MITx 6.86x - Machine Learning with Python: from Linear Models to Deep Learning

https://www.edx.org/course/machine-learning-with-python-from-linear-models-to

Lecturers: Regina Barzilay, Tommi Jaakkola, Karene Chu

Student's notes (2020 run)

Disclaimer: The following notes are a mesh of my own notes, selected transcripts, some useful forum threads and various course material. I do not claim any authorship of these notes, but at the same time any error could well be arising from my own interpretation of the material.

Contributions are really welcome. If you spot an error , want to specify something in a better way (English is not my primary language), add material or just have comments, you can clone, make your edits and make a pull request (preferred) or just open an issue.

(This PDF versions may be outdated ! Please refer to the GitHub repository source files for latest version.)

Table of contents

[MITx 6.86x Notes Index]

Unit 00 - Course Overview, Homework 0, Project 0

Recitation 1 - Brief Review of Vectors, Planes, and Optimization

Points and Vectors

Norm of a vector

Norm: Answer the question how big is a vector

- $\|X\|_l \equiv l - NORM := (\sum_{i=1}^{size(X)} x_i^l)^{(1/l)}$
- Julia: `norm(x)`
- NumPy: `numpy.linalg.norm(x)`

If l is not specified, it is assumed to be the 2-norm, i.e. the Euclidian distance. It is also known as “length”.

Dot product of vector

Aka “scalar product” or “inner product”.

It has a relationship on how vectors are arranged relative to each other

- Algebraic definition: $x \cdot y \equiv x'y := \sum_{i=1}^{n} x_i * y_i$
- Geometric definition: $x \cdot y := \|x\| * \|y\| * cos(\theta)$ (where θ is the angle between the two vectors)
- Julia: `dot(x,y)`
- Numpy: `np.dot(x,y)`

Note that using the two definitions and the `arccos`, the inverse function for the cosene, you can retrieve the angle between two functions as `angle_x_y = acos(dot(x,y)/(norm(x)*norm(y)))`.

Geometric interpretation of a vector

Geometrically, the elements of a vector can be seen as the coordinates of the position of arrival *compared to the position of departing*. They represent hence the *shift* from the departing point.

For example the vector [-2,0] could refer to the vector from the point (4,2) to the point (2,2) but could also represent the vector going from (6,4) to (4,4).

Vectors whose starting point is the origin are called “position vectors” and they define the coordinates in the n-space of the points where they arrive to.

Vector projections

Let’s be *a* and *b* two (not necessary unit) vectors. We want to compute the vector *c* being the projection of *a* on *b* and its l-2 norm (or length).

Let’s start from the length. We know from a well-known trigonometric equation that

$\|c\| = \|a\| * cos(\alpha)$, where α is the angle between the two vectors *a* and *b*.

But we also know that the dot product $a \cdot b$ is equal to $\|a\| * \|b\| * cos(\alpha)$.

By substitution we find that $\|c\| = \frac{a \cdot b}{\|b\|}$. This quantity is also called the

component of a in the direction of b.

To find the vector c we now simply multiply $\|c\|$ by the unit vector in the direction of b, $\frac{b}{\|b\|}$, obtaining $c = \frac{a \cdot b}{\|b\|^2} * b$.

If b is already a unit vector, the above equations reduce to:

$\|c\| = a \cdot b$ and $c = (a \cdot b) * b$

In Julia:

```
using LinearAlgebra
a = [4,1]
b = [2,3]
normC = dot(a,b)/norm(b)
c = (dot(a,b)/norm(b)^2) * b
```

In Python:

```
import numpy as np
a = np.array([4,1])
b = np.array([2,3])
normC = np.dot(a,b)/np.linalg.norm(b)
c = (np.dot(a,b)/np.linalg.norm(b)**2) * b
```

Planes

An (hyper)plane in n dimensions is any n−1 dimensional subspace defined by a linear relation. For example, in 3 dimensions, hyperplanes span 2 dimensions (and they are just called “planes”) and can be defined by the vector formed by the coefficients {A,B,C,D} in the equation $Ax + By + Cz + D = 0$. Note that the vector is not unique: in relation to the A-D coefficients, the equation is homogeneous, i.e. if we multiply all the A-D coefficients by the same number, the equation remains valid.

As hyperplanes separate the space into two sides, we can use (hyper)planes to set boundaries in classification problems, i.e. to discriminate all points on one side of the plane vs all the point on the other side.

- *Normal* of a plane: any vector perpendicular to the plane.
- *Offset of the plane with the origin*: the distance of the plan with the origin, that is the specific normal between the origin and the plane

A plane can now be uniquely identified in an other way, starting from a point x_p on the plane and a vector $\vec{v}$ normal to the plane (not necessarily departing from the point or even from the plane).

Given a generic point x, we call $\vec{x}$ its position vector and we call $\overrightarrow{x_p}$ the position vector of the point x_p sitting on the plane. The point x is part of the plane iff the vector connecting the two points, that is $x \overrightarrow{-} x_p$, lies on the plane. In turn this is true iff such vector is orthogonal to the normal of the plane $\vec{v}$, where we can

check this using the dot product.

To sum up, we can define the plane as the set of all points x x such that $(x \vec{-} x_p) \cdot \vec{v} = 0$.

As from the coefficients A-D in the equation, while x_p and $\vec{v}$ unambiguously identify the plane, the converse is not true: any plane has indeed infinite points and normal vectors.

For example, let's define a plane in two dimensions passing by the point $x_p = (3, 1)$ and with norm $\vec{v} = (2, 2)$, and let's check if point $a = (1, 3)$ is on the plane. Using the above equation we find $(\vec{a} - \vec{x_p}) \cdot \vec{v} = \left(\begin{bmatrix} 1 \\ 3 \end{bmatrix} - \begin{bmatrix} 3 \\ 1 \end{bmatrix} \right) \cdot \begin{bmatrix} 2 \\ 2 \end{bmatrix} = 0$, a is part of the plane.

Let's consider instead the point b = (1,4). As $(\vec{b} - \vec{x_p}) \cdot \vec{v} = \left(\begin{bmatrix} 1 \\ 4 \end{bmatrix} - \begin{bmatrix} 3 \\ 1 \end{bmatrix} \right) \cdot \begin{bmatrix} 2 \\ 2 \end{bmatrix} = 2$, b is not part of the plane.

Starting from the information on the point and the normal we can retrieve the algebraic equation of the plane rewriting the equation $(x \vec{-} x_p) \cdot \vec{v} = 0$ as $\vec{x} \cdot \vec{v} - \vec{x_p} \cdot \vec{v} = 0$: the first dot product gives us the polynomial terms in the n-dimensions (often named θ), the last (negatively signed) term gives the associated offset (often named θ_0). Seen from the other side, this also means that the coefficients of the dimensions of the algebraic equation represent the normal of the plane. We can hence define our plane with just the normal and the corresponding offset (always a scalar).

Note that when θ is a unit vector (a vector whose 2-norm is equal to 1) the offset θ_0 is equal to the offset of the plane with the origin.

Using the above example, we find that the algebraic equation of the plane is $2x_1 + 2x_2 - (6 + 2) = 0$, or, equivalently, $\frac{x_1}{\sqrt{2}} + \frac{x_2}{\sqrt{2}} - \frac{4}{\sqrt{2}} = 0$.

Distance of a point to a plane

Given a plan defined by its norm θ and the relative offset θ_0, which is the distance of a generic point x from such plane ? Let's start by calling $\vec{f}$ the vector between any point on the plan x_p and the point x, that is $\vec{f} = \vec{x} - \vec{x_p}$. The distance $\|\vec{d}\|$ of the point with the plane is then $\|\vec{d}\| = \|\vec{f}\| * cos(\alpha)$, where α is the angle between the vector $\vec{f}$ and $\vec{d}$.

But we know also that $\vec{f} \cdot \vec{\theta} = \|\vec{f}\| * \|\vec{\theta}\| * cos(\alpha)$.

By substitution, we find that $\|\vec{d}\| = \|\vec{f}\| * \frac{\vec{f} \cdot \vec{\theta}}{\|\vec{f}\| * \|\vec{\theta}\|} = \frac{(\vec{x} - \vec{x_p}) \cdot \vec{\theta}}{\|\vec{\theta}\|} = \frac{\vec{x} \cdot \vec{\theta} + \theta_0}{\|\vec{\theta}\|}$

The distance is positive when x is on the same side of the plane as $\vec{\theta}$ points and negative when x is on the opposite side.

For example the distance between the point $x = (6, 2)$ and the plane as define earlier with $\theta = (1, 1)$ and $\theta_0 = -4$ is $\frac{\vec{x}\cdot\vec{\theta}+\theta_0}{\|\vec{\theta}\|} = \frac{\begin{bmatrix}6\\2\end{bmatrix}\cdot\begin{bmatrix}1\\1\end{bmatrix}-4}{\|\begin{bmatrix}1\\1\end{bmatrix}\|} = \frac{8-4}{\sqrt{2}} = 2 * \sqrt{2}$

Projection of a point on a plane

We can easily find the projection of a point on a plane by summing to the positional vector of the point, the vector of the distance from the point to the plan, in turn obtained multiplying the distance (as found earlier) by the *negative* of the unit vector of the normal to the plane.

Algebraically: $\overrightarrow{x_p} = \vec{x} - \frac{\vec{x}\cdot\vec{\theta}+\theta_0}{\|\vec{\theta}\|} * \frac{\vec{\theta}}{\|\theta\|} = \vec{x} - \vec{\theta} * \frac{\vec{x}\cdot\vec{\theta}+\theta_0}{\|\vec{\theta}\|^2}$

For the example before, the point x_p is given by $\begin{bmatrix}6\\2\end{bmatrix} - \frac{\begin{bmatrix}1\\1\end{bmatrix}}{\sqrt{2}} * 2 * \sqrt{2} = \begin{bmatrix}6\\2\end{bmatrix} - \begin{bmatrix}2\\2\end{bmatrix} = \begin{bmatrix}4\\0\end{bmatrix}$

On this subject see also:

- https://www.khanacademy.org/math/linear-algebra/vectors-and-spaces/dot-cross-products/v/defining-a-plane-in-r3-with-a-point-and-normal-vector
- https://www.khanacademy.org/math/linear-algebra/vectors-and-spaces/dot-cross-products/v/point-distance-to-plane

<a name="“lossfunctiongradientdescentandchainrule<aname="“test”>”>

Loss Function, Gradient Descent, and Chain Rule

Loss Function

This argument in details: segments 3.4 (binary linear classification), 5.4 (Linear regression)

The loss function, aka the *cost function* or the *race function*, is some way for us to value how far is our model from the data that we have.

We first define an “error” or “Loss”. For example in Linear Regression the “error” is the Euclidean distance between the predicted and the observed value:

$L(x, y; \Theta) = \sum_{i=1}^{n} |\hat{y} - y| = \sum_{i=1}^{n} |\theta_1 x + \theta_2 - y|$

The objective is to minimise the loss function by changing the parameter theta. How?

Gradient Descent

This argument in details: segments 4.4 (gradient descent in binary linear classification), 4.5 (stochastic gradient descent) and 5.5 (SGD in linear regression)

The most common iterative algorithm to find the minimum of a function is the gradient descent.

We compute the loss function with a set of initial parameter(s), we compute the gradient of the function (the derivative concerning the various parameters), and we move our parameter(s) of a small delta against the direction of the gradient at each step:

$$\hat{\theta}_{s+1} = \hat{\theta}_{s+1} - \gamma \nabla L(x, y; \theta)$$

The γ parameter is known as the *learning rate*.

- too small learning rate: we may converge very slowly or end up trapped in small local minima;
- too high learning rate: we may diverge instead of converge to the minimum

Chain rule

How to compute the gradient for complex functions.

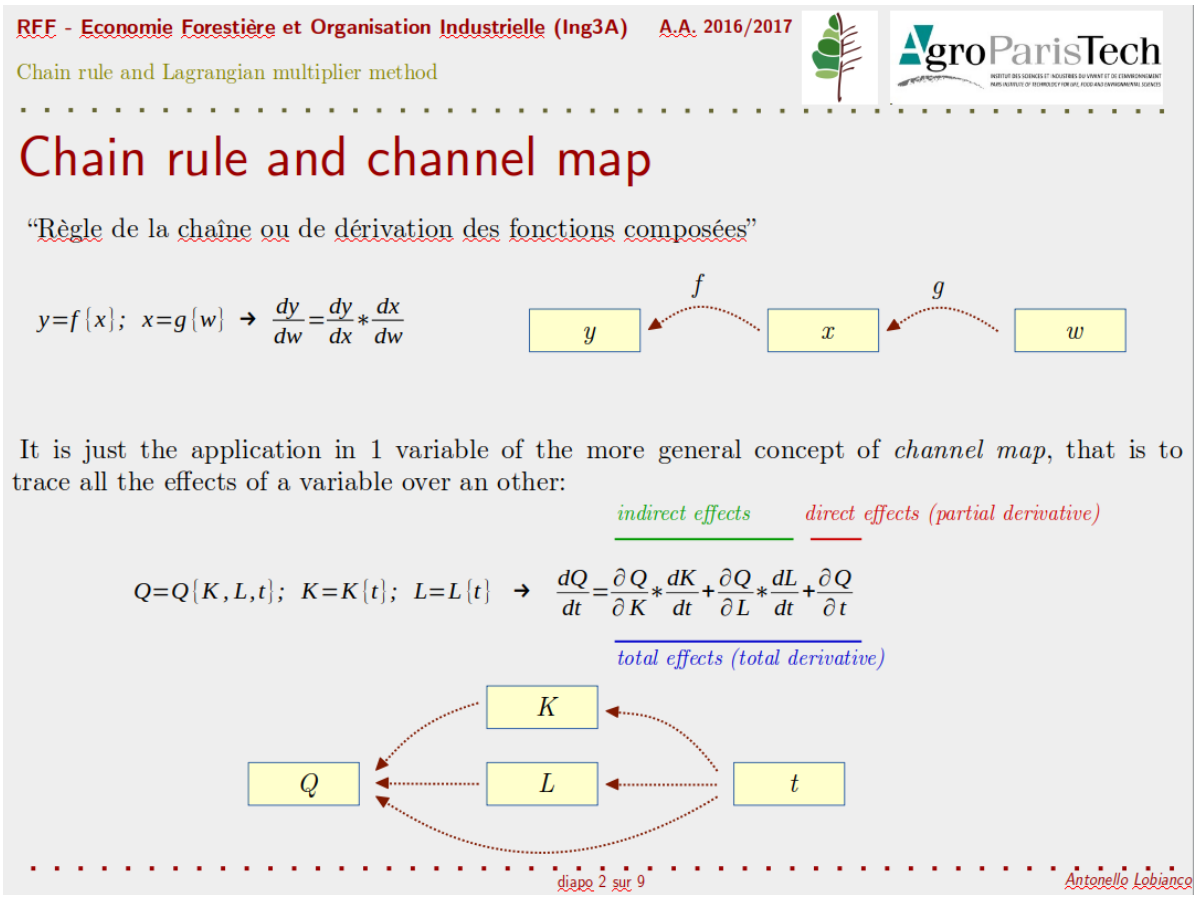
$$\frac{\partial y}{\partial x} = \frac{\partial y}{\partial z} * \frac{\partial z}{\partial x}$$

e.g. $\hat{y} = \frac{1}{1+e^{-(\theta_1 x + \theta_2)}} = (1 + e^{-(\theta_1 x + \theta_2)})^{-1}$, $\frac{\partial \hat{y}}{\partial \theta_1} = -\frac{1}{(1+e^{-(\theta_1 x + \theta_2)})^2} * e^{-(\theta_1 x + \theta_2)} * -x$

For computing derivatives one can use SymPy, a library for symbolic computation. In this case the derivative can be computed with the following script:

```
from sympy import *
x, p1, p2 = symbols('x p1 p2')
y = 1/(1+exp( - (p1*x + p2)))
dy_dp1 = diff(y,p1)
print(dy_dp1)
```

It may be useful to recognise the chain rule as an application of the *chain map*, that is tracking all the effect from one variable to the other:



Geometric progressions and series

Geometric progressions are sequence of numbers where each term after the first is found by multiplying the previous one by a fixed, non-zero number called the *common ratio*.

It results that geometric series can be wrote as $a, ar, ar^2, ar^3, ar^4, \ldots$ where r is the common ratio and the first value a is called the *scale factor*.

Geometric series are the sum of the values in a geometric progression.

Closed formula exist for both finite geometric series and, provided $|r| < 1$, for infinite ones:

- $\sum_{k=m}^{n} ar^k = \frac{a(r^m - r^{n+1})}{1-r}$ with $r \neq 1$ and $m < n$
- $\sum_{k=m}^{\infty} ar^k = \frac{ar^m}{1-r}$ with $|r| < 1$ and $m < n$.

Where m is the first element of the series that you want to consider for the summation. Typically $m = 0$, i.e. the summation considers the whole series from the scale factor onward.

For many more details on geometric progressions and series consult the relative excellent Wikipedia entry.

[MITx 6.86x Notes Index]

[MITx 6.86x Notes Index]

Unit 01 - Linear Classifiers and Generalizations

Course Introduction

There are lots and lots of applications out there, but what's so interesting about it is that, in terms of algorithms, there is a relatively small toolkit of algorithms you need to learn to understand how these different applications can work so nicely and be integrated in our daily life.

The course covers these 3 topics

Supervised Learning

- Make predictions based on a set of data for which correct examples are given
- E.g. Attach the label "dog" to an image, based on many couple (image/dog label) already provided
- The correct answers of the examples provided to the algorithm are already given
- What to produce is given explicitly as a target
- First 3 units in increasing complexity from simple linear classification methods to more complex neural network algorithms

Unsupervised Learning

- A generalisation of supervised learning
- The target is no longer given
- "Learning" to produce output (still at the end, "make predictions"), but now just observing existing outputs, like images, or molecules
- Algorithms to learn how to generate things
- We'll talk about probabilistic models and how to generate samples, mix and match (???)

Reinforced Learning

- Suggest how to act in the world given a certain goal, how to achieve an objective optimally
- It involves making predictions, aspects of SL and UL, but also it has a goal
- Learning from the failures, how to do things better
- E.g. a robot arm trying to grasp an object
- Objective is to control a system

Interesting stuff (solving real problems) happens at the mix of these 3 categories.

Lecture 1. Introduction to Machine Learning

Slides

1.1. Unit 1 Overview

Unit 1 will cover Linear classification

Exercise: predicting whether an Amazon product review, looking at only the text, will be positive or negative.

1.2. Objectives

Introduction to Machine Learning

At the end of this lecture, you will be able to:

- understand the goal of machine learning from a movie recommender example
- understand elements of supervised learning, and the difference between the training set and the test set
- understand the difference of classification and regression - two representative kinds of supervised learning

1.3. What is Machine Learning?

Many examples: Interpretation of web search queries, movie reccomendations, digital assistants, automatic translations, image analysis… playing the go game

Machine learning as a discipline aims to design, understand, and apply computer programs that learn from experience (i.e. data) for the purpose of **modelling**, **prediction**, and **control**.

There are many ways to learn or many reasons to learn:

- You can try to model, understand how things work;
- You can try to predict about, say, future outcomes;
- Or you can try to control towards a desired output configuration.

Prediction

We will start with prediction as a core machine learning task. There are many types of predictions that we can make.:

- We can predict outcomes of *events that occur in the future* such as the market, weather tomorrow, the next word a text message user will type, or anticipate pedestrian behavior in self driving vehicles, and so on.
- We can also try to predict *properties that we do not yet know*. For example, properties of materials such as whether a chemical is soluble in water, what the object is in an image, what an English sentence translates to in Hindi, whether a product review carries positive sentiment, and so on.

1.4. Introduction to Supervised Learning

Common to all these “prediction problems" mentioned previously is that they would be very hard to solve in a traditional engineering way, where we specify rules or solutions directly to the problem. It is far easier to provide examples of correct behavior. For example, how would you encode rules for translation, or image classification? It is much easier to provide large numbers of translated sentences, or examples of what the objects are on a large set of images. I don’t have to specify the solution method to be able to illustrate the task implicitly through examples. The ability to learn the solution from examples is what has made machine learning so popular and pervasive.

We will start with supervised learning in this course. In supervised learning, we are given an example (e.g. an image) along with a target (e.g. what object is in the image), and the goal of the machine learning algorithm is to find out how to produce the target from the example.

More specifically, in supervised learning, we hypothesize a collection of functions (or mappings) parametrized by a parameter (actually a large set of parameters), from the examples (e.g. the images) to the targets (e.g. the objects in the images). The machine learning algorithm then automates the process of finding the parameter of the function that fits with the example-target pairs the best.

We will automate this process of finding these parameters, and also specifying what the mappings are.

So this is what the machine learning algorithms do for you. You can then apply the same approach in different contexts.

1.5. A Concrete Example of a Supervised Learning Task

Let’s consider a movie recommender problem. I have a set of movies I’ve already seen (the **training set**), and I was to use the experience from those movies to make recommendations of whether I would like to see tens of thousands of other movies (the **test set**).

We will compile a **feature vector** (a binary list of its characteristics… genre, director, year…) for each movie in the training set as well for those I want to test (we will denote these feature vectors as x).

I also give my recommendation (again binary) if I want to see again or not the films in the training set.

Now I have the training set (feature vectors with associated labels), but I really wish to solve the task over the test set. It is this discrepancy that makes the learning task interesting. I wish to generalize what I extract from the training set and apply that information to the test set.

More specifically, I wish to learn the mapping from x to labels (plus minus one) on the basis of the training set, and hope and guarantee that that mapping, if applied now in the same way to the test examples, it would work well.

I want to predict the films in the test set based on their characteristics and my previous recommendation on the training set.

1.6. Introduction to Classifiers: Let's bring in some geometry!

On the basis of the training set, pairs of feature vectors associated with labels, I need to learn to map each each x, each future vector, to a corresponding label, which is a plus or minus 1.

What we need is a **classifier** (here denoted with h) that maps from points to corresponding labels.

So what the classifier does is divide the space into essentially two halves – one that's labeled 1, and the other one that's labeled minus 1. –> a "linear classifier" is a classifier that perform this division of the full space in the two half linearly

We need to now, somehow, evaluate how good the classifier is in relation to the training examples that we have. To this end, we define something called **training error** (here denoted with ϵ) It has a subscript n that refers to the number of training examples that I have. And I apply it to a particular classifier. I evaluate a particular classifier, in relation to the training examples that I have.

$$\epsilon_n(h) = \sum_{i=1}^{n} \frac{[\![h(x^i) \neq y^i]\!]}{n}$$

(The double brackets denotes a function that takes a comparison expression inside and returns 1 if the expression is true, 0 otherwise. It is basically an indicator function.)

In the second example of classifier given in the lesson, the training error is 0. So, according to just the training examples, this seems like a good classifier.

The fact that the training error is zero however doesn't mean that the classifier is perfect!

Let's now consider a non-linear classifier.

We can build an "extreme" classifier that accept as +1 only a very narrow area around the +1 training set points.

As a result, this classifier would still have training error equal to zero, but it would classify all the points in the test set, no matter how much close to the +1 training set points they are, as -1 !

what is going on here is an issue called **generalization** --how well the classifier that we train on the training set generalizes or applies correctly, similarly to the test examples, as well? This is at the heart of machine-learning problems, the ability to generalize from the training set to the test set.

The problem here is that, in allowing these kind of classifiers that wrap themselves just around the examples (in the sense of allowing any kind of non-linear classifier), we are making the **hypothesis class**, the set of possible classifiers that we are considering, too large. We improve generalization, we generalize well, when we only have a very limited set of classifiers. And, out of those, we can find one that works well on the training set.

The more complex set of classifiers we consider, the less well we are likely to generalize (this is the same concept as overfitting in a statistical context).

We would wish to, in general, solve these problems by finding a small set of possibilities that work well on the training set, so as to generalize well on the test set.

Training data can be graphically depicted on a (hyper)plane. Classifiers are mappings that take feature vectors as input and produce labels as output. A common kind of classifier is the linear classifier, which linearly divides space (the hyperplane where training data lies) into two. Given a point x in the space, the classifier h outputs $h(x) = 1$ or $h(x) = -1$, depending on where the point x exists in among the two linearly divided spaces.

Each classifier represents a possible “hypothesis" about the data; thus, the set of possible classifiers can be seen as the space of possible hypothesis

1.7. Different Kinds of Supervised Learning: classification vs regression

A supervised learning task is one where you have specified the correct behaviour. Examples are then feature vector and what you want it to be associated with. So you give “supervision” of what the correct behaviour is (from which the term).

Types of supervised learning:

- Multi-way classification : $h : \mathrm{X} \rightarrow$ finite set
- Regression: $h : \mathrm{X} \rightarrow R$
- Structured prediction: $h : \mathrm{X} \rightarrow$ structured object, like a language sentence

Types of machine learning:

- Supervised learning
- Unsupervised learning: You can observe, but the task itself is not well-defined. The problem there is to model, to find the irregularities in how those examples vary.
- Semi-supervised learning
- Active learning (the algorithm asks for useful additional examples)
- Transfer learning
- Reinforcement learning

The training set is illustration of the task, the test set is what we wish to really do well on. But those are examples that we don’t have available at the time we need to select the mapping.

Classification maps feature vectors to categories. The number of categories need not be two - they can be as many as needed. Regression maps feature vectors to real numbers. There are other kinds of supervised learning as well.

Fully labelled training and test examples corresponds to supervised learning. Limited annotation is semi-supervised learning, and no annotation is

unsupervised learning. Using knowledge from one task on another task means you're "transferring" information. Learning how to navigate a robot means learning to act and optimize your actions, or reinforcement learning. Deciding which examples are needed to learn is the definition of active learning.

Lecture 2. Linear Classifier and Perceptron Algorithm

Slides

2.1. Objectives

At the end of this lecture, you will be able to

- understand the concepts of Feature vectors and labels, Training set and Test set, Classifier, Training error, Test error, and the Set of classifiers
- derive the mathematical presentation of linear classifiers
- understand the intuitive and formal definition of linear separation
- use the perceptron algorithm with and without offset

2.2. Review of Basic Concepts

A supervised learning task is when you are given the input and the corresponding output that you want, and you're supposed to learn irregularity between the two in order to make predictions for future examples, or inputs, or feature vectors x.

Test error is defined exactly similarly over the test examples. So defined similarly to the training error, but over a disjoint set of examples, those future examples that you actually wish to do well.

We typically drop the n on it, assuming that the test set is relatively large,

Much of machine learning, really, the theory part is in relating how a classifier that might do well on the training set would also do well on the test set. That's the problem called Generalization, as we've already seen. We can effect generalization by limiting the choices that we have at the time of considering minimizing the training error.

So our classifier here belongs to a **set of classifiers** (here denoted with H), that's not the set of all mappings, but it's a limited set of options that we constrain ourselves to.

The trick here is to somehow guide the selection of the classifier based on the training example, such that it would do well on the examples that we have not yet seen.

In order for this to be possible at all, you have to have some relationship between the training samples and the test examples. Typically, it is assumed that both sets are samples from some large collection of examples as a random subset. So you get a random subset as a training set.

What is this lecture going to be about ?

This lecture we will consider only linear classifiers, such that we keep the problem well generalised (we will return to the question of generalization more formally later on in this course).

We're going to formally define the set of linear classifiers, the set H restricted set of classifiers. We need to introduce parameters that index classifiers in this set so that we can search over the possible classifiers in the set.

We'll see what's the limitation of using linear classifiers.

We'll next need to define the learning algorithm that takes in the training set and the set of classifiers and tries to find a classifier, in that set, that somehow best fits the training set.

We will consider initially the **perceptron algorithm**, which is a very simple online mistake driven algorithm that is still useful as it can be generalized to high dimensional problems. So perceptron algorithm finds a classifier $\hat{h}$, where hat denotes an estimate from the data. It's an algorithm that takes, as an input, the training set and the set of classifiers and then returns that estimated classifier,

2.3. Linear Classifiers Mathematically Revisited

The dividing line here is also called **decision boundary**:

- If x was a one-dimensional quantity, the decision boundary would be a point.
- In 2D, that decision boundary is a line.
- In 3D, it would be a plane.
- And in higher dimensions, it's called hyperplane that divides the space into two halves.

So now we need to parameterize the linear classifiers, so that we can effectively search for the right one given the training set.

Through the origin classifiers

Definition of a decision boundary through the origin:

All points x such that $x : g_1 * x_1 + g_2 * x_2 = 0$ or, calling θ the vector of coefficients and x the vector of the points, $x : \theta \cdot x = 0$.

Note that this means that the vector θ is ortoghonal to the positional vectors of the points in the boundary.

The classifier is now parametrised by theta: $h(x; \theta)$ So each choice of theta defines one classifier. You change theta, you get a different classifier. It's oriented differently. But it also goes through origin.

Our classifier become: $h(x; \theta) = sign(\theta \cdot x)$

Note that this association between the classifier and the parameter vector theta is not unique. There are multiple parameter vectors theta that defined exactly

the same classifier. The norm of the parameter vector of theta is not relevant in terms of the decision boundary.

General linear classifiers

Decision boundary: $x : \theta \cdot x + \theta_0 = 0$.

Our classifier becomes: $h(x; \theta, \theta_0) = sign(\theta \cdot x + \theta_0)$

2.4. Linear Separation

A training set is said to be **linearly separable** if it exists a linear classifier that correctly classifies all the training examples (i.e. if parameter $\hat{\theta}$ and $\hat{\theta}_0$ exists such that $y^i * (\hat{\theta} \cdot x^i + \hat{\theta}_0) > 0 \ \ \forall i = 1, \ldots, n$).

Given θ and θ_0, a linear classifier $h : X \to \{-1, 0, +1\}$ is a function that outputs $+1$ if $\theta \cdot x + \theta_0$ is positive, 0 if it is zero, and -1 if it is negative. In other words, $h(x) = sign(\theta \cdot x + \theta_0)$.

2.5. The Perceptron Algorithm

When the output of a classifier is exactly 0, if the example lies exactly on the linear boundary, we count that as an error, since we don't know which way we should really classify that point.

Training error for a linear classifier:

$\epsilon_n(h) = \sum_{i=1}^{n} \frac{[\![h(x^i) \neq y^i]\!]}{n}$

becomes:

$\epsilon_n(\theta, \theta_0) = \sum_{i=1}^{n} \frac{[\![y^i * (\theta \cdot x^i + \theta_0) \leq 0]\!]}{n}$

We can now turn to the problem of actually finding a linear classifier that agrees with the training examples to the extent possible.

Through the origin classifier

- We start with a $\theta = 0$ (vector) parameter
- We check if, with this parameter, the classifier make an error
- If so, we progressively update the classifier

The update function is $\theta^i = \theta^{i-1} + y^i * x^i$.

As we start with $\theta^0 = \begin{bmatrix} 0 \\ 0 \end{bmatrix}$, the first attempt is always leading to an error and to a first "update" that will be $\theta^1 = \begin{bmatrix} 0 \\ 0 \end{bmatrix} + y^1 * x^1$.

For example, given the pair $(x^1 = \begin{bmatrix} 2 \\ 4 \end{bmatrix}, y^1 = -1)$, θ_1 becomes $\theta^1 = \begin{bmatrix} 0 \\ 0 \end{bmatrix} + -1 * \begin{bmatrix} 2 \\ 4 \end{bmatrix} = \begin{bmatrix} -2 \\ -4 \end{bmatrix}$

Its error is then $\epsilon_n(\theta^1) = [\![y^i * (\theta^1 \cdot x^i)) \leq 0]\!] = [\![(y^i)^2 * ||x^i||^2) \leq 0]\!] = 0$

Now, what are we going to do here over the whole training set, is we start with the 0 parameter vector and then go over all the training examples. And if the i-th example is a mistake, then we perform that update that we just discussed.

So we'd nudge the parameters in the right direction, based on an individual update. Now, since the different training examples might update the parameters in different directions, it is possible that the later updates actually make the earlier undo some of the earlier updates, and some of the earlier examples are no longer correctly classified. In other words, there may be cases where the perceptron algorithm needs to go over the training set multiple times before a separable solution is found.

So we have to go through the training set here multiple times, here capital T times go through the training set, either in order or select at random, look at whether it's a mistake, and perform a simple update.

- function perceptron $\left(\left\{(x^{(i)}, y^{(i)}), i = 1, \ldots, n\right\}, T\right)$:
  - initialize $\theta = 0$ (vector);
  - for $t = 1, \ldots, T$ do
    - for $i = 1, \ldots, n$ do
      - if $y^{(i)}(\theta \cdot x^{(i)}) \leq 0$ then
        - update $\theta = \theta + y^{(i)}x^{(i)}$
  - return θ

So the perceptron algorithm takes two parameters: the training set of data (pairs feature vectors => label) and the T parameter that tells you how many times you try to go over the training set.

Now, this classifier for a sufficiently large T, if there exists a linear classifier through origin that correctly classifies the training samples, this simple algorithm actually will find a solution to that problem. There are many solutions typically, but this will find one. And note that the one found is not, generally, some "optimal" one, where the points are "best" separated, just one where the points *are* separated.

Generalised linear classifier

We can generalize the perceptron algorithm to run with the general class of linear classifiers with the offset parameter. The only difference here is that now we initialize the parameter back to 0, as well as the scalar to 0, that we consider also θ_0 in the error check, that we update θ_0 as well and that we return it together with θ:

- function perceptron $\Big(\{(x^{(i)}, y^{(i)}), i = 1, \ldots, n\}, T\Big)$:
  - initialize $\theta = 0$ (vector), $\theta_0 = 0$ (scalar);
  - for $t = 1, \ldots, T$ do
    - for $i = 1, \ldots, n$ do
      - if $y^{(i)}(\theta \cdot x^{(i)} + \theta_0) \leq 0$ then
        - update $\theta = \theta + y^{(i)} x^{(i)}$
        - update $\theta_0 = \theta_0 + y^{(i)}$
  - return θ, θ_0

The update of θ_0 can be seen as the update of a linear model trough the origin, where the offset is a further dimension of the data:

The model $\theta \cdot x + \theta_0$ can be then see equivalently as $\begin{bmatrix} \theta \\ \theta_0 \end{bmatrix} \cdot \begin{bmatrix} x \\ 1 \end{bmatrix}$.

The update function $\theta = \theta + x^i * y^i$ becomes then $\begin{bmatrix} \theta \\ \theta_0 \end{bmatrix} = \begin{bmatrix} \theta \\ \theta_0 \end{bmatrix} + y^i * \begin{bmatrix} x \\ 1 \end{bmatrix}$ from which, going back to our original model, we obtain the given update functions.

So now we have a general learning algorithm. The simplest one, but it can be generalized to be quite powerful and therefore, hence it is a useful algorithm to understand.

Code implementation: functions `perceptron_single_step_update()`, `perceptron()`, `average_perceptron()`, `pegasos_single_step_update()` *and* `pegasos()` *in project1.*

Lecture 3 Hinge loss, Margin boundaries and Regularization

Slides

3.1. Objective

Hinge loss, Margin boundaries, and Regularization

At the end of this lecture, you will be able to

- understand the need for maximizing the margin
- pose linear classification as an optimization problem
- understand hinge loss, margin boundaries and regularization

3.2. Introduction

Today, we will talk about how to turn machine learning problems into optimization problems. That is, we are going to turn the problem of finding a linear classifier on the basis of the training set into an optimization problem that can be solved in many ways. Today's lesson we will talk about what linear large margin classification is and introduce notions such as margin loss and

regularization.

Why optimisation ?

The perceptron algorithm choose a correct classifier, but there are many possible correct classifiers. Between them, we would favor the solution that somehow is drawn between the two sets of training examples, leaving lots of space on both sides before hitting the training examples. This is known as a **large margin classifier**.

A large margin linear classifier still correctly classifies all of the test examples, even if these are noisy samples of unknown true values. A large margin classifier in this sense is more robust against noises in the examples. In general, large margin means good generalization performance on test data.

How to find such large margin classifier? --> optimisation problem

We consider the parallel plane to the classifier passing by the closest positive and negative point as respectively positive and negative **margin boundaries**, and we want them to be equidistant from the classifier.

The goal here is now to use these margin boundaries essentially to define a fat decision boundary that we will still try to fit with the training examples. So we will push these margin boundaries apart that will force us to reorient there and move that system boundary into a place that carves out this large empty space between the two sets of examples.

We will minimise an objective function made of two terms, the **regularisation term**, that push to set the boundaries as much apart as possible, but also a **loss function term**, that gives us the penalty from points that now, as the boundaries extend, got trapped inside the margin or even more to the other part (becoming misclassified)

3.3. Margin Boundary

The first thing that we must do is to define what exactly the margin boundaries are and how we can control them, how far they are from the decision boundary. Remember that they are equidistant from the decision boundary.

Given the linear equation $\theta \cdot x + \theta_0 = k$, the decision boundary is the set of points where $k = 0$, the positive margin boundary is the set of points where k is some positive constant and the negative margin boundary is where k is some negative constant.

The equation of the decision boundary can be wrote in normalised version by dividing it by the norm of the θ vector:

$$\frac{\theta}{||\theta||} \cdot x + \frac{\theta_0}{||\theta||} = \frac{k}{||\theta||}$$

3.4. Hinge Loss and Objective Function

The objective is to maximise the distance of the decision boundary from the

marginal boundaries, $k/||\theta||$ for the choosen k (plus/minus 1 in the lesson). This is equivalent to minimise $||\theta||$, in turn equivalent to minimize $\frac{||\theta||^2}{2}$

We can define the **Hinge loss** function as a function of the amount of *agreement* (here denoted with z) between the classifier score and the data given by an equation we already saw: $y^i * (\theta \cdot x^i + \theta_0)$.

Previously, in the perceptron algorithm, we used the indicator function of such agreement in the definition of the error ϵ.

Now we use its value, as the loss function is defined as:

$loss(z) = 0$ if $z \geq 1$ (the point is outside the boundary, on the right direction) and $loss(z) = 1 - z$ if $z < 1$ (either the point is well classified, but inside the boundary - $0 \leq z < 1$ - or it is even misclassified - $z \leq 0$).

Note that while for a point on the decision boundary, being exactly on the decision boundary implies a misclassification, being on the (right) margin boundary implies a zero loss.

A bit of terminology

- **Support vectors** are the data points that lie closest to the decision surface (or hyperplane). They are the data points most difficult to classify but they have direct bearing on the optimum location of the decision surface.
- **Support-vector machines** (SVMs, also support-vector networks) are supervised learning models with associated learning algorithms that analyze data used for classification and regression analysis. Given a set of training examples, each marked as belonging to one or the other of two categories, an SVM training algorithm builds a model that assigns new examples to one category or the other, making it a non-probabilistic binary linear classifier (SVMs can perform also non-linear classification using what is called the kernel trick, implicitly mapping their inputs into high-dimensional feature spaces).
- In mathematical optimization and decision theory, a **Loss function** or cost function is a function that maps an event or values of one or more variables onto a real number intuitively representing some “cost” associated with the event. An optimization problem seeks to minimize a loss function. An objective function is either a loss function or its negative (in specific domains, variously called a reward function, a profit function, a utility function, a fitness function, etc.), in which case it is to be maximized.
- The **Hinge loss** is a specific loss function used for training classifiers. The hinge loss is used for “maximum-margin” classification, most notably for support vector machines (SVMs). For an intended output $y = \pm 1$ and a classifier score s, the hinge loss of the prediction s is defined as $\ell(s) = \max(0, 1 - y \cdot s)$. Note that s should be the “raw” output of the classifier’s decision function, not the predicted class label. For instance, in linear SVMs, $s = \theta \cdot \mathbf{x} + \theta_0$, where $\mathbf{x}$ is the input variable(s). When y and s have the same sign (meaning s predicts the right class) and $|s| \geq 1$, the hinge loss $\ell(s) = 0$. When they have opposite signs, $\ell(s)$ increases linearly with s, and similarly if $|s| < 1$, even if it has the same sign (correct prediction, but not by enough margin).

We are now ready to define the objective function (to minimise) as:

$J(\theta, \theta_0) = \frac{1}{n} \sum_{i=1}^{n} \text{Loss}_h(y^{(i)} * (\theta \cdot x^{(i)} + \theta_0)) + \frac{\lambda}{2} \mid\mid \theta \mid\mid^2$

Now, our objective function how we wish to guide the solution is a balance between the two. And we set the balance by defining a new parameter called **regularization parameter** (in the previous equation denoted by λ) that simply weighs how these two terms should affect our solution. Regularization parameter here is always greater than 0.

Greater λ : we will favor large margin solutions but potentially at a cost of incurring some further loss as the margin boundaries push past the examples.

Optimal value of θ and θ_0 is obtained by minimizing this objective function. So we have turned the learning problem into an optimization problem.

Code implementation: functions `hinge_loss_single()`, `hinge_loss_full()` *in project1.*

Homework 1

1. Perceptron Mistakes

1. (e) Perceptron Mistake Bounds

Novikoff Theorem (1962): https://arxiv.org/pdf/1305.0208.pdf

Assumptions:

- There exists an optimal θ^* such that $\frac{y^{(i)}(\theta^* x^{(i)})}{\|\theta^*\|} \geq \gamma \forall i = 1, \ldots, n$ and some $\gamma > 0$ (the data is separable by a linear classifier through the origin)
- All training data set examples are bounded by being $\|x^{(i)}\| \leq R, i = 1, \cdots, n$

Predicate:

Then the number of k updates of the perceptron algorithm is bounded by $k < \frac{R^2}{\gamma^2}$

Lecture 4. Linear Classification and Generalization

Slides

4.1. Objectives

At the end of this lecture, you will be able to:

- understanding optimization view of learning

- apply optimization algorithms such as gradient descent, stochastic gradient descent, and quadratic program

4.2. Review and the Lambda parameter

Last time (lecture 3), we talked about how to formulate maximum margin linear classification as an optimization problem. Today, we're going to try to understand the solutions to that optimization problem and how to find those solutions.

The objective function to minimise is still

$$J(\theta, \theta_0) = \frac{1}{n} \sum_{i=1}^{n} \text{Loss}_h(y^{(i)} * (\theta \cdot x^{(i)} + \theta_0)) + \frac{\lambda}{2} \mid\mid \theta \mid\mid^2$$

where the first term is the average loss and the second one is the regularisation.

The average loss is what we are trying to minimize.

Minimizing the regularization term will try instead to push the margin boundaries further and further apart.

And the balance between these two are controlled by the regularization parameter lambda.

Minimising the regularisation term $\frac{\lambda}{2} \mid\mid \theta \mid\mid^2$ we maximise the distance of the margin boundary $\frac{1}{||\theta||}$.

And as we do that, we start hitting the actual training examples. The boundary needs to orient itself, and we may start incurring losses.

The role of the lambda parameter

As we change the regularization parameter, lambda, we change the balance between these two terms. The larger the value lambda is, the more we try to push the margin boundaries apart; the smaller it is, the more emphasis we put on minimizing the average laws on the training example, reducing the distance of the margin boundaries up to the extreme $\lambda = 0$ where the margin boundaries themselves collapse towards the actual decision boundary.

The further and further the margin boundaries are posed, the more the solution will be guided by the bulk of the points, rather than just the points that are right close to the decision boundary. So the solution changes as we change the regularization parameter.

In other terms:

- $\lambda = 0$: if we consider a minimisation of only the loss function, without the margin boundaries, (admitting a separable problem) we would have infinite solutions within the right classifiers (similarly to the perceptron algorithm);
- $\lambda = \epsilon^+$: With a very small (but positive) λ, the margin boundaries would be immediately pushed to the first point(s) at the minimal distance to the

classifier (aka the “support vectors”, from which the name of this learning method), and only this point(s) would drive the “optimal” classifier within all the possible ones (we overfit relatively to these boundary points);
- $\lambda = \text{large}^{+}$: As we increase λ, we are starting to give weight also to the points that are further apart, and the optimal classifier may be changing as a consequence, eventually even misclassifying some of the closest points, if there are enough far points pushing for such classifier.

4.3. Regularization and Generalization

We minimise a new function $J/\lambda = \frac{1}{\lambda} * \frac{1}{n} \sum_{i=1}^{n} \text{Loss}_h(\cdot) + \frac{1}{2} \mid\mid \theta \mid\mid^2$ (as λ is positive minimising the new function is equal to minimising the old one).

We can think the objective function as function of the $1/\lambda$ ratio (that we denote as “c”):

- $1/\lambda$ small -> wide margins, high losses
- $1/\lambda$ big -> narrow margins, small/zero losses

Such function, once minimised, will result in optimal classifiers $h(\theta^*, \theta_0^*, c)$ from which I can compute the average loss. Such average loss would be a convex, monotonically decreasing function of c, although it would not reach zero losses even for very large values of c if the problem is not linearly separable.

If we could measure also the test loss (from the test set) obtained from such optimal classifiers, we would find a typical u-shape where test loss first decreases with c, but then would growth up again. Also, the test loss (or “error”) would be always higher than the training loss, as we are fitting from the training set, not the test set.

We call c^* the “optimal” c value that minimise the test error. If we could find it, we would have found the value of λ that best generalise the method.

Now, we cannot do that, but we will talk about how to approximately do that.

With c below c^* I am underfitting, with c above c^* I am overfitting (around the point(s) closest to the decision margin). How to find c^* ? By dividing my training set in a “new” training set, and a (smaller) **validation set**, our new “pretending” test set.

Using this “new” training set we can find the optimal classifier as function of c, and at that point use $h(\theta^*, \theta_0^*, c)$ to evaluate the **validation loss** (or “validation error”) to numerically find an approximate value of c^* .

4.4. Gradient Descent

Objective: So far we have seen how to qualitatively understand the type of solutions that we get when we vary the regularization parameter and optimize with respect to theta and theta naught. Now, we are going to talk about, actually, algorithms for finding those solutions.

Gradient descent algorithm to find the minimum of a function $J(\theta)$ with respect to a parameter $\theta \in \mathbb{R}$ (for simplicity).

- I start with a given θ_{start}
- I evaluate the derivative $\partial J / \partial \theta$ at θ_{start}, i.e. the slope of the function in θ_{start}. If this is positive, increasing θ will increase the J function and the opposite if it is negative. In both cases, if I move θ in the opposite direction of the slope, I move toward a lower value of the function.
- My new, updated value of θ is then $\theta = \theta - \eta \frac{\partial J}{\partial \theta}$ where η is known as the **step size** or **learning rate**, and the whole equation as **gradient descent update rule**.
- We stop when the variation in θ goes below a certain threshold.

If η is too small: we may converge very slowly or end up trapped in small local minima;

If η is too small: we may diverge instead of converge to the minimum (going to the opposite direction respect to the minimum).

If the function is strictly convex, then there would be a constant learning rate small enough that I am guaranteed quickly to get to the minimum of that function.

Multi-dimensional case

The multi-dimensional case is similar, we update each parameter with respect to the corresponding partial derivative (i.e. the corresponding entry in the gradient):

$\theta_j = \theta_j - \eta \frac{\partial J}{\partial \theta_j}$ or, in vector form, $\theta = \theta - \eta \nabla J(\theta)$

(where $\nabla J(\theta_\mathbf{j})$ is the gradient of J, i.e. the column vector concatenating the partial derivatives with respect to the various parameters)

4.5. Stochastic Gradient Descent

Let firs note that we can write $\frac{1}{n} \sum_{i=1}^{n} (a_i) + b$ as $\frac{1}{n} \sum_{i=1}^{n} (a_i + b)$.

Our original objective function can hence be written as

$J(\theta, \theta_0) = \frac{1}{n} \sum_{i=1}^{n} \left[\text{Loss}_h(y^{(i)} * (\theta \cdot x^{(i)} + \theta_0)) + \frac{\lambda}{2} \mid\mid \theta \mid\mid^2 \right]$

Let's also, for now, simplify the calculation omitting the offset parameter:

$J(\theta) = \frac{1}{n} \sum_{i=1}^{n} \left[\text{Loss}_h(y^{(i)} * (\theta \cdot x^{(i)})) + \frac{\lambda}{2} \mid\mid \theta \mid\mid^2 \right]$

We can now see the above objective function as an *expectation* of the function $J(\theta)$: $E[J(\theta)] = \frac{1}{n} \sum_{i=1}^{n} J[\theta]$ where t

We can then use a stochastic approach (SGD, standing for **Stochastic Gradient Descen**), where we sample at random from the training set a single data pair i, we compute the gradient of the objective function with respect to that single parameter $\frac{\partial J_i(\theta)}{\partial \theta}$, we compute an update of theta following the same rule as before ($\theta = \theta - \eta_t \nabla J_i(\theta)$), we sample an other data pair j, we re-compute the gradient with respect to this new pair, we update again ($\theta = \theta - \eta_t \nabla J_j(\theta)$), and so on.

This has the advantage to be more efficient than taking all the data, compute the objective function and the gradients with respect to all the data points (could be milions!), and perform the gradient descent with respect to such function.

By sampling we introduce however some stochasticity and noises, and hence we need to put care on the learning parameter that need to be reduced to avoid to diverge.

The new learning rate η needs to:

- Be lower than the one we would employ for the deterministic gradient descent algorithm in order to reduce the variance from the stochasticity that we're introducing:
  - reduce at each update t with the limit as the steps go to infinite to go to zero $\lim_{t \to \infty} \eta_t = 0$
  - be such that are "square summable", i.e. the sum of the square values must be finite, $\sum_{t=1}^{\infty} \eta_t^2 < \infty$
- But at the same time retain enough weight that we can reach the minimum wherever it is:
  - if we sum at the infinite the learning rates we must be able to reach infinity, i.e. $\sum_{t=1}^{\infty} \eta_t = \infty$

One possible learning rate that satisfies all the above constraints is $\eta_t = \frac{1}{1+t}$.

Which is the gradient of the objective function with respect to data pair i ?

$$\nabla J_i(\theta) = \begin{cases} 0 & \text{when loss=0} \\ -y^i \mathbf{x}^{(i)} & \text{when loss >0} \end{cases} + \lambda \theta$$

The update rule for the i data pair is hence:

$$\theta = \theta - \eta_t * \left(\begin{cases} 0 & \text{when loss=0} \\ -y^i \mathbf{x}^{(i)} & \text{when loss >0} \end{cases} + \lambda \theta \right)$$

There are three differences between the update in the stochastic gradient descent method and the update in our earlier perception algorithm ($\theta = \theta + y^{(i)} \mathbf{x}^{(i)}$):

- First, is that we are actually using a decreasing learning rate due to the stochasticity.
- The second difference is that we are actually performing the update, even if we are correctly classifying the example, because the regularization term

will always yield an update, regardless of what the loss is. And the role of that regularization term is to nudge the parameters a little bit backwards, so decrease the norm of the parameter vector at every stamp, which corresponds to trying to maximize the margin.
- To counterbalance that, we will get a non-zero derivative from the last terms, if the loss is non-zero. And that update looks like the perceptor update, but it is actually made even if we correctly classify the example. If the example is within the margin boundaries, you would get a non-zero loss.

An other point of view on SGD compared to a deterministic approach:

The original concern for using SGD is due to computational constraint: it is inefficient to compute the gradients over all the samples and update the model, think about you have more than a million training samples, and you can only make progress after computing all gradients. Thus, instead of compute the exact value of the gradient over the entire training data set, we randomly sample a fraction of them to estimate the true gradient. You see, there is a trade-off controlled by the batch size (number of samples used in each SGD updates), large batch size is more accurate but slower, and it need to be tuned in practice.

Nowadays, researchers start to understand SGD from a different perspective. The noise caused by SGD could make the trained more robust, (high level idea, noisy updates will make you converge to a flatter minima), which results in a better generalization performance. Thus, SGD could also be viewed as a way of regularization, in that sense, we may say it is better compared to gradient descent.

4.6. The Realizable Case - Quadratic program

Let's consider a simple case where (a) the problem is separable and (b) we do not allow any error, i.e. we want the loss part of the objective function to be zero (that's the meaning of "realisable case": we set as constraint that the loss function must return zero).

The problem becomes:

$\min_{(\theta,\theta_0)} \frac{1}{2}||\theta||^2$

Subject to:

$y^{(i)} * (\theta \cdot \mathbf{x}^{(i)} + \theta_0) \geq 1 \ \ \forall i = 1, \ldots, n$

We want hence extend the margins as much as possible while keeping zero losses, that is until the first data pair(s) is reach (we can't go over them otherwise we would start encoring a loss). Note that this is exactly equivalent to the minimisation of our original $J(\theta)$ function when λ is very small but not zero. There the zero loss output (in case of a separable problem) is the result of the minimisation of the loss function, here is it imposed as a constraint.

This is a problem quadratic in the objective and with linear constraints, and it can be solved with so-called quadratic solvers.

Relaxing the loss constraint and incorporate the loss in the objective would still leave the problem as a quadratic one.

Homework 2

Project 1: Automatic Review Analyzer

Recitation 1: Tuning the Regularization Hyperparameter by Cross Validation and a Demonstration

Outline

Supervised learning: Importance to have a model Objective: classification or regression

Cross validation: using the validation set to find the optimal parameters of our learning algorithm.

Supervised Learning

obj: derive a function f_{est} that approximate the real function f that link an input x to an output y.

We started by sampling from X and corresponding y --> training dataset We call the relation between this sampled pairs f_{data}

Our first obj is then to find a f_{est}, parametrised by some parameters (θ, θ_0), that approximate *this* f_{data}.

In order to find (θ, θ_0), we use an optimisation approach that minimise the function $J()$ made of a "loss" term $L(\theta, \theta_0; x, y)$ that describes the difference between our f_{est} and f_{data}.

But $f_{data} \neq f$. Because of that we add to our loss function a regularisation term $R(\theta)$ that constraints f_{est} not to be too similar to f_{data} that then is no longer able to generalise to f.

A **hyperparameter** α then balance these two terms in our function to minimise. And we remain with the task to choose this parameter, as it is not determined by the minimisation of the function as it is for (θ, θ_0). We'll see that to find α we will use indeed the cross validation using training data only and how choosing alpha will affect the performance of the model.

We'll go trough cross the method of cross validation which this is actually how we find α in practice using training data only, not the test data.

Support Vector Machine

What are the Loss and the regularisation functions for Support Vector Machines (SVM)

SVM obj: maximise the margins of the decision boundary

Distance from point i to the decision boundary:

$\gamma = \frac{y^i * (\theta \cdot x^i + \theta_0)}{||\theta||}$ (note that this is a distance with positive sign if the point side match the label side, negative otherwise)

The margin d is the minimal distance of any point with the decision boundary:

$d = \min_i \gamma(x^i, y^i, \theta, \theta_0)$

Large margin --> more ability of our model to generalise

Loss term

For SVM the loss term is the hinge loss L_h:

$$L_h = f\left(\frac{\gamma}{\gamma_{ref}}\right) = \begin{cases} 1 - \frac{\gamma}{\gamma_{ref}} & \text{when} \gamma < \gamma_{ref} \\ 0 & \text{otherwise} \end{cases}$$

As a function of γ, the hinge loss function for each point is above 1 for negative gammas, is 1 for $\gamma = 1$, continue to linearly decrease up to reaching 0 when $\gamma = \gamma_{ref}$ and then remains zero, where γ_{ref} is indeed the margin d as previously defined.

Regularisation term

The goal of the regularisation term is to make the model more generalisable.

For that, we want to maximise the margin, hence γ_{ref}. As we have a minimisation problem, that is equivalent to minimise $1/\gamma_{ref}$ or (as did in practice) $1/\gamma_{ref}^2$.

Objective function

$J = \frac{1}{n} \sum_{i=1}^{n} L_h(\frac{\gamma_i}{\gamma_{ref}}) + \alpha * \frac{1}{\gamma_{ref}^2}$

We now are left to define what γ_{ref} is in terms of (θ, θ_0).

Maximum margin

Note that we can scale θ by any constant (i.e. change it's magnitude) without changing the decision boundary (obviously also θ_0 will be rescaled).

In particular, we can scale θ such that $y^m(\theta \cdot x^m + \theta_zero) = 1$, where m denotes

the point at the minimum distance with the decision boundary.

In such case θ_{ref} simply becomes $\frac{1}{||\theta||}$.

The objective function becomes the one we saw in the lecture:

$J = \frac{1}{n} \sum_{i=1}^{n} L_h(\$y^i(\theta \cdot x^i + \theta_0)) + \alpha * ||\theta||^2$

Note that the regularisation term is a function of θ alone.

Testing and Training Error as Regularization Increases

We will now see how different values of alpha affect the performance of the model.

Our objective function is

$J = L_(\theta, \theta_0, x_{data}, y_{data}) + \alpha R(\theta)$

At $\alpha = 0$, the obj model becomes just the loss model, and our functions should approximate very well f_{data}, the relation between training data and training outputs.

Accuracy is defined as $1/J$, as we are trying to minimise J.

As we increase α, J increases and the accuracy reduces.

What about the J function as computed from the testing dataset? As α increases it first reduce, as the model is gaining generability, but then it start back to increases as the model becomes too much general. The opposite for its accuracy.

There is indeed a value α^* for which indeed the J function under the testing set is minimised (accuracy is maximised).

And what't the α that we want to use. How do we find it with only training set data ? (we can't use testing data, as we don't yet have that data). The method is "cross validation" (next video).

Cross Validation

We want to find α^* . We just split our training data in a "new" training data, and a "validation set" to act as the "test set" and fine tune our α hyperparameter.

We start to discretize our hyperparameter in a number of K values to test (there are many ways to do that... ex-ante using a homogeneous grid or sampling approach, or updating the values to test as we have the results of the first α we tested...)

We then divide our training data in n partitions (where n depends on the data available), and, for each α_k, we choose one of that n partitions to be the

"validation set": we minimise the J function using α_k and the remaining $(n-1)$ partitions, and we compute the accuracy $S_n(\alpha_k)$ for the validation set.

We then compute the average accuracy for α_k: $S(\alpha_k) = \frac{1}{n}\sum_{i=1}^{n} S_n(\alpha_k)$.

Finally we "choose" the α^* with the lowest $S(\alpha_k)$.

Tumor Diagnosis Demo

- Python code
- Julia code

In this demo we found the best value of alpha in a SVM model that links breast cancer characteristics (such as cancer texture or size) with its benign/malign nature.

The example use sikit-learn, a library that allows to train your own machine learning algorithm (linear SVM in our case).

We use the function `linear_model.SGDClassifier(loss='hinge', penalty='l2', alpha=alpha[i])` that set up a SVM model using gradient descendent algorithm on hinge loss, using the L2 norm as regularisation term ("penalty") and the specific α we want to test.

We then use `ms.cross_val_score(model, X, y, cv=5)` to actually train the model on the training set divided in 5 different partitions. The function return already an array of the scores of the remaining validation partition. To compute the score for that particular alpha we now need just to average the score array obtained by that function.

[MITx 6.86x Notes Index]

[MITx 6.86x Notes Index]

Unit 02 - Nonlinear Classification, Linear regression, Collaborative Filtering

Lecture 5. Linear Regression

5.1. Unit 2 Overview

Building up from the previous unit, in this unit we will introduce:

- linear regression (output a number in R)

- non-linear classification methods
- recommender problems (sometime called collaborative filtering problems)

5.2. Objectives

At the end of this lecture, you will be able to

- write the training error as least squares criterion for linear regression
- use stochastic gradient descent for fitting linear regression models
- solve closed-form linear regression solution
- identify regularization term and how it changes the solution, generalization

5.3. Introduction

Today we will see Linear Classification In the last unit we saw linear *classification*, where we was trying to land the mapping between the feature vectors of our data ($x^{(t)} \in \mathbb{R}^d$) and the corresponding (binary) label ($y^{(t)} \in \{-1, +1\}$).

This relatively simple set up can be already used to answer pretty complex questions, like making recommendations to buy or not some stocks, where the feature vector is given by the stock prices in the last d-days.

We can extend such problem to return, instead of just a binary output (price will increase - buy, vs price will drop - sell) a more informative output on the extent of the expected variation in price.

With regression we want to predict things that are continuous in nature.

The set up is very similar to before with $x^{(t)} \in \mathbb{R}^d$. The only differences is that now we consider $y^{(t)} \in \mathbb{R}$).

The goal of our model will be to map-- to learn how to map-- feature vectors into these continuous values.

We will consider for now only linear regression:

$f(\mathbf{x}; \theta, \theta_0) = \sum_{i=1}^{d} \theta_i x_i + \theta_0 = \theta \cdot \mathbf{x} + \theta_0$

For compactness of the equations we will consider $\theta_0 = 0$ (we can always think of θ_0 of being just a $d + 1$ dimension of θ).

Are we limiting ourself to just linear relations? No, we can still use linear classifier by doing an appropriate representation of the feature vector, by mapping into some kind of complex space and from that mapping then apply linear regression (note that at this point we will not yet talk about how we can construct this feature vector position. We will assume instead that somebody already gave us an appropriate feature vector).

3 questions to address:

1. Which would be an appropriate objective that can quantify the extent of our mistake?. How do we address the correctness of our output? In classification we had a binary error term, here it must be a range, our prediction could be “almost there” or very far.
2. How do we set up the learning algorithm. We will see two today, a numerical, gradient based ones, and a analytical, closed-form algorithm where we do not need to approximate.
3. How we do regularisation. How we perform a better generalization to be more robust when we don’t have enough training data or when the data is noisy

5.4. Empirical Risk

Let’s deal with the first question, the objective. We want to measure how much our prediction deviates from the know values of y, how far from y they are. We call our objective **empirical risk** (here denoted with R) and will be a sort of average loss of all our data points, depending from the parameter to estimate.

$$R_n(\theta) = \frac{1}{n}\sum_{t=1}^{n} \text{Loss}_h(y^{(t)} - \theta \cdot \mathbf{x}^{(t)}) = \frac{1}{n}\sum_{t=1}^{n} \frac{(y^{(t)} - \theta \cdot \mathbf{x}^{(t)})^2}{2}$$

Why squaring the deviation? Intuitively, since our training data may be noisy and the values that we record may be noisy, if it is a small deviation between our prediction and the true value, it’s OK. However, if the deviation is large, we want really, truly penalize. And this is the behaviour we are getting from the squared function, that the bigger difference would actually result in much higher loss.

Note that the above equation use the squared error as loss function, but other loss functions could also be used, e.g. the hinge loss we already saw in unit 1:

$$\text{Hinge Loss}_h(z) = \begin{cases} 0 & \text{if } z \geq 1 \\ 1 - z & \text{oth.} \end{cases}$$

I will minimise this risk for the known data, but what I really want to do it so get it minimised for the unknown data I don’t already see.

2 mistakes are possible:

1. **structural mistakes** Maybe the linear function is not sufficient for you to model your training data. Maybe the mapping between your training vectors and y’s is actually highly nonlinear. Instead of just considering linear mappings, you should consider a much broader set of function. This is one class of mistakes.
2. **estimation mistakes** The mapping itself is indeed linear, but we don’t have enough training data to estimate the parameters correctly.

There is a trade-off between these two kind of error: on one side, minimising the structural mistakes ask for a broader set of functions with more parameters, but this, at equal training set size, would increase the estimation mistakes. On the other side, minimising the estimation mistakes call for simpler set of functions, with less parameters, where however I become susceptible for structural mistakes.

In this lesson we remain commit to linear regression, and we want to minimise the empirical risk.

5.5. Gradient Based Approach

In this segment we will study the first of the two algorithms to implement the learning phase, the gradient based approach.

The advantage of the Empirical Risk function with the squared error as loss is that it is differentiable everywhere. Its gradient with respect to the parameter is:

$$\nabla_\theta\left(\frac{(y^{(t)}-\theta\cdot\mathbf{x}^{(t)})^2}{2}\right) = -\left(y^{(t)}-\theta\cdot\mathbf{x}^{(t)}\right)*\mathbf{x}^{(t)}$$

We will implement its stochastic variant: we start by randomly select one sample in the training set, look at its gradient, and update our parameter in the opposite direction (as we want to minimise the empirical risk).

The algorithm will then be as follow:

1. We initialise the thetas to zero
2. We randomly pick up a data pair
3. We compute the gradient and update θ as $\theta = \theta - \eta * \nabla_\theta = \theta + \eta * \left(y^{(t)}-\theta\cdot\mathbf{x}^{(t)}\right)*\mathbf{x}^{(t)}$ where η is the learning rate, influencing the size of the movement of the parameter at each iteration. We can have η constant or making it depends from the number of k iteration we already have done, e.g. $\eta = 1/(1+k)$, so to minimise the steps are we get closer to our minimum.

Note that the parameter updates at each step, not only on some “mistake” like in classification (i.e. we treat all deviations as “mistake”), and that the amount depends on the deviation itself (i.e. not of a fixed amount like in classification). Going against the gradient assure that the algorithm self-correct itself, i.e. we obtain parameters that lead to predictions closer and closer to the actual true y.

5.6. Closed Form Solution

The second learning algorithm we study is the closed form (analytical) solution. This is quite an exception in the machine learning field, as typically closed form solutions do not typically exist. But here the empirical risk happens to be a convex function that we can solve it exactly.

Let’s compute the gradient with respect of θ, but this time of the whole empirical risk, not just the loss function:

$$R_n(\hat{\theta}) = \frac{1}{n}\sum_{t=1}^{n}\frac{(y^{(t)}-\hat{\theta}\cdot\mathbf{x}^{(t)})^2}{2}$$

$$\nabla_\theta R_n(\hat{\theta}) = \frac{1}{n}\sum_{t=1}^{n}\nabla_\theta\left(\frac{(y^{(t)}-\hat{\theta}\cdot\mathbf{x}^{(t)})^2}{2}\right) = -\frac{1}{n}\sum_{t=1}^{n}\left(y^{(t)}-\hat{\theta}\cdot\mathbf{x}^{(t)}\right)*\mathbf{x}^{(t)}$$

$$\nabla_\theta R_n(\hat{\theta}) = -\frac{1}{n}\sum_{t=1}^{n}y^{(t)}*\mathbf{x}^{(t)} + \frac{1}{n}\sum_{t=1}^{n}\mathbf{x}^{(t)}*\left(\mathbf{x}^{(t)}\right)^T*\hat{\theta}$$

$\nabla_\theta R_n(\hat{\theta}) = -\mathbf{b} + \mathbf{A}\hat{\theta} = 0$

With $b = \frac{1}{n}\sum_{t=1}^{n} y^{(t)}x^{(t)}$ is a $(d \times 1)$ vector and $\mathbf{A} = \frac{1}{n}\sum_{t=1}^{n} x^{(t)}(x^{(t)})^T$ is an $(d \times d)$ matrix. If $\mathbf{A}$ is invertible we can finally write $\hat{\theta} = \mathbf{A}^{-1}b$.

We can invert $\mathbf{A}$ only if the feature vectors $\mathbf{x}^{(1)}, \ldots, \mathbf{x}^{(n)}$ span over R^d, that is if $n >> d$.

Also, we should put attention that inverting $\mathbf{A}$ is an operation of order $O(d^3)$, so we should be carefully when d is very large, like in bag of words approaches used in sentiment analysis where d can be easily be in the tens of thousands magnitude.

5.7. Generalization and Regularization

We now focus the discussion in how do we assure that the algorithm we found, the parameters we estimated, will be good also for the unknown data, will be robust and not too much negatively impacted by the noise that it is in our training data?

This question is more and more important as we have less data to train the algorithm.

The way to solve this problem is to use a mechanism called regularisation that try to push us away to fit the training data “perfectly” (where we “fit” also the errors, the noises embedded in our data) and try to instead generalise to possibly unknown data.

The idea is to introduce something that push the thetas to zero, so that it would be only worth for us to move our parameters if there is really a very strong pattern that justify the move.

5.8. Regularization

The implementation of the regularisation we see in this lesson is called **ridge regression** and it is the same we used in the classification problem:

The new objective function to minimise becomes:

$J_{n,\lambda} = R_n(\theta) + \frac{\lambda}{2}||\theta||^2$

The first term is our empirical risk, and it catches how well we are fitting the data. The second term, being the square norm of thetas, tries to push the thetas to remain zero, to not move unless there is a significant advantage in doing so. And λ is the parameter that determine the trade off, the relative contribution, between these two terms. Note that being the norm it doesn’t influence any specific dimension of theta. Its role is is actually to determine how much do I care to fit my training examples versus how much do I care to be staying close to zero. In other words, we don’t want any weak piece of evidence to pull our thetas very strongly. We want to keep them grounded in some area and only pulls them when we have enough evidence that it would really, in substantial

way, impact the empirical loss.

In other terms, the effect of regularization is to restrict the parameters of a model to freely take on large values. This will make the model function smoother, leveling the 'hills' and filling the 'vallyes'. It will also make the model more stable, as a small perturbation on x will not change y significantly with smaller $\|\theta\|$.

What's very nice about using the squared norm as regularisation term is that actually everything that we discussed before, both the gradient and closed form solution, can be very easily adjusted to this new loss function.

Gradient based approach with regularisation

With respect to a single point the gradient of J is :

$$\nabla_\theta(\frac{(y^{(t)}-\theta\cdot\mathbf{x}^{(t)})^2}{2}+\frac{\lambda}{2}||\theta||^2)=-(y^{(t)}-\theta\cdot\mathbf{x}^{(t)})*\mathbf{x}^{(t)}+\lambda\theta$$

We can modify the gradient descent algorithm where the update rule becomes:

$$\theta=\theta-\eta*\nabla_\theta=\theta-\eta*(\lambda\theta-(y^{(t)}-\theta\cdot\mathbf{x}^{(t)})*\mathbf{x}^{(t)})=(1-\eta\lambda)\theta+\eta*(y^{(t)}-\theta\cdot\mathbf{x}^{(t)})$$

The difference with the empirical risk without the regularisation term is the $(1-\eta\lambda)$ term that multiply θ trying to put it down at each update.

The closed form approach with regularisation

First solve for θ_O and then solve for θ (see exercise given in Homework 3, tab "5. Linear Regression and Regularization").

In matrix form the closed form solution is:

$\theta=(X^TX+\lambda I)^{-1}X^TY$ (also known as Tikikhonov regularisation)

Note that again we can consider θ_0 just as a further dimension of θ where the X are all ones.

5. 9. Closing Comment

By using regularisation, by requiring much more evidence to push the parameters into the right direction we will increase the mistakes of our prediction within the training set, but we will reduce the test error when the fitted thetas are used with respect to data that has not been used for the fitting step.

However if we continue to increase λ, if we continue to give weight to the regularisation term, then also the testing error will start to increase as well.

Our objective is to find the "sweet spot" of lambda where the test error is those that is minimised, and we can do that using the validation set to calibrate the value that lambda should have.

We saw regularization in the context of linear regression, but we will see regularization across many different machine learning tasks. This is just one way to implement it. We will see some other mechanism to implement regularisation, for example in neural networks.

Lecture 6. Nonlinear Classification

6.1. Objectives

At the end of this lecture, you will be able to

- derive non-linear classifiers from feature maps
- move from coordinate parameterization to weighting examples
- compute kernel functions induced from feature maps
- use kernel perceptron, kernel linear regression
- understand the properties of kernel functions

6.2. Higher Order Feature Vectors

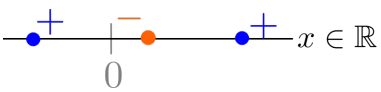
Outline

- Non-linear classification and regression
- Feature maps, their inner products
- Kernel functions induced from feature maps
- Kernel methods, kernel perceptron
- Other non-linear classifiers (e.g. Random Forest)

This lesson deals with non-linear classification, with the basic idea to expand the feature vector, map it to a higher dimensional space, and then feed this new vector to a linear classifier.

The computational disadvantage of using higher dimensional space can be avoided using so-called kernel functions. We will then see linear models applied to these kernel models, and in particular, for simplicity, the perceptron linear model (that when used with kernel function becomes the “kernel perceptron”).

Let’s see an example in 1D: we have the real line on which we have our points we want to classify. We can easily see that if we have the set of points {-3: positively labelled, 2: negatively labelled, 5:positively labelled} there is no linear classifier that can correctly classify this data set:



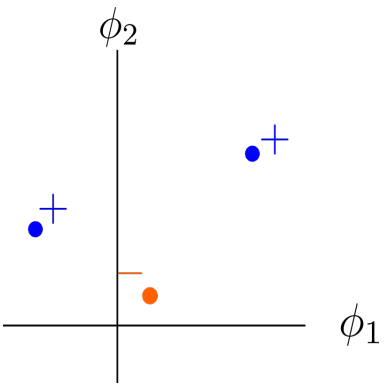
We can remedy the situation by introducing a feature transformation feeding a different type of example to the linear classifier.

We will always include in the new feature vector the original vector itself, so as to be able to retain the power that was available prior to feature transformation. But we will also add to it additional features. Note that, differently from statistics, in Machine Learning we know nothing (assume nothing) about the distribution of our data, so removing the original data to keep only the new

added feature would risk to remove information that is not captured in the transformation.

In this case we can add for example x^2. So the mapping is $\mathbf{x} \in \mathbb{R}^2 = \phi(x \in \mathbb{R}) = \begin{bmatrix} x \\ x^2 \end{bmatrix}$. As result also the θ parameter of the classifier became bidimensional.

Our dataset becomes {(-3,9)[+], (2,4)[-], (5,25)[+]} that can be easily classified by a linear classifier in 2D (i.e. a line) $h(x; \theta, \theta_0) = \text{sign}(\theta \cdot \phi(x) + \theta_0)$:



Note that the linear classifier in the new feature space we had found, back in the original space becomes a non-linear classifier: $h(x; \theta, \theta_0) = \text{sign}(\theta_1 * x + \theta_2 * x^2 + \theta_0)$.

An other example would be having our original dataset in 2D as {(2,2)[+],(-2,2)[-],(-2,-2)[+],(2,-2)[-]} that is not separable in 2D, but it becomes separable in 3D when I use a feature transformation like $\mathbf{x} \in \mathbb{R}^3 = \phi(\mathbf{x} \in \mathbb{R}^2) = \begin{bmatrix} x_1 \\ x_2 \\ x_1 x_2 \end{bmatrix}$, for example by the plane given by $\left(\theta = \begin{bmatrix} 0 \\ 0 \\ 1 \end{bmatrix}, \theta_0 = 0 \right)$.

6.3. Introduction to Non-linear Classification

We can get more and more powerful classifiers by adding linearly independent features, x^2, x^3,... This x, x^2,..., as functions are linearly independent, so the original coordinates always provide something above and beyond what were in the previous ones.

Note that when $\mathbf{x}$ is already multidimensional, even just $\phi(\mathbf{x}) = \mathbf{x}^2$ would result in dimensions exploding, e.g. $\mathbf{x} \in \mathbb{R}^5 = \phi(\mathbf{x} \in \mathbb{R}^2) = \begin{bmatrix} x_1 \\ x_2 \\ x_1^2 \\ x_2^2 \\ \sqrt{2} x_1 x_2 \end{bmatrix}$ (the meaning of the scalar associated to the cross term will be discussed later).

Once we have the new feature vector we can make non-linear classification or regression in the original data making a linear classification or regression in the new feature space:

- Classification: $h(x;\theta,\theta_0) = \text{sign}(\theta \cdot \phi(x) + \theta_0)$
- Regression: $f(x;\theta,\theta_0) = \theta \cdot \phi(x) + \theta_0$

More feature we add (e.g. more polinomian grades we add), better we fit the data. The key question now is when is time to stop adding features ? We can use the validation test to test which is the polynomial form that, trained on the training set, respond better in the validation set.

At the extreme, you hold out each of the training example in turn in a procedure called **leave one out cross validation**. So you take a single training sample, you remove it from the training set, retrain the method, and then test how well you would predict that particular holdout example, and do that for each training example in turn. And then you average the results.

While very powerful, this explicit mapping into larger dimensions feature vectors is indeed… that the dimensions could become quickly very high as our original data is already multidimensional

Let's our original $\mathbf{x} \in \mathbb{R}^d$. Then a feature transformation:

- quadratic (order 2 polynomial): would involve $d+ \approx d^2$ dimensions (the original dimensions plus all the cross products)
- cubic (order 3 polynomial): would involve $d+ \approx d^2+ \approx d^3$ dimensions

The exact number of terms of a feature transformation of order p of a vector of d dimensions is $\sum_{i=1}^{p} \binom{d+i-1}{i}$ (the sum of multiset numbers).

So our feature vector becomes very high-dimensional very quickly if we even started from a moderately dimensional vector.

So we would want to have a more efficient way of doing that – operating with high dimensional feature vectors without explicitly having to construct them. And that is what kernel methods provide us.

6.4. Motivation for Kernels: Computational Efficiency

The idea is that you can take inner products between high dimensional feature vectors and evaluate that inner product very cheaply. And then, we can turn our algorithms into operating only in terms of these inner products.

We define the kernel function of two feature vectors (two different data pairs) applied to a given ϕ transformation as the dot product of the transformed feature vectors of the two data:

$k(x, x'; \phi) \in \mathbb{R}^+ = \phi(x) \cdot \phi(x')$

We can hence think of the kernel function as a kind of similarity measure, how

similar the x example is to the x' one. Note also that being the dot product symmetric and positive, kernel functions are in turn symmetric and positive.

For example let's take x and x' to be two dimensional feature vectors and the feature transformation $\phi(x)$ defined as $\phi(x) = [x_1, x_2, x_1^2, \sqrt(2)x_1x_2, x_2^2]$ (so that $\phi(x')$ is $[x'_1, x'_2, {x'_1}^2, \sqrt(2)x'_1x'_2, {x'_2}^2]$)

This particular ϕ transformation allows to compute the kernel function very cheaply:

$k(x, x'; \phi) = \phi(x) \cdot \phi(x')$

$= x_1x'_1 + x_2x'_2 + {x_1}^2{x'_1}^2 + 2x_1x'_1x_2x'_2 + {x_2}^2{x'_2}^2$

$= (x_1x'_1 + x_2x'_2) + (x_1x'_1 + x_2x'_2)^2$

$= x \cdot x' + (x \cdot x')^2$

Note that even if the transformed feature vectors have 5 dimensions, the kernel function return a scalar. In general, for this kind of feature transformation function ϕ, the kernel function evaluates as $k(x, x'; \phi) = \phi(x) \cdot \phi(x') = (1 + x \cdot x')^p$, where p is the order of the polynomial transformation ϕ.

However, it is only for *some* ϕ for which the evaluation of the kernel function becomes so nice! As soon we can prove that a particular kernel function can be expressed as the dot product of two particular feature transformations (for those interested the *Mercer's theorem* stated in these notes) the kernel function is *valid* and we don't actually need to construct the transformed feature vector (the output of ϕ).

Now our task will be to turn a linear method that previously operated on $\phi(x)$, like $\text{sign}(\theta \cdot \phi(x) + \theta_0)$ to an inter-classifier that only depends on those inner products, that operates in terms of kernels.

And we'll do that in the context of kernel perception just for simplicity. But it applies to any linear method that we've already learned.

6.5. The Kernel Perceptron Algorithm

Let's show how we can use the kernel function in place of the feature vectors in the perceptron algorithm.

Recall that the perceptron algoritm was (excluding for simplicity θ_0):

```
θ = 0               # initialisation
for t in 1:T
  for i in 1:n
    if yⁱ θ·Φ(xⁱ) ≦ 0   # checking if sign is the same, i.e. data is on the right side of its label
      θ = θ + yⁱΦ(xⁱ)   # update θ if mistake
```

Which is the final value of the parameter θ resulting from such updates ? We can write it as

$\theta^* = \sum_{j=1}^{n} \alpha^{(j)} y^{(j)} \phi(x^{(j)})$

where α is the vector of number of mistakes (and hence updates) underwent for each data pair (so $\alpha^{(j)}$ is the (scalar) number of errors occurred with the *j*-th data pair).

Note that we can interpret α^j in terms of the relative importance of the *j*-th training example to the final predictor. Because we are doing perceptron, the importance is just in terms of the number of mistakes that we make on that particular example.

When we want to make a prediction of a data pair $(x^{(i)}, y^{(i)})$ using the resulting parameter value θ^* (that is the "optimal" parameter the perceptron algorithm can give us), we take an inner product with that:

$\text{prediction}^{(i)} = \theta^* \cdot \phi(x^{(i)})$

We can rewrite the above equation as :

$\theta^* \cdot \phi(x^{(i)}) = [\sum_{j=1}^{n} \alpha^{(j)} y^{(j)} \phi(x^{(j)})] \cdot \phi(x^{(i)})$

$= \sum_{j=1}^{n} [\alpha^{(j)} y^{(j)} \phi(x^{(j)}) \cdot \phi(x^{(i)})]$

$= \sum_{j=1}^{n} \alpha^{(j)} y^{(j)} k(x^{(j)}, x^{(i)})$

But this means we can now express success or errors in terms of the α vector and a valid kernel function (typically something cheap to compute) !

An error on the data pair $(x^{(i)}, y^{(i)})$ can then be expressed as $y^{(i)} * \sum_{j=1}^{n} \alpha^{(j)} y^{(j)} k(x^{(j)}, x^{(i)}) \leq 0$. We can then base our perceptron algorithm on this check, where we start with initiating the error vector α to zero, and we run trought the data set checking for errors and, if found, updating the corresponding error term. I practice, our endogenous variable to minimise the errors is no longer directly theta, but became the α vector, that as said implicitly gives the contribution of each data pair to the θ parameter. The perceptron algorithm becomes hence the **kernel perceptron algorithm**:

```
α = 0               # initialisation of the vector
for t in 1:T
  for i in 1:n
    if yⁱ ∑ⱼ[αʲyʲk(xʲ,xⁱ)] ≦ 0   # checking if prediction is right
      αⁱ += 1  # update αⁱ if mistake
```

When we have run the algorithm and found the optimal α^* we can either:

- immediately retrieve the optimal θ^* by the above equation and make

predictions using $\hat{y}^{(i)} = \theta^* \cdot \phi(x^{(i)})$
- entirely skip $\phi(x^{(i)})$ making predictions on $x^{(i)}$ just using $\hat{y}^{(i)} = \sum_{j=1}^{n} \alpha^{(j)} y^{(j)} k(x^{(j)}, x^{(i)})$

Which method to use depends on which one is easier to compute, noting that when $\phi(x)$ has infinite dimensions we are forced to use the second one.

6.6. Kernel Composition Rules

Now instead of directly constructing feature vectors by adding coordinates and then taking it in the product and seeing how it collapses into a kernel, we can construct kernels directly from simpler kernels by made of the following **kernel composition rules**:

1. $K(x, x') = 1$ is a valid kernel whose feature representation is $\phi(x) = 1$;
2. Given a function $f : \mathbb{R}^d \to \mathbb{R}$ and a valid kernel function $K(x, x')$ whose feature representation is $\phi(x)$, then $\tilde{K}(x, x') = f(x)K(x, x')f(x')$ is also a valid kernel whose feature representation is $\tilde{\phi}(x) = f(x)\phi(x)$
3. Given $K_a(x, x')$ and $K_b(x, x')$ being two valid kernels whose feature representations are respectively $\phi_a(x)$ and $\phi_b(x)$, then $K(x, x') = K_a(x, x') + K_b(x, x')$ is also a valid kernel whose feature representation is $\phi(x) = \begin{bmatrix} \phi_a(x) \\ \phi_b(x) \end{bmatrix}$
4. Given $K_a(x, x')$ and $K_b(x, x')$ being two valid kernels whose feature representations are respectively $\phi_a(x) \in \mathbb{R}^A$ and $\phi_b(x) \in \mathbb{R}^B$, then $K(x, x') = K_a(x, x') * K_b(x, x')$ is also a valid kernel whose feature representation is $\phi(x) = \begin{bmatrix} \phi_{a,1}(x) * \phi_{b,1}(x) \\ \phi_{a,1}(x) * \phi_{b,2}(x) \\ \phi_{a,1}(x) * \phi_{b,\dots}(x) \\ \phi_{a,1}(x) * \phi_{b,B}(x) \\ \phi_{a,2}(x) * \phi_{b,1}(x) \\ \phi_{a,\dots}(x) * \phi_{b,\dots}(x) \\ \phi_{a,A}(x) * \phi_{b,B}(x) \end{bmatrix}$ (see this lecture notes for a proof)

Armed with these rules we can build up pretty complex kernels starting from simpler ones.

For example let's start with the identity function as ϕ, i.e. $\phi_a(x) = x$. Such feature function results in a kernel $K(x, x'; \phi_a) = K_a(x, x') = (x \cdot x')$ (this is known as the **linear kernel**).

We can now add to it a squared term to form a new kernel, that by virtue of rules (3) and (4) above is still a valid kernel:

$$K(x, x') = K_a(x, x') + K_a(x, x') * K_a(x, x') = (x \cdot x') + (x \cdot x')^2$$

6.7. The Radial Basis Kernel

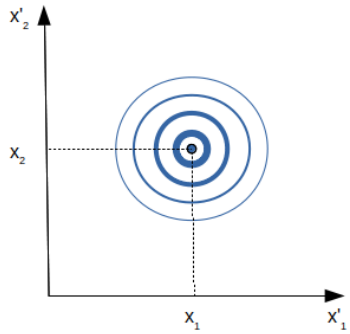
We can use kernel functions, and have them in term of simply, cheap-to-evaluate functions, even when the underlying feature representation would have infinite dimensions and would be hence impossible to explicitly construct.

One example is the so called **radial basis kernel**:

$$K(x, x') = e^{-\frac{1}{2}||x-x'||^2}$$

It can be proved that suck kernel is indeed a valid kernel and its corresponding feature representation $\phi(x) \in \mathbb{R}^{\infty}$, i.e. involves polynomial features up to an infinite order.

Does the radial basis kernel look like a Gaussian (without the normalisation term) ? Well, because indeed it is:



The above picture shows the contour lines of the radial basis kernel when we keep fixed x (in 2 dimensions) and we let x' to move away from it: the value of the kernel then reduces in a shape that in 3-d would resemble the classical bell shape of the Gaussian curve. We could even parametrise the radial basis kernel replacing the fixed $1/2$ term with a parameter γ that would determine the width of the bell-shaped curve (the larger the value of γ the narrower will be the bell, i.e. small values of γ yield wide bells).

Because the feature has infinite dimensions, the radial basis kernel has infinite expressive power and can correctly classify any training test.

The linear decision boundary in the infinite dimensional space is given by the set $\{x : \sum_{j=1}^{n} \alpha^{(j)} y^{(j)} k(x^{(j)}, x) = 0\}$ and corresponds to a (possibly) non-linear boundary in the original feature vector space.

The more difficult task it is, the more iterations before this kernel perception (with the radial basis kernel) will find the separating solution, but it always will in a finite number of times. This is by contrast with the “normal” perceptron algorithm that when the set is not separable would continue to run at the infinite, changing its parameters unless it is stopped at a certain arbitrary point.

Other non-linear classifiers

We have seen as we can have nonlinear classifiers extending to higher dimensional space and evenutally using kernel methods to collapse the calculations and operate only *implicitly* in those high dimension spaces.

There are certainly other ways to get nonlinear classifiers.

Decision trees make classification operating sequentially on the various dimensions and making first a separation on the first dimension and then, in a subsequent step, on the second dimension and so on. And you can "learn" these trees incrementally.

There is a way to make these decision trees more robust, called **random forest classifiers**, that adds two type of randomness: the first one is in randomly choosing the dimension on which to operate the cut, the second in randomly selecting the single example on which operate from the data set (with replacement) and then just average the predictions obtained from these trees.

So the procedure of a random forest classifier is:

- boostrap the sample
- build a randomized (by dimension) decision tree
- average the predictions (ensemble)

Summary

- We can get non-linear classifiers (or regression) methods by simply mapping our data into new feature vectors that include non-linear components, and applying a linear method on these resulting vectors;
- These feature vectors can be high dimensional, however;
- We can turn linear methods into kernel methods by casting the computation in terms of inner products;
- A kernel function is advantageous when the inner products are faster to evaluate than using explicit feature vectors (e.g. when the vectors would be infinite dimensional!)
- We saw the radial basis kernel that is particularly powerful because it is both (a) cheap to evaluate and (b) has a corresponding infinite dimensional feature vector

Lecture 7. Recommender Systems

7.1. Objectives

At the end of this lecture, you will be able to

- understand the problem definition and assumptions of recommender systems
- understand the impact of similarity measures in the K-Nearest Neighbor method
- understand the need to impose the low rank assumption in collaborative filtering
- iteratively find values of U and V (given $X = UV^T$) in collaborative filtering

7.2. Introduction

This lesson deals about recommender systems, when the algorithm try to guess preferences based on choices already made by the user (like film to watch or

products to buy).

We'll see:

- the exact problem definition
- the historically used algorithm, the K-Nearest Neighbor algorithm (KNN);
- the algorithm in use today, the matrix factorization or collaborative filtering.

Problem definition

We keep as example across the lecture the recommendation of movies.

We start with a $(n \times m)$ matrix Y of preferences for user $a = 1, \dots, n$ of movie $i = 1, \dots, m$. While there are many ways to store preferences, we will use real numbers.

The goal is to base the prediction on the prior choices of the users, considering that this Y matrix could be very sparse (e.g. out of 18000 films, each individual ranked very few of them!), i.e. we want to fill these "empty spaces" of the matrix.

Why not to use classification/regression based on feature vectors as learned in Lectures 1? For two reasons:

1. Deciding which feature to use or extracting them from data could be hard/infeasible (e.g. where can I get the info if a film has a happy or bad ending ?), or the feature vector could become very very large (things about in general "products" for Amazon, could be everything)
2. Often we have little data about a single users preferences, while to make a recommendation based on its own previous choices we would need lot of data.

The "trick" is then to "borrow" preferences from the other users and trying to measure how much a single user is closer to the other ones in our dataset.

7.3. K-Nearest Neighbor Method

The number K here means, how big should be your advisory pool on how many neighbors you want to look at. And this can be one of the hyperparameters of the algorithm.

We look at the k closest users that did score the element I am interested to, look at their score for it, and average their score.

$$\hat{Y}_{a,i} = \frac{\sum_{b \in KNN(a,i;K)} Y_{b,i}}{K}$$

where $KNN(a, i; K)$ defines the set of K users closer to the user a that have a score for item i.

Now, the question is of course how do I define this similarity? We can use any method to define similarity between vectors, like cosine similarity

($\cos\theta = \frac{x_a \cdot x_b}{\|x_a\|\|x_b\|}$) or Euclidean distance ($\|x_a - x_b\|$).

We can make the algorithm a bit more sophisticated by weighting the neighbour scores to the level of similarity rather than just take their unweighted average:

$$\hat{Y}_{a,i} = \frac{\sum_{b \in KNN(a,i;K)} sim(a,b) * Y_{b,i}}{\sum_{b \in KNN(a,i;K)} |sim(a,b)|}$$

where $sim(a, b)$ is some similarity measure between users a and b.

There has been many improvements that has been added to this kind of algorithm, like adjusting for the different "average" score that each user gives to the items (i.e. they compare the deviations from user's averages rather than the raw score itself).

Still they are very far from today's methods. The problem of KNN is that it doesn't enable us to detect the hidden structures that is there in the data, which is that users may be similar to some pool of other users in one dimension, but similar to some other set of users in a different dimension. For example, loving machine learning books, and having there some "similarity" with other readers of machine learning books on some hidden characteristics (e.g. liking equation-rich books or more discursive ones), and plant books, where the similarity with other plant-reading users would be based on completely different hidden features (e.g. loving photos or having nice tabular descriptions of plants).

Conversely in collaborative filtering, the algorithm would be able to detect these hidden groupings among the users, both in terms of products and in terms of users. So we don't have to explicitly engineer very sophisticated similarity measure, the algorithm would be able to pick up these very complex dependencies that for us, as humans, would be definitely not tractable to come up with.

7.4. Collaborative Filtering: the Naive Approach

Let's start with a *naive* approach where we just try to apply the same method we used in regression to this problem, i.e. minimise a function J made of a distance between the observed score in the matrix and the estimated one and a regularisation term.

For now, we treat each individual score independently… and this will be the reason for which (we will see) this method will not work.

So, we have our (sparse) matrix Y and we want to find a dense matrix X that is able to replicate at best the observed points of $Y_{a,i}$ when these are available, and fill the missing ones when $Y_{a,i} = missing$.

Let's first define as D the set of points for which a score in Y is given: $D = \{(a, i) : Y_{a,i} \neq \text{missing}\}$.

The J function then takes any possible X matrix and minimise the distance between the points in the D set less a regularisation parameter (we keep the individual scores to zero unless we have strong belief to move them from such

state):

$$J(X;Y,\lambda) = \frac{\sum_{(a,i)\in D}(Y_{a,i}-X_{a,i})^2}{2} + \frac{\lambda}{2}\sum_{(a,i)} X_{a,i}^2$$

To find the optimal $X^*_{a,i}$ that minimise the FOC $(\partial X_{a,i}/\partial Y_{a,i}) = 0$ we have to distinguish if (a,i) is in D or not:

- $(a,i) \in D$: $X^*_{a,i} = \frac{Y_(a,i)}{1+\lambda}$
- $(a,i) \notin D$: $X^*_{a,i} = 0$

Clearly this result doesn't make sense: for data we already know we obtain a bad estimation (as worst as we increase lambda) and for unknown scores we are left with zeros.

7.5. Collaborative Filtering with Matrix Factorization

What we need to do is to actually relate scores together instead of considering them independently.

The idea is then to constrain the matrix X to have a lower rank, as rank captures how much independence is present between the entries of the matrix.

At one extreme, constraining the matrix to be rank 1, would means that we could factorise the matrix X as just the matrix product of two single vectors, one defining a sort of general sentiment about the items for each user (u), and the other one (v) representing the average sentiment for a given item, i.e. $X = uv^T$.

But representing users and items with just a single number takes us back to the KNN problem of not being able to distinguish the possible multiple groups hidden in each user or in each item.

We could then decide to divide the users and/or the items in respectively $(n \times 2)U$ and $(2 \times m)V^T$ matrices and constrain our X matrix to be a product of these two matrices (hence with rank 2 in this case): $X = UV^T$

The exact numbers K of vectors to use in the user/items factorisation matrices (i.e. the rank of X) is then a hyperparameter that can be selected using the validation set.

Still for simplicity, in this lesson we will see the simplest case of constraining the matrix to be factorisable by a pair of single vectors (i.e. $K = 1$).

7.6. Alternating Minimization

Using rank 1, we can adapt the J function to take the two vectors u and v instead of the whole X matrix, and our objective becomes to find their elements that minimise such function:

$$J(\mathbf{u},\mathbf{v};Y,\lambda) = \frac{\sum_{a,i\in D}(Y_{a,i}-u_a * v_i)^2}{2} + \frac{\lambda}{2}\sum_a^n u_a^2 + \frac{\lambda}{2}\sum_i^m v_i^2$$

How do we minimise J ? We can take an iterative approach where we start by randomly sampling values for one of the vector and minimise for the other vector (by setting the derivatives with respect on its elements equal to zero), then fix this second vector and going minimise for the first one, etc., until the value of the function J doesn't move behind a certain threshold, in an alternating minimisation exercise that will guarantee us to find a local minima (but not a global one!).

Note also that when we minimise for the individual component of one of the two vectors, we obtain derivatives with respect to the individual vector elements that are independent, so the first order condition can be expressed each time in terms of a single variable.

Numerical example

Let's consider a value of λ equal to 1 and the following score dataset:

$$Y = \begin{bmatrix} 5 & ? & 7 \\ 1 & 2 & ? \end{bmatrix}$$

and let start out minimisation algorithm with $v = [2, 7, 8]$

L becomes :

$$J(\mathbf{u}; \mathbf{v}, Y, \lambda) = \frac{(5-2u_1)^2+(7-8u_1)^2+(1-2u_2)^2+(2-7u_2)^2}{2} + \frac{u_1^2+u_2^2}{2} + \frac{2^2+7^2+8^2}{2}$$

From where, setting $\partial L/\partial u_1 = 0$ and $\partial L/\partial u_2 = 0$ we can retrieve the minimising values of (u_1, u_2) as 22/23 and 8/27. We can now compute $J(\mathbf{v}; \mathbf{u}, Y, \lambda)$ with these values of u to retrieve the minimising values of v and so on.

Project 2: Digit recognition (Part 1)

The softmax function

Summary from: https://en.wikipedia.org/wiki/Softmax_function

The softmax function is a "normalisation function" defined as:

$$\text{Softmax}(\mathbf{Z} \in \mathbb{R}^K) \in \mathbb{R}^{+K} = \frac{1}{\sum_{j=1}^{k} e^{Z_j}} * e^{\mathbf{Z}}$$

- it maps a vector in $\mathbb{R}^K$ to a new vector in $\mathbb{R}^{+K}$ where all values are positive and sum up to 1, hence loosing one degree of freedom and with the output interpretable as probabilities
- the larger (in relative term) is the input Z_K, the larger will be its output probability
- note that, for each K, the map $Z_K \rightarrow P(Z_K)$ is continuous
- it can be seen as a smooth approximation to the arg max function: the function whose value is which index has the maximum

The softmax function can be used in multinomial logistic regression to represent the predicted probability for the j'th class given a sample vector $\mathbf{x}$ and a weighting vector $\mathbf{w}$:

$$P(y = j \mid \mathbf{x}) = \frac{e^{\mathbf{x}^\mathsf{T}\mathbf{w}_j}}{\sum_{k=1}^{K} e^{\mathbf{x}^\mathsf{T}\mathbf{w}_k}}$$

It can be parametrised by a parameter τ referred as "temperature" in allusion to statical mechanics:

$$\text{softmax}(\mathbf{Z}; \tau \in \mathbb{R}) = \frac{1}{\sum_{j=1}^{k} e^{Z_j/\tau}} * e^{\mathbf{Z}/\tau}$$

- For high temperatures ($\tau \to \infty$), all actions have nearly the same probability and the lower the temperature, the more expected rewards affect the probability.
- For a low temperature ($\tau \to 0^+$), the probability of the action with the highest expected reward tends to 1.

[MITx 6.86x Notes Index]

[MITx 6.86x Notes Index]

Unit 03 - Neural networks

Lecture 8. Introduction to Feedforward Neural Networks

8.1. Unit 3 Overview

This unit deals with Neural Networks, powerful tools that have many different usages as self-driving cars or playing chess.

Neural networks are really composed of very simple units that are akin to linear classification or regression methods.

We will consider:

- feed-forward neural networks (simpler)
- recurrent neural networks: can model sequences and map sequences to class labels and even to other sequences (e.g. in machine translation)
- convolutional neural network: a specific architecture designed for processing images (used in the second part of project 2 for image classification)

At the end of this **unit**, you will be able to

- Implement a feedforward neural networks from scratch to perform image classification task.
- Write down the gradient of the loss function with respect to the weight parameters using back-propagation algorithm and use SGD to train neural networks.
- Understand that Recurrent Neural Networks (RNNs) and long short-term memory (LSTM) can be applied in modeling and generating sequences.
- Implement a Convolutional neural networks (CNNs) with machine learning packages.

8.2. Objectives

At the end of this **lecture**, you will be able to

- Recognize different layers in a feedforward neural network and the number of units in each layer.
- Write down common activation functions such as the hyperbolic tangent function `tanh`, and the rectified linear function `ReLU`.
- Compute the output of a simple neural network possibly with hidden layers given the weights and activation functions .
- Determine whether data after transformation by some layers is linearly separable, draw decision boundaries given by the weight vectors and use them to help understand the behavior of the network.

8.3. Motivation

The topic of feedforward neural networks is split into two parts:

- part one (this lesson): the model
  - motivations (what they bring beyond non-linear classification methods we have already seen);
  - what they are, and what they can capture;
  - the power of hidden layers.
- part two (next lesson): learning from data: how to use simple stochastic gradient descent algorithms as a successful way of actually learning these complicated models from data.

Neural networks vs the non-linear classification methods that we saw already

Let's consider a linear classifier $\hat{y} = sign(\theta \cdot \phi(\mathbf{x}))$.

We can interpret the various dimensions of the original feature vector $\mathbf{x} \in \mathbb{R}^d$ as d nodes. Then these nodes are mapped (non necessairly in a linear way) to D nodes of the feature representation of $\phi(\mathbf{x} \in \mathbb{R}^D)$. Finally we take a linear combination of these nodes (dimensions), we apply our sign function and we obtain the classification.

The key difference between neural networks and the methods we saw in unit 2 is that, there, the mapping from x to $\phi(x)$ in not part of the data analysis, it is done ex-ante (even, although implicitly, with the choice of a particular kernel), and

then we optimise once we chose ϕ.

In neural network instead the choice of the ϕ mapping is endogenous to the learning step, together with the choice of θ.

Note that we have a bit of a chicken and egg problem here, as, in order to do a good classification - understand and learn the parameters theta for the classification decision - we would need to know what that feature representation is. But, on the other hand, in order to understand what a good feature representation would be, we would need to know how that feature representation is exercised in the ultimate classification task.

So we need to learn ϕ and θ jointly, trying to adjust the feature representation together with how it is exercised towards the classification task.

Motivation to Neural Networks (an other summary):

So far, the ways we have performed non-linear classification involve either first mapping x explicitly into some feature vectors $\phi(x)$, whose coordinates involve non-linear functions of x, or in order to increase computational efficiency, rewriting the decision rule in terms of a chosen kernel, i.e. the dot product of feature vectors, and then using the training data to learn a transformed classification parameter.

However, in both cases, the feature vectors are chosen. They are not learned in order to improve performance of the classification problem at hand.

Neural networks, on the other hand, are models in which the feature representation is learned jointly with the classifier to improve classification performance.

8.4. Neural Network Units

Real neurons:

- take a signal in input using dendroids that connect the neuron to ~ 10^3 _ 10^4 other neurons
- the signal potential increases in the neuron's body
- once a threshold is reached, the neuron in turn emits a signal that, through its axon, reaches ~ 100 other neurons

Artificial neural networks:

- the parameter associated to each nodes of a layer (input coordinates) takes the role of weights trying to mimic the strength of the connection that propagates to the cell body;
- the response of a cell is in terms of a nonlinear transformation of the aggregated, weighted inputs, resulting in a single real number.

A bit of terminology:

- The individual coordinates are known as **nodes** or **units**.

- The **weights** are denoted with w instead of θ as we were used to.
- Each node's **aggregated input** is given by $z = \sum_{i=1}^{d} x_i w_i + w_0$ (or, in vector form, $z = \mathbf{x} \cdot \mathbf{w} + w_0$, with $z \in \mathbb{R}, \mathbf{x} \in \mathbb{R}^d, \mathbf{w} \in \mathbb{R}^d$)
- The output of the neuron is the result of a non-linear transformation of the aggregated input called **activation function** $f = f(z)$
- A **neural network unit** is a primitive neural network that consists of only the "input layer", and an output layer with only one output.

Common kind of activation functions:

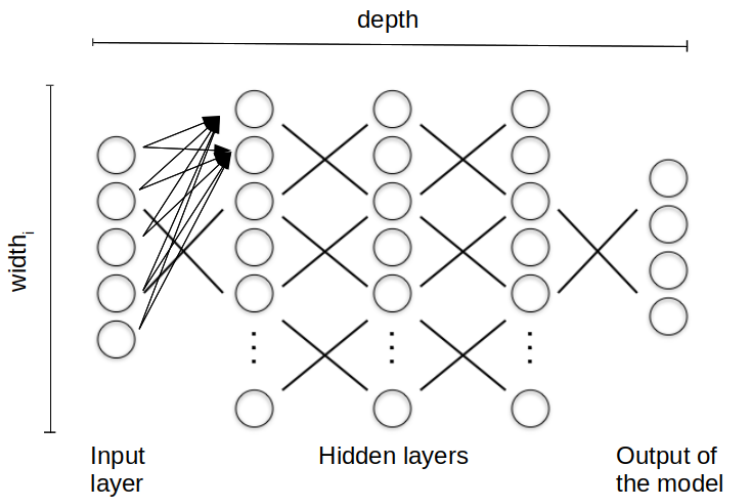
- linear. Used typically at the very end, before measuring the loss of the predictor;
- relu ("rectified linear unit"): $f(z) = max\{0, z\}$
- tanh ("hyperbolic tangent"): it mimics the sine function but in a soft way: it is a sigmoid curve spanning from -1 (at $z = -\infty$) to +1 (at $z = +\infty$), with a value of 0 at $z = 0$. Its smoothness property is useful since we have to propagate the training signal through the network in order to adjust all the parameters in the model. $tanh(z) = \frac{e^z - e^{-z}}{e^z + e^{-z}} = 1 - \frac{2}{e^{2z}+1}$. Note that $tanh(-z) = -tanh(z)$.

Our objective will be to learn the weights that make this node, in the context of the whole network, to then function appropriately, to behave well.

8.5. Introduction to Deep Neural Networks

Overall architecture

In **deep forward neural networks**, neural network units are arranged in **layers**, from the *input layer*, where each unit holds the input coordinate, through various *hidden layer* transformations, until the actual *output* of the model:



In this layerwise computation, each unit in a particular layer takes input from *all* the preceding layer units. And it has its own parameters that are adjusted to perform the overall computation. So parameters are different even between different units of the same layer. A deep (feedforward) neural network refers hence to a neural network that contains not only the input and output layers, but also hidden layers in between.

- **Width** (*of the layer*): number of units in that specific layer
- **Depth** (*of the architecture*): number of layers of the overall transformation before arriving to the final output

Deep neural networks

- loosely motivated by biological neurons, networks
- adjustable processing units (~ linear classifiers)
- **highly parallel** (important!), typically organized in layers
- deep = many transformations (layers) before output

For example the input could be an image, so the input vector is the individual pixel content, from which the first layer try to detect edges, then these are recombined into parts (in subsequent layers), objects and finally characterisation of a scene: edges -> simple parts-> parts -> objects -> scenes

One of the main advantages of deep neural networks is that in many cases, they can learn to extract very complex and sophisticated features from just the raw features presented to them as their input. For instance, in the context of image recognition, neural networks can extract the features that differentiate a cat from a dog based only on the raw pixel data presented to them from images.

The initial few layers of a neural networks typically capture the simpler and smaller features whereas the later layers use information from these low-level features to identify more complex and sophisticated features.

Note: it is interesting to note that a neural network can represent any given binary function.

Subject areas

Deep learning has overtaken a number of academic disciplines in just a few years:

- computer vision (e.g., image, scene analysis)
- natural language processing (e.g., machine translation)
- speech recognition
- computational biology, etc.

Key role in recent successes in corporate applications such as:

- self driving vehicles
- speech interfaces
- conversational agents, assistants (Alexa, Cortana, Siri, Google assistant,…)
- superhuman game playing

Many more underway (to perform prediction and/or control):

- personalized/automated medicine
- chemistry, robotics, materials science, etc.

Deep learning … why now?

1. Reason #1: lots of data

- many significant problems can only be solved at scale
- lots of data enable us to model very complex rich models solving more realistic tasks;

2. Reason #2: parallel kid computational resources (esp. GPUs, tensor processing units)

- platforms/systems that support running deep (machine) learning algorithms at scale in parallel

3. Reason #3: large models are actually easier to train

- contrary to small, rigid models, large, richer and flexible models can be successfully estimated even with simple gradient based learning algorithms like stochastic gradient descent.

4. Reason #4: flexible neural “lego pieces”

- common representations, diversity of architectural choices
- we can easily compose these models together to perform very interesting computations. In other words, they can serve as very flexible computational Lego pieces that can be adapted overall to perform a useful role as part of much larger, richer computational architectures.

Why #3 (large models are easier to learn) ?

We can think about the notions of width (number of units in a layer) and depth (number of layers). Small models (low width and low depth) are quite rigid and don’t allow for a good abstraction of reality i.e. learning the underlying structures based on observations. Large models can use more width and more depth to generate improved abstractions of reality i.e. improved learning.

See also the conclusion of the video on the next segment: “Introducing redundancy will make the optimization problem that we have to solve easier.”

8.6. Hidden Layer Models

Let’s now specifically consist a deep neural network consisting of the input layer $\mathbf{x}$, a single hidden layer $\mathbf{z}$ performing the aggregation $z_i = \sum_j x_j * w_{j,i} + w_{0,i}$ and using $tanh(\mathbf{z})$ as activation function f, and the output node with a linear activation function.

We can see each layer (one in this case) as a “box” that takes as input $\mathbf{x}$ and return output $\mathbf{f}$, mediated trough its weights $\mathbf{W}$, and the output layer as those box taking $\mathbf{f}$ as input and returning the final output f, mediated trough its weights $\mathbf{W}'$

And we are going to try to understand how this computation changes as a function of W and W'.

What these hidden units are actually doing ? Since they are like linear classifiers, they take linear combination of the inputs, pass through that nonlinear function, we can also visualize them as if they were linear classifiers with norm equal to $\mathbf{w}$.

The difference is that instead of having a binary output (like in $sign(\mathbf{w} \cdot \mathbf{x})$) we have now $f(\mathbf{w} \cdot \mathbf{x})$. f typically takes the form of $tanh(\mathbf{w} \cdot \mathbf{x})$ for the hidden layers that we will study, whose output range is (-1,+1). More the point is far from the boundary, more $tanh()$ move toward the extremes, with a speed that is proportional to the norm of $\mathbf{w}$.

If we have (as it is normally) multiple nodes per layer, we can thing on a series of linear classifiers on the same $depth_{i-1}$ space, each one identified by its norm $\mathbf{w}_1, \mathbf{w}_2, \ldots, \mathbf{w}_{depth_i}$.

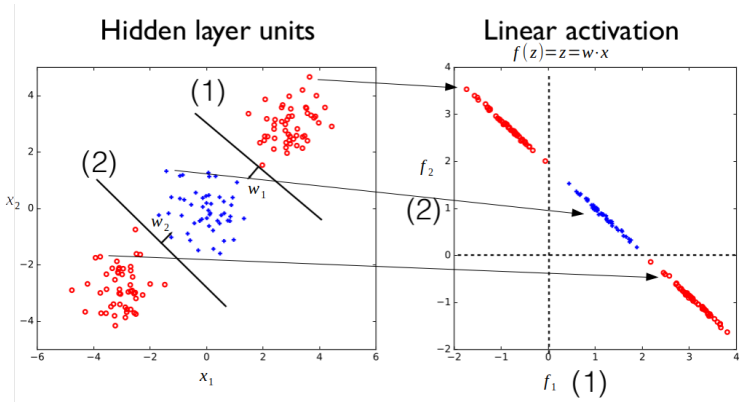
Given a neural network with one hidden layer for classification, we can view the hidden layer as a feature representation, and the output layer as a classifier using the learned feature representation.

There're also other parameters that will affect the learning process and the performance of the model, such as the learning rate and parameters that control the network architecture (e.g. number of hidden units/layers) etc. These are often called hyper-parameters.

Similar to the linear classifiers that we covered in previous lectures, we need to learn the parameters for the classifier. However, in this case we also learn the parameters that generate a representation for the data. The dimensions and the hyper-parameters are decided with the structure of the model and are not optimized directly during the learning process but can be chosen by performing a grid search with the evaluation data or by more advanced techniques (such as meta-learning).

2-D Example

Let's consider as example a case in 2-D where we have a cloud of negative points (red) in the bottom-left and top-right corners and a cloud of positive points (blue) in the center, like in the following chart (left side):



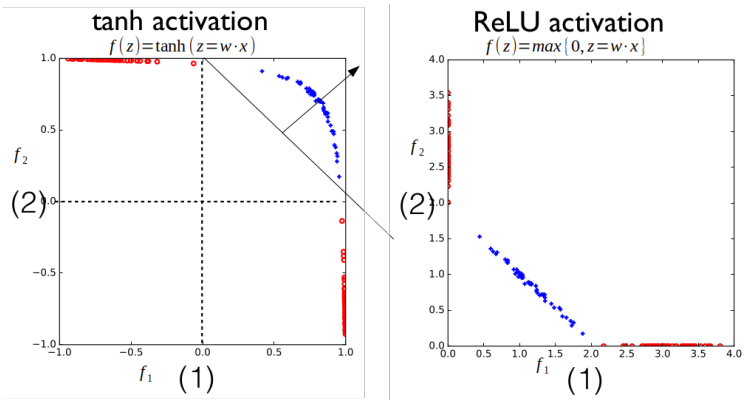
Such problem is clearly non linearly separable.

The chart on the right depicts the same points in the space resulting from the

application of the two classifiers, using just a linear activation, i.e. the dot product between w and x.

The appearance that it draws exactly as a line derives from the fact that the two planes are parallel, but the general idea is that still we have a problem that is not separable, as any linear transformation of the feature space of a linearly inseparable classification problem would still continue to remain linearly inseparable.

However, when we use $tanh(x)$ as activation function we obtain the output as depicted in the following chart (left), where the problem now is clearly linearly separable.



This is really where the power of the hidden layer lies. It gives us a transformation of the input signal into a representation that makes the problem easier to solve.

Finally we can try the ReLU activation ($f(z) = max\{0, z\}$): in this case the output is not actually strictly linearly separable.

So this highlights the difficulty of learning these models. It's not always the case that the same non-linear transformation casts them as linearly separable.

However if we flip the planes directions, both the tanh and the ReLU activation results in outputs that become linearly separable (for the ReLU, all positive points got mapped to the (0,0) point).

What if we chose the two planes as random (i.e. $\mathbf{w}_1$ and $\mathbf{w}_2$ are random) ? The resulting output would likely not be linearly separable. However if we introduce redundancy, e.g. using 10 hidden units for an original two dimensional problem, the problem would likely become linearly separable even if these 10 planes are chosen at random. Notice this is quite similar to the systematic expansion that we did earlier, in terms of polynomial features.

So introducing redundancy here is actually helpful. And we'll see how this is helpful also when we are actually learning these hidden unit representations from data. Introducing redundancy will make the optimization problem that we have to solve easier.

Summary

- Units in neural networks are linear classifiers, just with different output non-

linearity
- The units in feed-forward neural networks are arranged in layers (input, one or plus hidden, output)
- By learning the parameters associated with the hidden layer units, we learn how to represent examples (as hidden layer activations)
- The representations in neural networks are learned directly to facilitate the end-to-end task
- A simple classifier (output unit) suffices to solve complex classification tasks if it operates on the hidden layer representations

The outward layer is their prediction that we actually want. And the role of the hidden layers is really to adjust their transformation, adjust their computation in such a way that the output layer will have an easier task to solve the problem.

The next lecture will deal with actually learning these representations together with the final classifier.

Lecture 9. Feedforward Neural Networks, Back Propagation, and Stochastic Gradient Descent (SGD)

9.1. Objectives

At the end of this lecture, you will be able to

- Write down recursive relations with back-propagation algorithm to compute the gradient of the loss function with respect to the weight parameters.
- Use the stochastic descent algorithm to train a feedforward neural network.
- Understand that it is not guaranteed to reach global (only local) optimum with SGD to minimize the training loss.
- Recognize when a network has overcapacity .

9.2. Back-propagation Algorithm

We start now to consider how to learn from data (feedforward) neural network, that is estimate its weights.

We recall that feed-forward neural networks, with multiple hidden layers mediating the calculation from the input to the output, are complicated models that are trying to capture the representation of the examples towards the output unit in such a way as to facilitate the actual prediction task.

It is this representation learning part – we're learning the feature representation as well as how to make use of it – that makes the learning problem difficult. But it turns out that a simple stochastic gradient descent algorithm actually succeeds in finding typically a good solution to the parameters, provided that we give the model a little bit of overcapacity. The main algorithmic question, then, is how to actually evaluate that gradient, the derivative of the Loss with respect to the parameters. And that can be computed efficiently using so-called back propagation algorithm.

Given an input X, a neural network characterised by the overall weight set W (so that its output, a scalar here for simplicity, is $f(X;W)$), and the "correct"

target vector Y, the task in the training step is find the W that minimise a given loss function $\mathcal{L}(f(X;W),Y)$. We do that by computing the derivative of the loss function for each weight $w_{i,j}^l$ applied by the j-th unit at each l-th layer in relation to the output of the i-th node at the previous $l-1$ layer: $\frac{\partial \mathcal{L}(f(X;W),Y)}{\partial w_{i,j}}$.

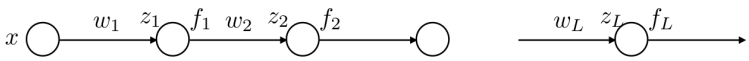
Then we simply apply the SDG algorithm in relation to this $w_{i,j}^l$:

$$w_{i,j}^l \leftarrow w_{i,j}^l - \eta * \frac{\partial \mathcal{L}(f(X;W),Y)}{\partial w_{i,j}}$$

The question turns now on how do we evaluate such gradient, as the mapping from the weight to the final output of the network can be very complicated (so much for the weights in the first layers!).

This computation can be efficiently done using a so called **back propagation algorithm** that essentially exploits the chain rule.

Let's take as example a deep neural network with a single unit per node, with both input and outputs as scalars, as in the following diagram:



Let's also assume that the activation function is $\tanh(z)$, also in the last layer (in reality the last unit is often a linear function, so that the prediction is in $\mathbb{R}$ and not just in $(-1,+1)$), that there is no offset parameter and that the specific loss function for each individual example is $Loss = \frac{1}{2}(y-f_L)^2$.

Then, for such network, we can write $z_1 = xw_1$ and, more in general, for $i = 2,\ldots,L$: $z_i = f_{i-1}w_i$ where $f_{i-1} = f(z_{i-1})$.

So, specifically, $f_1 = tanh(z_1) = tanh(xw_1)$, $f_2 = tanh(z_2) = tanh(f_1 w_2)$,...

In order to find $\frac{\partial Loss}{\partial w_i}$ we can use the chain rule by start evaluating the derivative of the loss function, then evaluate the last layer, and then eventually go backward until the first layer:

$$\frac{\partial Loss}{\partial w_1} = \frac{\partial f_1}{\partial w_1} * \frac{\partial f_2}{\partial f_1} * \frac{\partial f_3}{\partial f_2} * \ldots * \frac{\partial f_L}{\partial f_{L-1}} * \frac{\partial Loss}{\partial f_L}$$

$$\frac{\partial Loss}{\partial w_1} = [(1-tanh^2(xw_1))x] * [(1-tanh^2(f_1 w_2))w_2] * [(1-tanh^2(f_2 w_3))w_3] * \ldots [(1$$

Now based on the nature the calculator, the fact that we evaluate the loss at the very output, then multiply by these Jacobians also highlights how this can go wrong. Imagine if these Jacobians here, the value is the derivatives of the layer-wise mappings, are very small. Then the gradient vanishes very quickly as the depth of the architecture increases. If these derivatives are large, then the gradients can also explode.

So there are issues that we need to deal with when the architecture is deep.

9.3. Training Models with 1 Hidden Layer, Overcapacity, and Convergence Guarantees

Using the SGD, the average hinge loss evolves at each iterations (or epochs), where the algorithm runs through all the training examples (in sample order), performing a stochastic gradient descent update Typically, after a few runs over the training examples, the network actually succeeds in finding a solution that has zero hinge loss.

When the problem is however complex, a neural network with just the capacity in terms of number of nodes that would be theoretically enough to find a separable solution (classify all the example correctly) may not actually arrive to such optimal classification. More in general, for multi-layer neural networks, stochastic gradient descent (SGD) is not guaranteed to reach a global optimum (but they can find a locally optimal solution, which is typically quite good).

We can facilitate the optimization of these architectures by giving them **overcapacity** (increasing the number of nodes in the layer(s)), making them a little bit more complicated than they need to be to actually solve the task.

Using overcapacity however can lead to artefacts in the classifiers. To limit these artefacts and have good models even with overcapacity one can use two tricks:

1. Consider random initialisation of the parameters (rather than start from zero). And as we learn and arrive at a perfect solution in terms of end-to-end mapping from inputs to outputs, then in that solution, not all the hidden units need to be doing something useful, so long as many of them do. The randomization inherent in the initialization creates some smoothness. And we end up with a smooth decision boundary even when we have given the model quite a bit of overcapacity.
2. Use the ReLU activation function ($max(0, z)$) rather than $tanh(z)$. ReLU is cheap to evaluate , works well with sparsity and make parameters easier to be estimated in large models.

We will later talk about regularization – how to actually squeeze the capacity of these models a little bit, while in terms of units, giving them overcapacity.

Summary

- Neural networks can be learned with SGD similarly to linear classifiers
- The derivatives necessary for SGD can be evaluated effectively via back-propagation
- Multi-layer neural network models are complicated. We are no longer guaranteed to reach global (only local) optimum with SGD
- Larger models tend to be easier to learn because their units only need to be adjusted so that they are, collectively, sufficient to solve the task

Lecture 10. Recurrent Neural Networks 1

10.1. Objective

Introduction to recurrent neural networks (RNNs)

At the end of this lecture, you will be able to:

- Know the difference between feed-forward and recurrent neural networks(RNNs).
- Understand the role of gating and memory cells in long-short term memory (LSTM).
- Understand the process of encoding of RNNs in modeling sequences.

10.2. Introduction to Recurrent Neural Networks

In this and in the next lecture we will use neural networks to model sequences, using so called **recurrent neural networks** (*RNN*).

This lecture introduces the topic: the problem of modelling sequences, what are RNN, how they relate to the feedforward neural network we saw in the previous lectures and how to *encode* sequences into vector representations. Next lecture will focus on how to *decode* such vectors so that we can use them to predict properties of the sequences, or what comes next in the sequence.

Exchange rate example

Let's consider a time serie of the exchange rate between US Dollar and the Euro, with an objective of predict its value in the future.

We already saw how to solve this kind of problem with linear predictors or feedforward neural networks.

In both case the first task is to compile a feature vector, for example of the values of the exchange rate at various times, for example, at time $t-1$ to $t-4$ for a prediction at time t.

As we have long time-serie available we can use some of the observation as training, some as validation and some as test (ideally by random sampling).

Language completion example

In a similar way we can see a text as a sequence of words, and try to predict the next world based on the previous words, for example using the previous two words, where each of them is coded as a 1 in a sparse array of all the possible words (à la bag of words approach).

Limitations of these approaches

While we could use linear classifiers or feedforward neural networks to predict sequences we are still left with the problem of how much "history" look at in the creation of the feature vector and the fact that this history length may be variable, for example there may be words at the beginning of the sentence that are quite relevant for predicting what happens towards the end, and we would have to somehow retain that information in the feature representation that we are using for predicting what happens next.

Recurrent Neural Networks can be used in place of feedforwrd neural networks to learn not only the weight given the feature vectors, but also how to encode in the first instance the history into the feature vector.

10.3. Why we need RNNs

The way we chose how to encode data in feature vectors depends on the task we need. For example, sentiment analysis, language translation, and next word suggestion all requires a different feature representation as they focus on different parts of the sentence: while sentiment analysis focuses on the holistic meaning of a sentence, translation or next word suggestion focuses instead more on individual words.

While in feed-forward networks we have to manually engineer how history is mapped to a feature vector (representation) for the specific task at hand, using RNN's this task is also part of the learning process and hence automatised.

Note that very different types of objects can be encoded in feature vectors, like images, events, words or videos, and once they are encoded, all these different kind of objects can be used together.

In other words, RNNs can not only process single data points (such as images), but also entire sequences of data (such as speech or video).

While this lecture deals with **encoding**, the mapping of a sequence to a feature vector, the next lecture deals with **decoding**, the mapping of a feature vector to a sequence.

10.4. Encoding with RNN

While in feedforward neural network the input is only at the beginning of the chain and the parameter matrix W is different at each layer, in recurrent neural networks the "external input" arrives at each layer and contribute to the argument of the activation function together with the flow of information coming from the previous layer (*recurrent* refers to this state information that, properly transformed, flows across the various layers).

One simple implementation of each layer transformation (that here we can see it as an "update") is hence (omitting the offset parameters):

$$s_t = \tanh(W^{s,s} s_{t-1} + W^{s,x} x_t)$$

Where:

- s_{t-1} is a $m \times 1$ vector of the old "context" or "state" (the data coming from the previous layer)
- $W^{s,s}$ is a $m \times m$ matrix of the weights associated to the existing state, whose role is to deciding what part of the previous information should be keep (and note that this is not changing in each layer.). Can also be interpreted as giving how the state would evolve in absence of any new information.
- x_t is a $d \times 1$ feature representation of the new information (e.g. a new word)
- $W^{s,x}$ is a $m \times m$ weights whose role is deciding how to take into account the new information, so that the result of wx multiplication is specific to each new information arriving;
- $tanh(\cdot)$ is the activation function (to be applied elementwise)
- s_t is a $m \times 1$ vector of the new "context" or "state" (the updated state with

the new information taken into account)

RNN have hence a number of layers equal to the data in the sequence, like the words in a sentence. So there is a single, evolving, NN for the whole sequence rather than a different NN for each element of the sequence. The initial state (S_0) is a vector of m zeros.

Note that this parametric approach let the way we introduce new data ($W^{s,x}$) to adjust to the way we use the network, i.e. to be learned according to the specific problem on hand...

In other words, we can adjust those parameters to make this representation of a sequence appropriate for the task that we are trying to solve.

10.5. Gating and LSTM

Learning RNNs

Learning a RNN is similar to learning a feedforward neural network: the first task is to define a Loss function and then find the weights that minimise it, for example using a SGD algorithm where the gradient with respect to the weights is computed using back-propagation. The fact that the parameters are shared in RNN's means that we add the contribution of each of the suggested modifications for parameters at each of these positions where the transformation is flagged. One problem of simple RNN models is that the gradient can vanish or explode. Here even more than in feedforward NN, as the sequences can be quite long and we apply this transformation repeatedly. In real cases, the design of the transformation can be improved to counter this issue.

Simple gated RNN

Further, in simple RNN the state information is *always* updated with new data, so far away information is easily "forget". It is often helpful to retain (or learn to retain) some control over what is written over and what is incorporated as new information. We can apply this control over what we overwrite and what instead we retain from the old state when meet new data using a form of RNN called **gated recursive neural networks**. Gated networks adds *gates* controlling the flow of information. Its simplest implementation can be written as:

$$g_t = \text{sigmoid}(W^{g,s}s_{t-1} + W^{g,x}x_t) = \frac{1}{1+e^{-(W^{g,s}s_{t-1}+W^{g,x}x_t)}}$$

$$s_t = (1-g_t) \odot s_{t-1} + g_t \odot \tanh(W^{s,s}s_{t-1} + W^{s,x}x_t)$$

Where the first equation defines a **gate** responsible to filter in the new information (trough its own parameters to be learn as well) and results in a continuous value between 0 and 1.

The second equation then uses the gate to define, for each individual value of the state vector (the $\odot$ symbol stands for element-wise multiplication), how much to retain and how much to update with the new information. For example, if $g_t[2]$ (the second element of the gate at the t transformation) is 0 (an extreme value!), it means we keep in that transformation the old value of the state for the

second element, ignoring the transformation deriving from the new information.

Long Short Term Memory neural networks

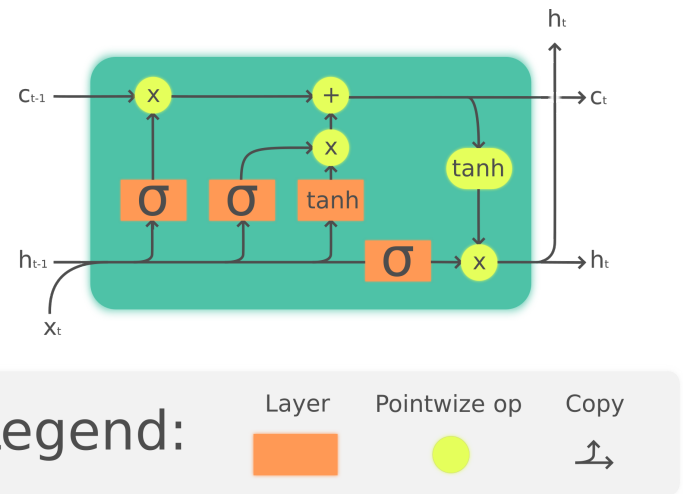
Real in-use gated RNN are even more complicated. In particular, **Long Short Term Memory** (recursive neural) networks (shortered as LSTM) are well-suited to classifying, processing and making predictions based on time series data, since there can be lags of unknown duration between important events in a time series and have the following gates defined:

$$
\begin{aligned}
f_t &= sigmoid(W^{f,h} h_{t-1} + W^{f,x} x_t) && \text{forget gate} \\
i_t &= sigmoid(W^{i,h} h_{t-1} + W^{i,x} x_t) && \text{input gate} \\
o_t &= sigmoid(W^{o,h} h_{t-1} + W^{o,x} x_t) && \text{output gate} \\
c_t &= f_t \odot c_{t-1} + i_t \odot \tanh(W^{c,h} h_{t-1} + W^{c,x} x_t) && \text{memory cell} \\
h_t &= o_t \odot tanh(c_t) && \text{visible state}
\end{aligned}
$$

The input, forget, and output gates control respectively how to read information into the memory cell, how to forget information that we've had previously, and how to output information from the memory cell into a visible form.

The "state" is now represented collectively by the memory cell c_t 'sometimes indicated as *long-term memory*) and its "visible" state h_t (sometimes indicated as *working memory* or *hidden state*).

LSTM cells have hence the following diagram:



The memory cell update is helpful to retain information over a longer sequences. But we keep his memory cell hidden and instead only reveal the visible portion of this tape. And it is this h_t then at the end of the whole sequence applying this box along the sequence that we will use as the vector representation for the sequence.

On the LSTM topic one could also look at these external resources:

- http://blog.echen.me/2017/05/30/exploring-lstms/
- https://colah.github.io/posts/2015-08-Understanding-LSTMs/
- https://machinelearningmastery.com/handle-long-sequences-long-short-term-

memory-recurrent-neural-networks/

Key things

- Neural networks for sequences: encoding
- RNNs, unfolded
  - state evolution, gates
  - relation to feed-forward neural networks
  - back-propagation (conceptually)
- Issues: vanishing/exploding gradient
- LSTM (operationally)

In other words:

- RNN's turn sequences into vectors (encoding)
- They can be understood as feed-forward neural networks whose architecture has changed from one sequence to another
- Can be learned with back-propagation of the error signal like for feedforward NN
- They too suffer of vanishing or exploding gradient issues
- Specific architectures such as the LSTM maintain a better control over the information that's retained or updated along the sequence, and they are therefore easier to train

Lecture 11. Recurrent Neural Networks 2

11.1. Objective

From Markov model to recurrent neural networks (RNNs)

- Formulate, estimate and sample sequences from Markov models.
- Understand the relation between RNNs and Markov model for generating sequences.
- Understand the process of decoding of RNN in generating sequences.

11.2. Markov Models

Outline

- Modeling sequences: language models
  - Markov models
  - as neural networks
  - hidden state, Recurrent Neural Networks (RNNs)
- Example: decoding images into sentences

While in the last lesson we saw how to transform a sentence into a vector, in a parametrized way that can be optimized for what we want the vector to do, today we're going to be talking about how to generate sequences using recurrent neural networks (decoding). For example, in the translation domain, how to unravel the output vector as a sentence in an other language.

Markov models

One way to implement prediction in a sequence is to just first define probabilities for any possible combination looking at existing data (for example, in a next word prediction - "language modelling", look at all two-word combinations in a serie of texts) and then, once we observe a case, sample the next element from this conditional (discrete) probability distribution. This is a first order Markov model.

Markov textual bigram model exemple

In this example we want to predict the next word based exclusively on the previous one.

We first learn the probability of any pair of words from data ("corpus"). For practical reasons, we consider a vocabulary V of the n more frequent words (and symbols), labelling all the others as `UNK` (a catch-all for "unknown"). To this vocabulary we also add two special symbols `<beg>` and `<end>` to mark respectively the beginning and the ending of the sentence. So a pair `(<beg>,w)` would represent a word $w \in V$ starting the sentence and `(w,<end>)` would represent the world w ending the sentence.

We can now estimate the probability for each words w and $w' \in V$ that w' follows w, that is the conditional probability that next word is w' given the previous one was w, by a normalised counting of the successive occurrences of the pair (w, w') (matching statistics):

$$\hat{P}(w'|w) = \frac{count(w,w')}{\sum_{w_i \in V} count(w,w_i)}$$

(we could be a bit smarter and consider at the denominator only the set of pairs whose first element is w rather than the whole $V \in n^2$)

For a bigram we would obtain a probability table like the following one:

w_{i-1} \ w_i	ML	course	is	UNK	<end>
<beg>	0.7	0.1	0.1	0.1	0.0
ML	0.1	0.5	0.2	0.1	0.1
course	0.0	0.0	0.7	0.1	0.2
is	0.1	0.3	0.0	0.6	0.0
UNK	0.1	0.2	0.2	0.3	0.2

At this point we can generate a sentence by each time sampling from the conditional probability mass function of the last observed word, starting with `<beg>` (first row in the table) and until we sample a `<end>`.

We can use the table also to define the probability of any given sentence by just using the probability multiplication rule. The probability of any N words sentence (including `<beg>` and `<end>`) is then $P = \prod_{i=2}^{N} P(w_i|w_{i-1})$. For example, given the above table, the probability of the sentence "Course is great" would be $0.1 * 0.7 * 0.6 * 0.2$.

Note that, using the counting approach, the model maximise the probability that the generated sequences correspond to the observed sequences in the corpus, i.e. counting corresponds here to the Maximum Likelihood Estimation of the conditional probabilities. Also, the same approach can be used to model words characted by character rather than sentences world by world.

Outline detailed

In this segment we considered language modelling, using the very simple sequence model called Markov model. In the next segments of this lecture we're going to turn those Markov models into a neural network models, first as feed-forward neural network models. And then we will add a hidden state and turn them into recurrent neural network models. And finally, we'll just consider briefly an example of unraveling a vector representation, in this case, coming from an image to a sequence.

11.3. Markov Models to Feedforward Neural Nets

We can represent the first order Markov model described in the previous segment as a feed-forward neural network,

We start by a one-hot encoding of the words, i.e. each input word would activate one unique node on the input layer of the network (**x**). In other words, if the vocabulary is composed of K words, the input layer of the neural network has width K, and it is filed all with zeros except for the specific word encoded as 1.

We want as output of the neural network the PMF conditional to that specific word in the input. So the output layer has too one unit for each possible word, returning each node the probability that the next word is that specific one given that the previous one was those encoded in x: $P_k = P(w_i = k|w_{i-1})$

Given the weights of this neural network W (not to be confused with the words w), the argument of the activation function of each node of the output layer is $z_k = \mathbf{x} \cdot \mathbf{W}_k + W_{0,k}$.

These z are real numbers. To transform in probabilities (all positive, sum equal to 1) we use as activation function the non-linear Softmax function:

$$P_k = \frac{e^{Z_k}}{\sum_{j=1}^{K} e^{Z_j}}$$

(and the output layer is then called **softmax output layer**).

Advantages of neural network representation

Frexibility

We can use as input of the neural network a vector x coposed of the one-hot econding of the previous world, *plus* the vector of the one-hot encoding of the second previous word, obtaining the probability that the next word is w_k conditional to the two preceding words (roughly similar to a second order Markov model).

Further, we could also insert an hidden layer in between th input and output layers, in order to look at more complex combinations of the preceding two words, in terms of how they are mapped to the probability values over the next word.

Parsimony

For the tri-gram model above, the neural network would need a weight matrix W of $2K \times K$ parameters, i.e; a total og $2K^2$ parameters.

Using a second order Markov model would require instead K^3 parameters: we take two preceding words, and for any such combination, we predict a probability of what the next word is. So if we have here the possible words roughly speaking, then each combination of preceding words, the square root of them, and then for each of those combinations, we have to have a probability value of each of the next words. So we will look at that times the number of parameters in the tri-gram model. So roughly, the number of words to the power of 3.

As a result, the neural network representation for a tri-gram model is not as general, but it is much more feasible in terms of the number of parameters that we have to estimate. But we can always increase the complexity of the model in the neural network representation by adding a hidden layer.

11.4. RNN Deeper Dive

We can now translate the feedforward rural network in a Recursive Neural Network that accepts a *sequence* of words as input and can account for a variable history in making predictionsfor the next word rather than on a fixed number of elements (as 1 or 2 in bigrams and trigrams respectively).

The framework is similar to the RNN we saw in the previous lesson, with a state (initially set to $\mathbf{0} \in R^K$) that is updated, at each new information, with a function of the previous state and the new observed word (typically with something like $s_t = tanh(W^{s,s}s_{t-1} + W^{s,w}x_t)$). Note that x_t is the one hot encoding of the new observed word.

The difference is that in addition, at each step, we have also an output that transforms the state in probability (representing the new, conditional PMF): $p_t = softmax(W^0 s_t)$.

Note that the state here retains information of all the history of the sentence, hence the probability is conditional to all the previous history in the sentence.

In other words, while $W^{s,s}$ and $W^{s,w}$ role is to select and encode the relevant features from the previous history and the new data respectively, W^0 role is to extract the relevant features from the memorised state with the aim of making a prediction.

Finally we can have more complex structures, like LSTM networks, with forget, input and output gates and the state divided in a memory cell and a visible state. Also in this case we would however have a further transformation that output

the conditional PMF as function of the visible state ($p_t = softmax(W^0 h_t)$).

Note that the training phase is done computing the average loss at the level of sentences, i.e. our labels are the full sentences, and are these that are compared in the loss function with those obtained by sampling the PMFs resulting from the RNN. While in *training* we use the true words specified as input for the next time step, in *testing* instead we let the rural network to predict the sentence on its own, using the sampled output at one time step as the input for the next step.

11.5. RNN Decoding

The question is how to use now these (trained) RNN models to generate a sentence from a given encoded state (e.g. in testing)?

The only difference is that the initial state is not a vector of zero, but the "encoded sentence". We just start with that state and the `<beg>` symbol as x and then let the RNN produce a PDF, sample from that and use that sampled data as the new x for the next step and so on until we sample an `<end>`.

We don't just have to take a vector from a sentence and translate it into another sentence. We can take a vector of representation of an image that we have learned or are learning in the same process and translate that into a sentence.

So we could take an image, translate into a state using a convolutional neural network and then this state is the initial state of an other neural network predicting the caption sentence associated to that image (e.g. "A person riding a motorcycle on a dirt road")

Key summary

- Markov models for sequences
  - how to formulate, estimate, sample sequences from
- RNNs for generating (decoding) sequences
  - relation to Markov models (how to translate Markov models into RNN)
  - evolving hidden state
  - sampling from the RNN at each point
- Decoding vectors into sequences (once we have this architecture that can generate sequences, such as sentences, we can start any vector coming from a sentence, image, or other context and unravel it as a sequence, e.g. as a natural language sentence.)

Homework 4

Lecture 12. Convolutional Neural Networks (CNNs)

12.1. Objectives

At the end of this lecture, you will be able to

- Know the differences between feed-forward and Convolutional neural

networks (CNNs).
- Implement the key parts in the CNNs, including *convolution*, *max pooling* units.
- Determine the dimension of each channel in different layers with a given CNNs.

12.2. Convolutional Neural Networks

CNNs Outline:

- why not use unstructured feed-forward models?
- key parts: convolution, pooling
- examples

The problem: image classification, i.e. multi-way classification of images mapping to a specific category (e.g. x is a given image, the label is “dog” or “car”).

Why we don’t use “normal” feed-forward neural networks ?

1. *It would need too many parameters*. If an image is mid-size resolution 1000×1000, each layer would need a weight matrix connecting all these 10^6 pixel in input with 10^6 pixel in output, i.e. 10^12 weights
2. *Local learning (in the image) would not extend to global learning*. If we train the network to recognise cars, and it happens that our training photos have the cars all in the bottom half, then the network would not recognise a car in the top half, as these would activate different neurons.

We solve these problems employing specialised network architectures called **convolution neural network**.

In these networks the layer l is obtained by operating over the image at layer $l-1$ a small **filter** (or **kernel**) that is slid across the image with a step of 1 (typically) or more pixels at the time. The step is called **stride**, while the whole process of sliding the filter throughout the whole image can be mathematically seen as a **convolution**.

So, while we slide the filter, at each location of the filter, the output is composed of the dot product between the values of the filter and the corresponding location in the image (both vectorised), where the values of the filters are the weigths that we want to learn, and they remain constant across the sliding. If our filter is a 10×10 matrix, we have only 10 weights to learn by layer (plus one for the offset). Exactly as for feedforward neural networks, then the dot product is passed through an activation function, here typically the `ReLU` function ($max(0, x)$) rather than `tanh`.

For example, given an image $x = \begin{bmatrix} 1 & 1 & 2 & 1 & 1 \\ 3 & 1 & 4 & 1 & 1 \\ 1 & 3 & 1 & 2 & 2 \\ 1 & 2 & 1 & 1 & 1 \\ 1 & 1 & 2 & 1 & 1 \end{bmatrix}$ and filter weights

$w = \begin{bmatrix} 1 & -2 & 0 \\ 1 & 0 & 1 \\ -1 & 1 & 0 \end{bmatrix}$, then the output of the filter z would be $\begin{bmatrix} 8 & -3 & 6 \\ 4 & -3 & 5 \\ -3 & 5 & -2 \end{bmatrix}$.
For example, the element of this matrix $z_{2,3} = 5$ is the result of the sum of the scalar multiplication between $x' = \begin{bmatrix} 4 & 1 & 1 \\ 1 & 2 & 2 \\ 1 & 1 & 1 \end{bmatrix}$ and w.

Finally, the output of the layer would be (using ReLU) $\begin{bmatrix} 8 & 0 & 6 \\ 4 & 0 & 5 \\ 0 & 5 & 0 \end{bmatrix}$.

For example you can obtain the above with the following Julia code:

```
ReLU(x) = max(0,x)
x = [1 1 2 1 1;
     3 1 4 1 1;
     1 3 1 2 2;
     1 2 1 1 1;
     1 1 2 1 1]
w = [ 1 -2  0;
      1  0  1;
     -1  1  0]
(xr,xc) = size(x)
(wr,wc) = size(w)
z = [sum(x[r:r+wr-1,c:c+wc-1] .* w) for c in 1:xc-wc+1 for r in 1:xr-wr+1] # Julia is column mayor
u = ReLU.(z)
final = reshape(u, xr-wr+1, xc-wc+1)
```

(You can play with the above code snippet online without registration on https://bitly.com/convLayer)

You can notice that, applying the filter, we obtain a dimensionality reduction. This reduction depends on both the dimension of the filter and the stride (sliding step). In order to avoid this, a padding of one or more zeros can be applied to the image in order to keep the same dimensions in the output (in the above example a padding of one zeros on both sides - and both dimensions - would suffice).

Because the weight of the filters are the same, it doesn't really matter where the object is learned, in which part of the image. The lecture explains this with a mushroom example: ff the mushroom is in a different place, in a feed-forward neural network the weight matrix parameters at that location need to learn to recognize the mushroom anew. With convolutional layers, we have translational invariance as the same filter is passed over the entire image. Therefore, it will detect the mushroom regardless of its location.

Still, it is often convenient to operate some **data augmentation** to the training set, that is to add slightly modified images (rotated, mirrored…) in order to improve this translational invariance.

A further way to improve translational invariance, but also have some dimensionality reduction, it called **pooling** and is adding a layer with a filter whose output is the `max` of the corresponding area in the input. Note that this

layer would have no weights! With pooling we contribute to start separating what is in the image from where it is in the image, that is pooling does a fine-scale, local translational invariance, while convolution does more a large-scale one.

Keeping the output of the above example as input, a pooling layer with a 2×2 filter and a stride of 1 would result in $\begin{bmatrix} 8 & 6 \\ 5 & 5 \end{bmatrix}$.

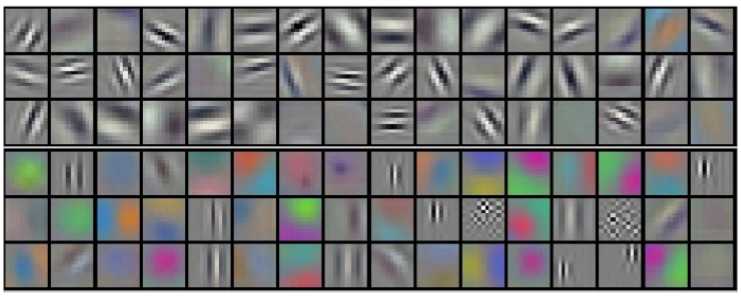
In a typical CNN, these convolutional and pooling layers are repeated several times, where the initial few layers typically would capture the simpler and smaller features, whereas the later layers would use information from these low-level features to identify more complex and sophisticated features, like characterisations of a scene. The learned weights would hence specialise across the layers in a sequence like edges -> simple parts-> parts -> objects -> scenes.

Concerning the topic of CNN, see also the superb lecture of Andrej Karpathy on YouTube (here or here).

12.3. CNN - Continued

CNN's architectures combine these type of layers successively in a variety of different ways.

Typically, one single layer is formed by applying multiple filter, not just one. This is because we want to learn different kind of features... for example one filter will specialize to catch vertical lines in the image, an other obliques ones... and maybe an other different colours. Indeed in real implementation, both the image and the filter have normally a further dimensions to account for colours, so they can modelled as volumes rather than areas:



96 convolutional filters on the first layer (filters are of size 11x11x3, applied across input images of size 224x224x3). They all have been learned starting from random initialisation.

So in each layer we are going to map the original image into multiple feature maps where each feature map is generated by a little weight matrix, the filter, that defines the little classifier that's run through the original image to get the associated feature map. Each of these feature maps defines a channel for information.

I can then combine these convolution, looking for features, and pooling, compressing the image a little bit, forgetting the information of where things are, but maintaining what is there.

These layers are finally followed by some “normal”, “fully connected” layers (*à la* feed-forward neural networks) and a final `softmax` layer indicating the probability that each image represents one of the possible categories (there could be thousand of them).

The best network implementation are tested in so called “competitions”, like the yearly ImageNet context.

Note that we can train this networks exactly like for feedformard NN, defining a loss function and finding the weights that minimise the loss function. In particular we can apply the stochastic gradient descendent algorithm (with a few tricks based on getting pairs of image and the corresponding label), where the gradient with respect to the various parameters (weights) is obtained by backpropagation.

Take home

- Understand what a convolution is;
- Understand the pooling, that tries to generate a slightly more compressed image forgetting where things are, but maintaining information about what’s there, what was activated.

Recitation: Convolution/Cross Correlation:

Convolution (of continuous functions): $(f * g)(t) := \int_{-\infty}^{\infty} f(\tau)g(t-\tau)d\tau$

Cross-correlation: $(f * g)(t) := \int_{-\infty}^{\infty} f(\tau)g(t+\tau)d\tau$ (i.e. like convolution, but where the g function is not reversed/flipped)

In neural network we use the cross-correlation rather than the convolution. Indeed, in such context, f is the data signal and g is the filter. But the filter needs to be learn, so if it is expressed straight or reversed, doesn’t really matter, so we just use the cross-correlation and save one operation.

1-D Discrete version

Cross correlation of discrete functions: $(f * g)(t) := \sum_{\tau=-\infty}^{\infty} f(\tau)g(t+\tau)$

The cross-correlation $h = (f * g)(t)$ for discrete functions can be pictured in the following scheme, where the output is the cross product of the values of f and g in the same column, and where out of domain values are padded with zeros.

```
f:           1 2 3
...
       0| 1 2
h(t=0) 2|   1 2
h(t=1) 5|     1 2
h(t=2) 8|       1 2
h(t=3) 3|         1 2
h(t=4) 0|           1 2
...
```

2-D Discrete version

In 2-D the cross correlation becomes: $(f * g)(x, y) := \sum_{\tau_1=-\infty}^{\infty} \sum_{\tau_2=-\infty}^{\infty} f(\tau_1, \tau_2) g(\tau_1 + x, \tau_2 + y)$.

Graphically, it is the sliding of the filter (first row, left to right, second row, left to right, ...) that we saw in the previous segment.

Project 3: Digit recognition (Part 2)

[MITx 6.86x Notes Index]
