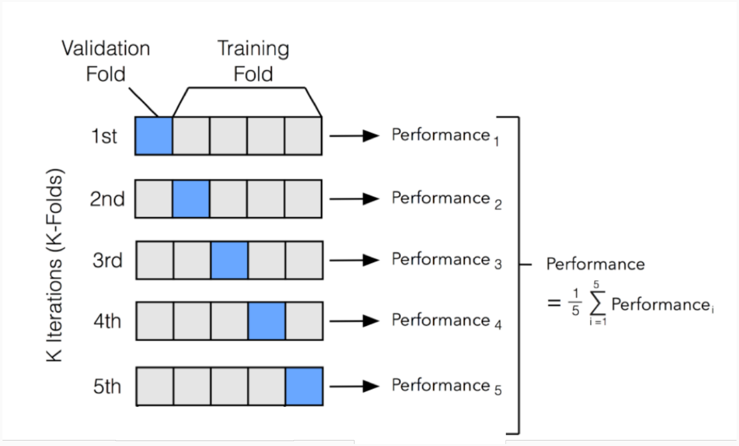
Splitting your data into training, validation, and test sets may seem straightforward, but there are a few advanced ways to do it that can come in handy when little data is available. Let's review three classic evaluation recipes:

- simple hold-out validation,
- K-fold validation,
- iterated K-fold validation with shuffling.

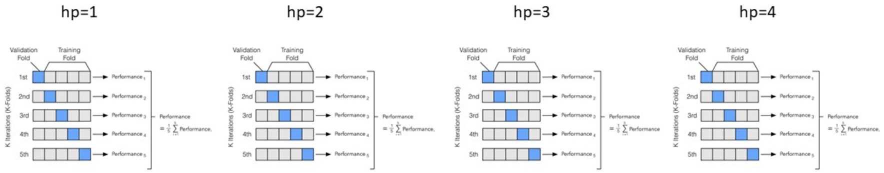
Simple hold-out

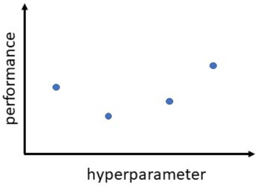


K-folds


Validation Fold
Training Fold
K Iterations (K-Folds)
1st
2nd
3rd
4th
5th
Performance $_1$
Performance $_2$
Performance $_3$
Performance $_4$
Performance $_5$
Performance $= \frac{1}{5}\sum_{i=1}^{5}$ Performance$_i$

Cross-Validation and Hyperparameter Tuning





Iterated K-folds

- This one is for situations in which you have relatively little data available and you need to evaluate your model as precisely as possible.
- It consists of applying K-fold validation multiple times, shuffling the data every time before splitting it K ways.
- The final score is the average of the scores obtained at each run of K-fold validation.
- Note that you end up training and evaluating $P \times K$ models (where P is the number of iterations you use), which can very expensive.

6. The workflow of machine learning

General blueprint

1. Defining the problem and assembling a dataset.
2. Choosing a measure of success.
3. Deciding on an evaluation protocol.
4. Preparing your data.
5. Developing a model that does better than a baseline.
6. Scaling up: developing a model that overfits.
7. Regularizing your model and tuning your hyperparameters.

Define the problem

- What will your input data be? What are you trying to predict?
  You can only learn to predict something if you have available training data. Identifying the problem type will guide your choice of model architecture, loss function, and so on.
- What type of problem are you facing?
  Is it binary classification? Multiclass classification? Scalar regression? Vector regression? Multiclass, multilabel classification? Something else, like clustering, generation, or reinforcement learning?
- Keep it in mind that machine learning can only be used to memorize patterns that are present in your training data.
- One class of unsolvable problems you should be aware of is nonstationary problems.
- You can only recognize what you've seen before.
  Using machine learning trained on past data to predict the future is making the assumption that the future will behave like the past. That often isn't the case.

Choosing a measure of success

To control something, you need to be able to observe it. To achieve success, you must define what you mean by success: accuracy? precision-recall? customer retention rate?

Your metric for success will guide the choice of a loss function: that is, what your model will optimize.

- Accuracy,
- TPR and FPR,
- Area under the receiver operating characteristic curve,
- Precision-recall,
- Mean average precision.

Deciding on an evaluation protocol

You must establish how you'll measure your current progress. We've previously reviewed three common evaluation protocols:

- Maintaining a hold-out validation set. The way to go when you have plenty of data
- Doing K-fold cross-validation. The right choice when you have too few samples for hold-out validation to be reliable
- Doing iterated K-fold validation. For performing highly accurate model evaluation when little data is available.

Just pick one of these. In most cases, the first will work well enough.

Preparing your data

You should format your data in a way that can be fed into a machine-learning model.

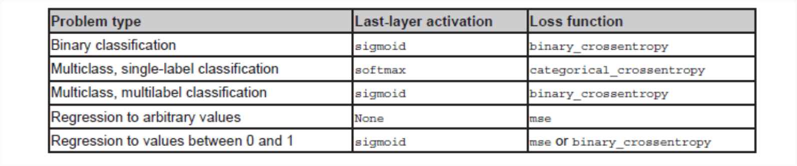
- If different features take values in different ranges (heterogenous data), then the data should be normalized.
- Data should be formatted as tensors.
- The values taken by these tensors should usually be scaled to small values: for example, in the [-1,1] range or [0, 1] range.
- Feature engineering, especially for small data problems.

Working model

Assuming that things go well, you need to make three key choices to build your first working model:

- Last-layer activation. This establishes useful constraints on the network's output.
- Loss function. This should match the type of problem you're trying to solve. For instance, binary crossentropy, mean square error, and so on.
- Optimization configuration. What optimizer will you use? What will its learning rate be?

Last-layer activation and loss function



Problem type	Last-layer activation	Loss function
Binary classification	`sigmoid`	`binary_crossentropy`
Multiclass, single-label classification	`softmax`	`categorical_crossentropy`
Multiclass, multilabel classification	`sigmoid`	`binary_crossentropy`
Regression to arbitrary values	`None`	`mse`
Regression to values between 0 and 1	`sigmoid`	`mse` or `binary_crossentropy`

Last-layer activation

- sigmoid: $f(x;\theta) = \frac{1}{1+e^{out(x;\theta)}}$
- softmax: $f(x;\theta)_i = \frac{e^{out_i(x;\theta)}}{\sum_j^C e^{out_j(x;\theta)}}$
- linear: $f(x;\theta) = out(x;\theta)$

Loss function

- binary cross-entropy: $\ell(y, f(x;\theta)) = -y\log(f(x;\theta)) - (1-y)\log(1-f(x;\theta))$
- categorical cross-entropy: $\ell(y, f(x;\theta)) = -\sum_{i=1}^C y_i \log(f(x;\theta)_i)$
- mse: $\ell(y, f(x;\theta)) = \big(y - f(x;\theta)\big)^2$

Scaling up: developing a model that overfits

- To figure out how big a model you'll need, you must develop a model that overfits. This is fairly easy: Add layers. Make the layers bigger. Train for more epochs.
- Always monitor the training loss and validation loss, as well as the training and validation values for any metrics you care about. When you see that the model's performance on the validation data begins to degrade, you've achieved overfitting.
- The next stage is to start regularizing and tuning the model, to get as close as possible to the ideal model that neither underfits nor overfits.

Regularizing your model and tuning your hyperparameters

This step will take the most time: you'll repeatedly modify your model, train it, evaluate on your validation data (not the test data, at this point), modify it again, and repeat, until the model is as good as it can get.

These are some things you should try:

- Add dropout.
- Try different architectures: add or remove layers.
- Add L1 and/or L2 regularization.
- Try different hyperparameters (such as the number of units per layer or the learning rate of the optimizer) to find the optimal configuration.
- Optionally, iterate on feature engineering: add new features, or remove features that don't seem to be informative.

Regularizing your model and tuning your hyperparameters

- Once you've developed a good enough model configuration, you can train your final production model on all the available data (training and validation) and evaluate it one last time on the test set.
- If it turns out that performance on the test set is significantly worse than the performance measured on the validation data, this may mean either that your validation procedure wasn't reliable after all, or that you started overfitting to the validation data while tuning the parameters of the model.
- In this case, you may want to switch to a more reliable evaluation protocol (such as iterated K-fold validation).