Notes on Machine Learning

Vincenzo Gargano

February 24, 2023

Contents

Chapter 1

Neural Networks

1.1 Regularization

Loss is Error + Regularization term and we have many approaches: like coefficient shrinkage aka Tikhnov regularization

$$Loss(W) = \sum_{p=1}^{l}(y_p - y_p)^2 + \lambda * ||W||^2$$

We can control it with the hyperparameter λ and we can use it to avoid overfitting. We can remeber when we talked about VC-dimension, this is a way to manage this dimensionality

Small lamba means high norm of the weights and a too complex model going to overfit data. Large lambda take the second term to grow too much, bringing to a growth of the error data that tends to underfit. We call the penalty term because penalize high values of the weights. Some weights can even go to zero. If you use only L2 is called Ridge, if you use only L1 is called Lasso, if you use both is called Elastic Net. L2 tends to bring weights to smaller values, L1 penalize the absolute values and bring some weights to 0 and allows some weights to be larger, but Lasso has non differentiable loss.

Exercises. Connect the regularization term with the VC-dimension
- Why this can have a better lower bound R?
- How λ values can rule the underfit and overfit cases?
Derive the new δ rule with weights decay using Tikhnov loss: computing ∂ of Loss with respect to , separating (η) and (λ)

□

1.2 Stopping Criteria

The basic is the used error mean error ¡ E, it is the best if you know the tolerance of the data (expert knowledge), but often we haven't information or tolerance of when to stop. Our project is like this. We need to use internal criteria for example weight changes are very smol near zero gradient, or error isn't decreasing for an epoch for example less than 0.1%NOTE: may be premature (for small eta), it can be applied observing k epochs (patients)

Just escape after an excessive number of epochs to escape from slow convergence), avoiding to stop in a fixed number of epochs

1.3 VC-Dimension

Theorem 1. *The **VC-Dimensions** of a class of function H is the maximum cardinality of a set of points in X that can be **shattered** by H.*

For linear functions:
$VC(H) \geq 3$ but we need at least one configuration that can be shattered, for example the 3 dots example. For $VC(H) < 4$ instead we need non linear functions

In general the VC-Dimension of a class of linear separator hyperplanes (LTU) in an n-dimensional space is n+1.

VC-dimension is not the number of free parameters, but they are related, there are model with one parameter and infinite VC-dim, see Haykin book. For Nearest-neighbour the VC-dim is infinite, for example for 1-Nearest-Neighbour we can have infinite points and 0 error! 1-Nearest-Neighbour is not a model indeed.

1.3.1 Statistical Learning Theory

Assume **N** the number of data (l)

The guaranteed risk: $R[h] = R_{emp}[h] + \epsilon(VC, N, \delta)$

The second term the VC-confidence $\epsilon(VC, N, \delta) = \sqrt{\frac{VC(ln\frac{2N}{VC}+1)-ln\delta/4}{N}}$, with probability at least $(1-\delta)$ for every $VC < N$

Remember the U-shaped plot of VC-Dimensions with ϵ going to zero, increasing **N**, ϵ grows with VC, and we get the U shape by R increasing the VC-dimension

This gives us a way to estimate the error on future data based only on the training error and VC-Dimension of H. This provides good information about model selection and assessment because don't take into account the process of cross-validation that is costful.

1.4 Intro to SVM

1.4.1 Linear SVM

We are now making a connection between VC-Dimension, SLT and Linear Threshol Unit (LTU)
We follow Haykin chapter 6.

- **N** is the number of examples, before used l
- **m** is dimension of input vector, before used n
- **b** instead of w_0 as intercept or bias

Assumptions: Hard margin SVM, we assume linear separable problem and no error in the data

Separating hyperplane: $w^T x + b = 0$
where $w^T x_i + b \geq 0$ for $d_i = +1$ and the other way around for $d_i = -1$

$g(x) = w^T x + b$ is the discriminant function and $h(x) = sign(g(x))$ is the hypotesis

The separation margin ρ is evaluated as the double of the distance between the hyperplane and the closes data point. We can call it "safe zone"

Also not all the hyperplanes have equals safe zones, maybe with larges or smaller margin. We want to make a preference for larger margins.

We define the optimal hyperplane as the one with that maximizethe margin ρ.

$$w_o^T x + b_o = 0 \tag{1.1}$$

We wat to find $\rho = \frac{2}{\|w\|}$, so we need to maximize ρ and minimize $\|w\|$ This is an optimization problem in terms of minimize the norm, in order to get a new LTU that maximize the margin. We can see later why is this relevant.

We can rescale, (like normalize the distance to 1 or -1 for the points) w and b and so that the closest points to hyperplane satisfy $g(x_i) = |w^T x_i + b| = 1$ and then write in a compact form:

$$d_i(w^T x_i + b) = 1, \forall i = 1, ..., N \tag{1.2}$$

A **Support Vector** $x^{(s)}$ satisfies the previous equation exactly:

$$d^{(s)}(w^T x^{(s)} + b) = 1 \tag{1.3}$$

We have many support vectors, they are the closes points/datapoints to the boundary or hyperplane.
Let's call $g(x)$, discriminant as $g(x) = w^T x + b$, and recall that w_o is a vector orthogonal to the hyperplane, let's denote the distance between x and optimal hyperplane with r

$$x = x_p + r\frac{w_o}{\|w_o\|} \tag{1.4}$$

We now derive the margin evaluating $g(x)$ in the point x_p, evaluating $g(x) =$

$w_o^T x + b_o$ we obtain:

$$\begin{aligned} g(x) = g(x_p + r\frac{w_o}{\|w_o\|}) = \\ = w_o^T x_p + b_o + w_o^T r\frac{w_o}{\|w_o\|} = \\ = g(x_p) + r w_o^T \frac{w_o}{\|w_o\|} = \\ = r\frac{\|w_o\|^2}{\|w_o\|} =_5 r\|w_o\| \end{aligned} \tag{1.5}$$

We use the definition of $g(x)$ to get to the second row, we multiply w_o for the two terms basically.
But the first part is equal to $g(x_p)$, so we can write like that in the third equation. Now for definition of decision boundary the value of $g(x_p)$ is zero and we get only the second part, multipling we obtain the final result.
Thus

$$r = \frac{g(x)}{\|w_o\|} \tag{1.6}$$

Now we consider the distance between the hyperplane and a positive support vector $x^{(s)}$

$$r for x^{(s)} = \frac{g(x^{(s)})}{\|w_o\|} = \frac{1}{\|w_o\|} = \frac{\rho}{2} \tag{1.7}$$

ρ in this case is divided by two because we're taking half of the maximum margin.

1.4.2 Quadratic Optimization Problem

Find the optimum values of w and b that maximize the margin yields to a quadratic optimization problem
Given the training samples $T = \{(x^{(i)}, d^{(i)})\}$, find the optimum values of w and b that minimize the following equation:

$$\Psi(w) = \frac{1}{2} w^T w \quad (min.\|w\|) \tag{1.8}$$

Satisfiying the constraints (zero classification errors): $d^{(i)}(w^T x^{(i)} + b) \geq 1, \forall i = 1, ..., N$

We are searching the hyperplane that correctly classifies all the data and at the same time have the largest margin. The Perceptron is able to solve this problem, but this is not a random possible solution but the one with the max margin, this function is quadratic and convex in w, costraints are linear in w, and the problem scales with the size of input space m.

Why this is related to the Structural Risk Minimization? The next theorem is fundamental to understand why we presented the SVM. Because otherwise we have no other reason to introduce it instead of perceptron, Least Mean Squares...
We fixed the training error on linearly separable problems. Than minimizing the norm of w is equivalent to minimize the VC dimension and thus the capacity term (VC Confidence) $\epsilon(VC, N, \delta)$.

Theorem 2 (Vapnik)**.** *Let D be the diameter of the smallest ball around the data points $x_1, ...x_N$.*
For the class of sepaimrating hyperplane described by the equation $\mathbf{w^T x} + b = 0$ the upper bound to the VC-dimension is

$$VC \leq min(\frac{D^2}{\rho^2}, m_0) + 1 \tag{1.9}$$

This means that if in the class of our model, the linear models, the standard VC-Dim was equal to dimension of the problem plus one (remember example of the 3 points linearly separable). The VC-Dimension can be reduced by increasing the margin, because the upper bound is the minimum between the two, plus one accounting the fact that the VC dimension of a class of linear separator hyperplane in n-dimensional space is n+1. As in 1.3. So we're optimizing the VC-bound, the empirical risk is fixed to 0 as we said about fixing training error before. This corrisponds to regularization thus reducing VC dimension as we did with Tikhnov but coming from a completely different reasoning.
What is the best solution among all the possible hyperplanes? The one in the middle. This theorem relate the maximum margin with idea of minimum VC dimension.
Why support vector in the name? We'll discuss in the next lesson.

Tipically the problem in solved in the dual form, to move in the dual form.

To solve this problem we use the Lagrangian multipliers method, we build a Lagrangian function corresponding to the quadratic optimization problem.

$$J(w, b, \alpha) = \frac{1}{2} w^T w - \sum_{i=1}^{N} \alpha_i [d^{(i)}(w^T x^{(i)} + b) - 1] \tag{1.10}$$

Where α are N lagrangian multipliers, each term in the sum corresponding to one constraint of our primal pro$_b$*lemandweminimizeJwithrespecttowandbandmaximizeitwithrespectt*
The solution corresponds to a saddle point of J.
At the end we find w_o the optimal value for w and we express it in terms of α and $x^{(i)}$.
We then moving in the dual form searching the α that minimize the function. Inside this optimization problem there are property of Kuhn-Tucker Conditions that we will see

$$\begin{aligned} \alpha_i(d_i(w^T x_i + b) - 1) = 0, \forall i = 1, ..., N \\ \mu_i \xi_i = 0, \forall i = 1, ..., N \end{aligned} \tag{1.11}$$

in the saddle point of J.
Then if $\alpha_i > 0$ then $d_i(w^T x_i + b) - 1 = 0$ and the equation above will be zero and x_i is a support vector. If x_i isn't a support vector then $\alpha_i = 0$, solving in dual form need you to find the value of α that can be used to find the separating hyperplane but also the support vectors, so we can restrict only to search for support vectors because for all the others $\alpha = 0$. This is why is called SVM, it's based on the support vectors or inputs or patterns, datapoints, for which $\alpha \neq 0$. Of course we don't know the support vectors, they are computed with primal and then dual form.
So the hyperplane depends only on support vectors and we're finding them, to find them we need to find α.

Let's see how the dual form looks like, given $T = \{(x^{(i)}, d^{(i)})\}$, find the optimum values of the Lagrangian multipliers $\{\alpha_i\}_{i=1}^N$

$$Q(\alpha) = \sum_{i=1}^{N} \alpha_i - \frac{1}{2} \sum_{i=1}^{N} \sum_{j=1}^{N} \alpha_i \alpha_j d_i d_j \overbrace{x_i^T x_j}^{\text{dot product}} \tag{1.12}$$

With constraints:

- $\alpha_i \geq 0, \forall i = 1, ..., N$

- $\sum_{i=1}^{N} \alpha_i d_i = 0$

α can be found solving the quadratic programming problem (QP) or with recent efficient approaches (eg. SMO: Sequential Minimal Optimization). Note that this problem scales with number of training samples (N).

1.4.3 Why is this elegant?

For linearly separable problems, Vapnik propose an "optimal separating hyperplane" that has an unique solution, remember that perceptron has many solution, then with zero errors for binary classifier instead of LMS that have clssification error even if is the minimum in LMS, automatized approach to SRM that minimize VC-confidence while training without hyperparameters in linear-separable case, the use of a solver in class of constrained Quadratic Programming instead of gradient descent with a dual form and finally a solution focuse on selected training data, the support vectors, but what for noise points at the boundaries or non linearly separable data?
We introduce Soft margin, in which we admits errors inside the margin, but allowing error allow also to have larger margin, because sometime if we don't allow error we get a very very little margin, but only allowing few datapoints into the margin to be errors we can have a lot larger one!
We introduce non-negative scalar variables:

$$\xi_i \geq 0, \forall i = 1, ..., N \tag{1.13}$$

called slack variables

$$d_i(w^T x_i + b) \geq 1 - \xi_i, \forall i = 1, ..., N \tag{1.14}$$

So now a support vector x_i satisfies the previous exatly: $d_i(w^T x_i + b) = 1 - \xi_i$
Unfortunately Vapnik theorem doesn't hold anymore (hard margin doesn't admits points inside the margin).
Now we get a different primal form with new constraints, our Ψ function will have a new term with the slack variables.
But what is the bad news? Well... ξ is an hyperparamter that regulate the control of admitted error and how much is large our margin. We lose the magic word without hyperparameter, so options are: invent a new theory, you're welcome or do crossvalidation for model selection and suffer.

1.4.4 Mapping to high-dimensional space

Idea: for non linearly separable datapoints we map to an higher dimensional feature space to make them linearly separable.
We must recall LBE, large basis expansion, so we find a $\Phi(x)$ that maps to a different space. Eg. from 2D space to a 3D spac that maps to a different space. Eg. from 2D space to a 3D space.
But you won't get the best transformation adding only 1 dimension, unless you have a prior knowledge allowing you to select the proper feature space.).
The choice of the mapping is the hard part.

Theorem 3. *The pattern are linearly separable with high probability in the feature space under such conditions*

- *Finding the optimal hyperplane to separate the patterns in the feature space*
- *Using large basis expansion can be computationally unfeasible and lead to overfitting*

Kernel trick

So we introduce a kernel approach to manage the feature space in the context of regularized modeling.

$$\Phi : \mathbb{R}_0^m \to \mathbb{R}_1^n \text{ is a mapping from the original space to the feature space.} \tag{1.15}$$

$$\mathrm{x} \to \Phi(x) \tag{1.16}$$

We get the bias in the weight vector:

- $w(0) = b$
- $\Phi_0(x) = 1$

$$\Phi(x) = (\phi_0(x) = 1, \phi_1(x), ..., \phi_{m1}(x))^T \tag{1.17}$$

Decision surface equation: $w^T\Phi(x) = 0$, linear expansion of the basis function, where (i) the dimension of enarged space is allowed to get very larg and (ii) the complexity depends form the margin, not directly on the space dimension.
To express the Φ we use the dual form and the dot product is computed in the feature space, we don't need to know the shape of Φ but we want the result. Evaluating it could be intractable, so we call k an inner product kernel function.

$$k(x_i, x) = \Phi(x_i)^T\Phi(x) \tag{1.18}$$

Some properties are that k is symmetric function $k(x_i, x) = k(x, x_i)$
What is the computational vantage?
We consider two points x, y in $\mathbb{R}^2$ we compute the k component by component.
The dot product in feature space is evaluated without considering feature mapping and space itself, we evaluate the dot product in terms of input patterns.

Kernel Matrix

We can arrange dot products in the feature space between the images of the input training patterns in an N by N matrix called *Kernel Matrix*:

$$K = \begin{bmatrix} k(x_1, x_1) & k(x_1, x_2) & ... & k(x_1, x_N) \\ k(x_2, x_1) & k(x_2, x_2) & ... & k(x_2, x_N) \\ ... & ... & ... & ... \\ k(x_N, x_1) & k(x_N, x_2) & ... & k(x_N, x_N) \end{bmatrix} \tag{1.19}$$

$K = \{k(x_i, x_j)\}_{(i,j=1)}^N$
The kernel matrix hold all the kernel of the training data, and is symmetrical.
If you define by fantasy a kernel are we always able to find a Φ that corresponds to this kernel? NO!
This property holds only for kernels gaining positive semi-definite kernel matrices (Mercer's theorem), this is related to having non-negative Eigevalues of the kernel matrix.

Algebra of Kernels

If you have two valid kernels you can have an algebra describing the system. $Let k_1, k_2$ be two valid kernels, over the same domain R_0^m.

$k_1(x,y) + k_2(x,y)$ is a valid kernel

$k_1(x,y)k_2(x,y)$ is a valid kernel

$\alpha k_1(x,y) \forall \alpha \in \mathbb{R}_+$ is a valid kernel

Now the kernel is useful, assuming we know the transformation, the primal form doesnt change from soft margin, but now the w is in the feature space and this can lead to intractable problem. (suffer with large dimensional spaces), but when we go in the dual form the advantage is that we can exploit the fact that the complexity is bounded by the number of number of data, so we can use large space!

$$Q(\alpha) = \sum_{i=1}^{N} \alpha_i - \frac{1}{2} \sum_{i=1}^{N} \sum_{j=1}^{N} \alpha_i \alpha_j t_i t_j k(x_i, x_j) \tag{1.20}$$

Where α_i is bounded by 0 and C the regularization parameter.
Sparsity: remember that the opt. problem solution gains sparse solution in $alpha_{i_{i=1}}^{N}$ because the sum can be only applied for support vectors the others are 0*α
Test phase: Use machine for unseen input patterns x, compute the sum of $\alpha_i d_i k(x, x_i)$, and then classify x as the sign of the previous :

$$\mathrm{h(x)} = \mathrm{sign}(\textstyle\sum_{i=1}^{N} \alpha_i d_i k(x, x_i)) \tag{1.21}$$

What does this resemble? K-nn!

1.4.5 SVM and Neuron architecture

Draw an SVM architecture example comparing with 1 layer neuron using latex and tikz

Think about what is different between these two models, SVM as we can see have one "hidden layer", while NN can have more, there are

different way we an manipulate the complexity, efficiency and other things, also NN has adaptive free parameters in the units

From professor: Intelligent is find the similarity when people think things are different and find differences when people think their are equal

1.4.6 Well known Kernels

- Polynomial kernel: $k(x, y) = (x^T y + 1)^d$
- Radial Basis Function Net: $k(x, y) = e^{-\frac{1}{2\sigma^2}||x - x_i||^2}$ where σ^2 is a user parameter, known as Gaussian kernel.
- Two-layer perceptron $k(x, y) = tanh(\beta_0 x^T y + \beta_1)$ where $\beta_0 > 0$ and $\beta_1 < 0$ are specified by user, you can youse hyperbolic tangent as a kernel if you want to be (no one apply this one)

The most popular is the Gaussian Kernel, picture if have a very small sigma, you get a narrow gaussian curve over the support vector x_i all the other is zero, you have 1 only when $x = x_i$ not considering distant points and this resemble 1-NearestNeighbour, this is like a shortcut to a different model, and this expalin why this is popular, 1-NN have small training error, and works well, also SVM doesn't fall in overfitting, well 1-NN is very prone to overfit, but σ is a relevant hyperparameter. Also the RBF brings always to an infinite feature space dimensions, we don't know the Φ.

1.4.7 SVM for non-linear regression

Recall regression problem: we have an unknown function $f(x)$ and a noisy term v statistically independent of the input vector x, we have a training set $T = \{(x_i, d_i)\}_{i=1}^N$, we estimate d using linear expansion of non-linear function $\{\phi_j(x)\}_{j=0}^{m_1}$.
What change is the loss function, an expedient of Vapnik, called $\epsilon - insensivite$ loss function, assume linear values positive or negative outer the $-\epsilon, +\epsilon$ zone.
The SVR for regression what does? Try to approximate and stay close

to the points, the ϵ is like a tube around the targed d, all the points inside are considered right and they are not support vectors, the support vectors are points outside the margin (the tube).
Now model produced by SVR depends on a subset of training data, because the cost function for given model ignore the training data close to the model prediction. The opposite from the other. Now described by point close or far. In mathematical terms.

$$-\xi_i^{'} - \epsilon \leq w^T\phi(x_i) \leq \epsilon - \xi_i, \forall i \in \{1, ..., N\} \tag{1.22}$$

$\xi_i, \xi_i^{'}$ are still slack variables non negative and we have this constraints:

$$\begin{aligned} d_i - w^T\phi(x_i) &\leq \epsilon + \xi_i \\ w^T\phi(x_i) - d_i &\leq \epsilon + \xi_i^{'} \\ \xi_i, \xi_i^{'} &\geq 0 \end{aligned} \tag{1.23}$$

Then you setup the primal form with the new constraints (stay inside the ϵ tube), the dual problem we have $\alpha, \alpha^{'}$, solving we get the optimal values for the Lagrangian multipliers (the alphas) and we compute the value vector of w.
The estimated function ca be expressed in terms of $h(x) = y = w^T\phi(x)$. Using linear expansion we get:

$$h(x) = \sum_{i=1}^{N} \gamma_i \phi(x_i)^T \phi(x) = \sum_{i=1}^{N} \gamma_i k(x_i, x) \tag{1.24}$$

Where γ_i is the difference between the lagrangian multipliers and support vectors corresponds to non-zero values of γ_i.
Now you have to choose also ϵ other than C, then solve automatically presenting new pattern as we saw before, assumed you memorized the support vectors.

What you should know?

- Defintion of a maximum margin classifier
- How maximum margin can be turned in QP problem?

- What QP can do for you? (but for this class you don't need to know how it does it)
- Why SVM approximate SRM?
- How we deal with noisy (non-separable) data? (Consider error term as a hyperparameter)
- How we permit non-linearb boundaries?
- How SVM Kernel function permit us to pretend we're working with ultra-high-dimensional basis function terms?

Answer to this question to be prepared on this.

References on this : Pratical guide to SVClassification, Burges a tutorial on SVM for pattern recognition, nice but not so easy, SLT and kernel methods Bernhard Scholkopf, most important is **Haykin chapter 6**

1.4.8 Practical aspects of SVM

Pro and Cons of SVM: Regularization embedded in optimization problem with idea of margin, approximation of theoretical SRM (bound to vc-dimension of hyperplane minimized with solution), convex problem so we have everytime global minimum, implicit feature transformation using kernels, but:
Must use kernel and kernel parameters
Batch Algorithm Very large problem are computationally intractable, problems with more than 20k examples are difficult with standard approach, many solution today use gradient descent. Advantages are: Linear model as solver bounding complexity by margin that is optimized, so simple and compact model, also a rich set of non-linear decision function in the input (sample) space via kernels, exploiting a large LBE (linear basis expansion but non necessary increase VC-dimension). For SVM they depends on margin, K. parameters, C etc...
Kernel trick that bring in higher dimenson to keep linear separation!

Historical SVM application

First famous application is the task of MNIST dataset on handwritten digits, 0.8% of error in 1998 with a degree-9 polynimial SVM, noteworthy comparable to LeNet (CNN)!
This teach to us relevance of application, more attention was given to this SVM and SLT, from good theory successful application.

"Currently there exists no theory which guarantees that a given family of SMVs will have high accuracy on a given problem" - Burges

Similar to the no free-lunch theorem.

Consider an RBF SVM with little σ can classify a larg number of training correctly like 1-NN or 1 SV point, and have infinite VC dimension (unless you regularize, in the other case you overfit), on the opposite a large width of the gaussian all SV point all considered a global average, and we obtain a low VC-dimension, so the width controls VC-dim.
To apply to pratical problems SVM one needs to choose type of kernel and parameters, the C values and ϵ for regression, hyperparameters affecting model complexity, that is used for model selection. Now SVM is not modern, but maybe in future you.

And now some SVM Folklore and errors:

- SVM can be used with default hyper-parameters values
- I choose SVM because it doesn't not fall into overfitting (Professor read this in a published paper, the author was an engeener)
- SVM has no problem using using high input dimension. (Does it solve curse of dimensionality? This is a rather intresting question)

  We have not solved the curse of dimensionality in the input space, we have transformed in a feature space with high dimension but this is not a changing in the input dimension, if your input is high still problem remains! So the answer is SVM doesn't not solve the CoD in the input space...
- By SVM is like to find automatically the "numbers of units"

The kernel trick and kernel methods also without SVM, are very efficient at fixed high-dimension (LBE), they are modular because of choice of basis function by kernel: data matrix, kernel function and solver, you can change the kernel without change the solver! Now you inherit form mathematich an unified theoretical framework for general dot product functions.

Associate distance to kernels

$K(s,t)$ can be related to a similarity measure comparing s and t

$$d_k(s,t) = \sqrt{K(s,s) + K(t,t) - 2K(s,t)} \tag{1.25}$$

Similar objects have small distances if and only if high values of kernel. This idea of use kernelization focus on represent pairwise comparison on instead of single object, for some domains it may be easier to compare two object istead defining some abstract space of features where to represent a single object. In mathematical term we have a distance function induced by the dot product, and this is a distance in a mathematical sense, it satisfies the property of distances, like distance between an object and itself is zero, symmetryic, then you have triangular inequality and so on.
However an SVM is highly characterized by choice of the kernel, but the best choice of kernel for a given problem is still an open problem!

Exercise: Re-think it as for distance based methods: when is it a good kernel? When is bad?

Answer of a good kernel: A good kernel for object is when you pick a set of object (original space) and project them in a feature (vector) space, similar figures are near each other like triangle so it's easy to be classified. In this case with respect to the shape. See the figure in the slides.

Answer of a bad kernel: A bad kernel is when you associate two pictures in the same point, then the SVM cannot separate the two, the point association in the feature space make it impossible or when you have a kernel matrix that is diagonal, so each point is similar only to

itself, so there is no way of do comparisons. No free-kernels...

Design of new kernels you need to choose a "good" kernel function for the domain and it can be generalized not only to vectors but string, trees, graphs, ad hoc data domain to encode useful measure in bioinformatic (for example is popular edit-distance kernel or language suffix-tree kernel), efficiency to make faster computation and adaptive kernels that is under research, how to change kernel considering current task (NN do it automatically) and this explains an important difference between the two.

1.5 Bias-Variance

A training set is a possible realiation from universe of data: different trainin sets can provide different estimation.
We can decompose the expected error on TR at a point x in:

- Bias: Will quantify the discrepances between true function and $h(x)$, if H is too small it is high, small hypotesis space
- Variance: Quantify variability of response of model h for different realization of training data, we want a training data that generalize all the realizations
- Noise: labels include random error. eg: if a given x there are more than one possible d

We assume regression with target y and L_2 (squared error loss)
Suppose an example $< X, y >$ where the true function is Gaussian noise with zero mean and stdev σ, in linear regression we fit $h(x) = wx + w_0$ such as we minimize the loss over training.
Because of hypothesis we will have a systematic prediction error and depending on dataset, the parameters w that we find will be different. If we take different samplings with 50 different fits we have a variance, and we can see how the fit changes.

1.5.1 Bias-Variance analysis

Given a data point x, what is the expected prediction error?
Assume data i.i.d from unique probability distribution P.
The goal is to compute for a point x:

$$\mathbb{E}_P[(y - h(x))^2] \tag{1.26}$$

The expectation is over all training sets drawn according to P. The $\mathbb{E}$ will be decomposed in those three components.

Recall of statistics

Let Z be a random variable with a probability distirbution $P(Z)$ and possible values $z_i, i = 1..l.$ the expected value or mean of Z and Variance are: By Variance lemma:

$$\begin{aligned} \mathbb{E}[Z] &= \sum_{i=1}^{l} z_i P(Z = z_i) \\ Var[Z] &= \sum_{i=1}^{l} (z_i - \mathbb{E}[Z])^2 P(Z = z_i) \end{aligned} \tag{1.27}$$

$$Var[Z] = \mathbb{E}[Z^2] - \mathbb{E}[Z]^2$$

Decomposition

$$\begin{aligned} &\mathbb{E}_P[h(x)2 - 2yx + y^2] \\ &\mathbb{E}_P[h(x)^2] - 2\mathbb{E}_P[y]\mathbb{E}_P[h(x)] + \mathbb{E}_P[y^2] \end{aligned} \tag{1.28}$$

We can call $\overline{h}(x)$ to denote the mean prediction of the hypotesis at x, when h is trained with data from P, using variance lemma we have:

$$\mathbb{E}_P[h(x)^2] = \mathbb{E}_P[h(x) - \overline{h}(x)]^2 + \overline{h}(x)^2 \tag{1.29}$$

For y is similar...
We consider indipendent the two expectation, the one of y and the other of $h(x)$, but this is the case, y is constant when you fix x but it's

not dependant on $h(x)$.
Putting togheter we have:

$$\begin{aligned} (variance) : \mathbb{E}_P[h(x)^2] - \overline{h}(x)^2 + \\ (bias)^2 : (\overline{h}(x) - f(x))^2 + \\ (noise)^2 : \mathbb{E}_P[y - f(x))^2] = \\ = Var[h(x)] + (Bias[h(x)])^2 + Var[\epsilon^2] = \\ = Var[h(x)] + (Bias[h(x)])^2 + \sigma^2 \end{aligned} \tag{1.30}$$

And our Expected prediction error is: $Variance + Bias^2 + Noise^2$
Maybe now is more clear the role of each:

- Bias: $[\overline{h}(x) - f(x)]^2$ systematic error, how you perform for that point in the mean, considering many hypotesis with respect to the target function squared. If too small h-space too rigid model (line is no able to approximate a curve)

- Variance: $\mathbb{E}_P[(h(x) - \overline{h}(x))^2]$ quantify variability, you can have different realization and in the mean you have this one, if you have a very rigid model this doesn't change is low or zero, if the hypotesis space is flexible will change and the value will increase

- Noise: even optimal solution could be wrong. We have a tolerance on responce σ

We want a tradeoff of the bias-variance, not too flexible not too rigid, reducing variance make rigid model that increase the bias (underfitting, overfitting recall that).
Low bias, Low variance is like to hit bulls eye in the center of the target. High bias and low variance (sort of underfitting) make you hit almost the same spot that could be the wrong one maybe not a lot of points, high variance and low bias (sort of overfitting) creates a spread around the target (bullseye) but rarely hit it. And high bias and variance is a complete mess. This not perfect for ML but gives a vague idea.

1.5.2 Bias-Variance and Regularization

Remember the Loss is:

$$\mathcal{L}(w) = \sum_p (d_p - o(x_p))^2 + \lambda \|w\|^2 \tag{1.31}$$

And varying the λ we can have complex solution with less regularization and more simple solution with high one. And example is using RBF network with LMS.
So finally if we over-regularize a model (large λ) we get high bias and if we under-regularize it we have high variance, again we want to find a trade-off

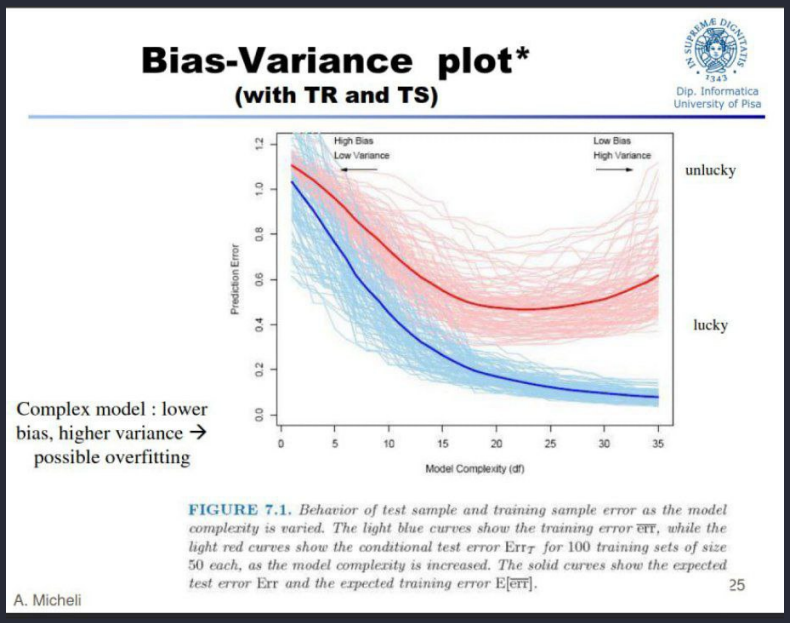

Figure 1.1: Bias-Variance trade-off

What if don't like the result, but if the mean is good... take the average! These approach are called ensables, popular nowadays

Chapter 2

Deep Learning

2.0.1 Inductive bias arguments

Taken from DLBook.
Choosing deep model encode the fact that we want to learn composition of simpler functions. Or a nested factor of variation. If our task match the bias of course the deep shape of the learner is suitable, sometimes generalization is better due to the many layers. The structure of images is a composition of sub-graphical parts and structure of language is text, speech, also music and new filds under discovering.
Discussing about deep networks approach we have a lot of arguments like Curse of Dimensionality: (number of example needed in high dim spaces) or Manifold learning : probability distribution for real images and text concentrated in a manifold using only a fraction of the volume all the possible random image and text, and we should be able to exploit it, image a box with each points a possible image, all the random are scattered in square, but the real image are relly narrow subcases hidden in a certain "corner" of this box.

Curse of dimensionality and Bias

Like K-NN, local kernels, but also decision tree work locally (local approximate) and local smootheness assumption is not enough also they need many example to cover the space (if high dimensional), to distinguish $O(K)$ region in input space, they require $O(K)$ examples.

Due to curse of dim this number if we make more assumptions on the data distribution, aka inductive bias we have an exponential number of regions in k to generalize non locally.
Instead deep learning frameworks choose a general inductive bias, we assume that data are generated by composition of factors and features, this allow us to achieve potential exponential gain between number of region between number of examples and number of regions, to generalize non-locally, and use less examples. There are some issues in the architecture, so how many layer or units to use? This is model selection issue, and this is costly! With many deep networks experience help with finding a better architecture.

2.0.2 Representation learning

"Representation learning is a set of methods that allows machine to be fed with raw data and to automatically discover the representation needed for detection or classification" - LeCun, Bengio

DL are representation learning methods with multiple level of representation, but the concept is more abstract and it can be applied for many models, also NN in general.
Representation learning or feature learnign is referred to data as images or text strings.
Many information processing task can be easy or difficult depending on how information is presented, in ML a good representation is one that makes the learnign task easier, manual design of feature is difficult, (design the ϕ of LBE manually), instead we use supervised learning in NN MLP lead to automatic representation at every hidden layer on properties that make the output layer taks easier. In DL we do it in mutiple level, how do we exploit it?

- Pretraining approaches: greedy layer-wise unsupervised pretraining was first approach to make possible deep supervised network in 2006, you take the learned weights on the shape of input distribution per layer, this makes easier the whole network training, instead end-to-end training from output to lower layers that use gradient descent that was costful at the time, this is the idea behind Autoencoders.
  Why this help? Because is a good initalization strategy, give some regularization in term of discovering feature that simplified unsupervised learning process and finally reduce the variance of the estimation training process.

- Transfer learning we transfer the knowledge from a model to another model to improve it, we assume that some features are useful for different tasks, from autoencoder we extract knowledge e use for a classifier, we can use a trained model for another task, same inputs but changing che target (multitask learning). Or change input domain but share the features eg sentiment analysis where you share similar characteristics or take advantage from larger models of pre-trained dataset.

Autoencoders

An autoencoder is a neural network that is trained to attempt to copy its input to its output, image a vector of input an hidden layer and output layer that has the same dimension of input, the task is learn to reproduce the original input.
But we're not copying the input to output, it more than that. The internal hidden layer describe the code used to represent the input.
So we have a net divided in two part an encoder $h = f(x)$ and a decoder that reconstruct $r = g(h)$ the input.

- **Undercomplete** Smaller hidden layer than input forces to capture most salient feature of training data. eg you obtain PCA (principal component analysis) by linear decoder and LMS approaches

- **Overcomplete** hidden layer bigger than input but we use regularization to make constraints of sparsity of representation, so

you force this sparness and you gain robustness to nois and other properties, this is also called **denoising autoencoder or auto-associative memories**.
On denoising: A good representation is one that can be obtained robustly from a corrupted input and that will be useful for recovering the corresponding clean output
We can start with very corrupted and cycle in a denoiser to get a cleaner image for example. But our use of autoencoders is to do pretraining, layer wise pre-training for unsupervised can exploit the unlabeled data, some autoencoders like RBM (Restricted Boltzmann Machines) have many layers.

The algorithm for training an autoencoder is: (Called stacked NN denoising autoencoders from bengio et all 2007)

- Train the first layer as autoassociator to minimize the reconstruction error
- The hidden units outputs in the autoassociator are used as input for another layer, also trained to be an autoassociator(unsupervised), so you remove the decoder part and insert the encoder, and you build the first layer.
- Iterate as in the previous step to add layers
- Take the last hidden layer output as input to a supervised layer and initialized it's parameters (randomly or by supervised training, keep the rest of the network fixed)
- Fine-tune all parameters of this architecture with respect to the supervised criterion

Needed?
Pretraining can yield improvements for some task in NLP, also allows to exploit large texts to learn distributed representation of words, it allowed to start DL, but it's difficult to be managed effect of hyper-parameters divided in two phases and in DLbook chapter 15, it says "Today unsupervised pretraining is abandoned" because of some other techniques are sufficient to make good training: end to end over deep network now is more doable. Still an open field!

AlexNet

AlexNet is a CNN trained on a large dataset from the ImageNet Dataset with 1000 objects categories, (keyboard, mouse, pencil...). This model has learned a lot of rich feature representation for wide range of images, you can change the MLP at the end replacing the last part of this net to fit your new specific task using a small number of images for training the new task, then you can fine tune the AlexNet CNN to perform classification on a new collection of images. There are many pre-trained models like ResNet, VGG, etc...

2.0.3 Changing design of Networks for different applications

Picture again the MNIST dataset (16x16 8-bit grayscale images) and we want to classify in 10 classes. MNIST is like "drosophila" benchmark of machine learning!
How do we shape the design of a model

2.1 Distributed Representations

Again a clim from LeCun from paper Deep Learning:

> *In a distributed representation their elements (features) are not mutually exclusive and their many configuration corresponds to the variations in the observed data*

Deep Learning methods explit distr. representation with multiple level of representation, the concept is more abstract and it can be applied for many models, also NN in general or prboabilistic graphical models. So not necessary with a deep architecture
Symbolic and distributed are different because in symbolic we have usually one hot encoding and each feature is mutually exclusive with a distance of root 2, while in distributed representation we have a vector of real numbers and each of these share similarity with others so they can share features and the distance reflect actually the meaning (our

knowledge or such learning).

- Richer and smoother representation
- Shared attribute allow to generalize better different concepts

2.1.1 Internal representation and input

Symbolic vs distributed refers to the input or to the hidden layers representation?
Learning act on the internal representation, but in general distributed is used in input if you have enough background knowledge to help the model, otherwise automatically via learning if you can use the distr. rep for the hidden layers.
We want to distinguish model able to exploit this representation (internally or not).
Less neuron because if you have 4 different concept to represent like blue and red cars and bikes, you can use 2 neurons, 1 for car and bike and the other for the colors, because distinguishing the redness in both the object is like the same task. So we're sharing the task that are common for identify our concepts.
With distributed representation we can have n features and k values to describe k^n different concepts, for example if $k = 2$ we have 2^n configuration versus the symbolic representation that can represent only n (with one-hot). Other ML models rely on non-distributed representation, like:

- K-NN 1 or few prototypes for each input they cannot sahre info with the training data and input components cannot be controlled to be similar to each other
- RBF Kernels reduce to K-NN as we saw in SVM lecture, because in the phase of "prediction" with RBF we for classify a point take the closest neighbors in the new feature space
- Mixture of Gaussian similar

- Decision Trees only one leaf is activated and the path from the root to classify a pattern they cannot share

With distr. representation we can cover an exponential number of subregion with $O(2^n)$ if you encode binary feature.

2.1.2 Disentangling concepts

Take the example we said before with bike and cars, we share the column of red between car and bike, sharing a feature we can disentangle the concept of being red.
Disentangling the two concepts by distributed representation we can learn about distinction between car and bike or the color, without having to learn all the combination!
From the point of view of statistical separability this make easier to generalize configuration unseen during training.
For example we can use distributed representation for words, symbolic is onehot encode words with symbles that have same dimension of vocabulary, then we can se what happens if we represent them in distributed representation-way. For example words like "dog" and "cat" would have a lot of shared features, against words like "book" that share nothing with them...
This is called Word Embedding.

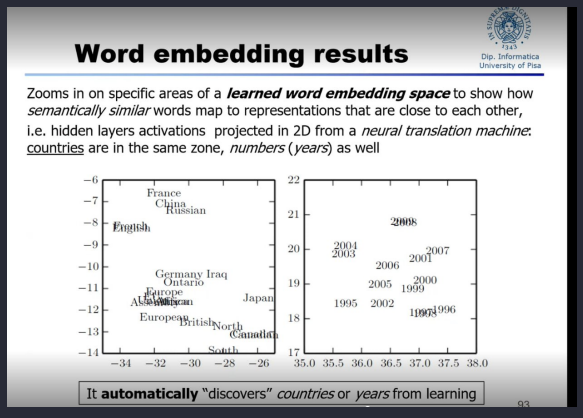

Figure 2.1: Word Embedding

One can zoom in areas of learned word embedding space to show how semantically similar words map to representation that are close to each other, hidden layers activation projected in 2D from a neural translation machine: countries are in the same zone, and number of years as well! It automatically discovers country or years by learning.
Distributed representation of words are obtained by use of backpropagation to jointly learn a representation for each word and a funtion that predicts a target quantity such as the next words in a sequence, or whole sequence (for machine translation), in this way models not only learn grammar, because result must be similar it learns things that are semantically significant just by feeding text, without knowledge provided.
Issue: Why our semantic corresponds to what is learnt bu the deep NN?
Well, probably the language (or image) intrisically shows a hierarchical structure in itself and the deep model has the proper inductive bis to learn that structure in a distribuited way. For example real images and noisy images: The real images that we perceive are a small fraction of all the possible images. This is still open field of research, the professor said: to be honest, we do not know how or why.

2.1.3 Beyond Neural Networks

This debate on distributed representation is extendable to logic-inspired paradigms for cognition and neural network-inspired paradigms for cognition.
Due to this Deep learning has taken over the ML scenario in these years, also the symboli approach is being used less.
Critically approaching distributed representation is less easy to be interprdeted instead symbolic is a lot more easy to be interpreted!

Deep distributed representation exploit this trought many layers obtaining composing of different levels of abstraction and with a hierarchy of reused features, this can be a boost to efficiency so two different exponential advantages against not-distributed and shallow models).
Globally DL models learn a distributed representation of data, finding (or better disentangling) shared causal factors that generated different level of abstraction

2.1.4 Smoothness trough network

For a given number of parameters DN, impose more smoothness than shallow ones, each layer smooths the already smoothed surface by previous layer, deep networks seems to learn better, for the same number of total neurons.

Non overfitting puzzle

Seems that huge NN lack overfitting, why?

A main puzzle of deep networks revolves around absence of overfitting despite large overparametrization and capacity demonstrated by zero training error on randomly labeled data

So seems that we can't overfit this deep networks, there are many studies like this paper: Explaining the non-overfitting puzzle

- Gradient descent enforces a form of implicit regularization
- Convergence to the maximum margin solution
- Robustness wrt curse of dimensionality implications

And now two suprising theorems:

Theorem 4. *Even standard GD maximize L_2 margin withoutexplicit normalization or regularization) or implicitly controls the complexity through an "implicit" unit L_2 norm constant*

Theorem 5. *Variant of SGD (bath normalization and weight normalization) perform minimization equivalent to maximize a margin*

Also now the Deep Double descent phenomenon that occurs in CNNs, ResNet and trasformers, nets with millions and billions of parameters: performance improves then get worse (the U-shaped curve), but then improves again with increasing model size, data or training time. This effect is avoided with regularization. While this behavior appears to be

universal, we don't fully undestand why it happens, and view further study of this phenomenon as important research direction: "Effective model complexity" studies wrt numbers of data etc..

We can also talk about redundancy in the parametrization of deep models or compress the parameters without accuracy loss: "Deep compression: compressing deep neural network with pruning, trained quantization and huffman coding" paper. So maybe small networks are enough!

Lottery ticket hypotesis

Why we use large networks?
Frankle and Carbin answered with this hypotesis: Good performance depends on lucky initalization of one of more subnetworks in out big deep models, so large networks have exponentially more subnetworks, maybe we can find this subnetworks and prune all the useless part of the big one.
Testable hypotesis: retraining pruned network on same initialization should give similar performances! More on their paper: "The lottery ticket hypotesis: finding sparse, trainble neural networks" or "training pruned neural networks"
Also Randomization is important for radomly initalized networks with random weights.

Deep Learning technicalities

Many aspects can be discussed like different types of units: activation function, generative models, different pooling in CNN, recurrent NN (active research in UniPi-CIML) and different types of learning algorithms: approximate training, pre-training, semi-supervised learning, improved gradient descent (nesterov & others), dropout, external memory (neural turing machines) and combining with reiforcement learning approaches (AlphaGo). Finally GPU computing for ffaster calculations!

Techniques

So in general DN uses less units in each layer, less parameters and less training data to get a good generalization, but is difficult to optimize many layer, hence: we want methods to improved GD, regularization, better exploitation of data. But also larger dataset for real application, software infrastracture and HPC improvements! (GPUs)

For DL of layered NN:

- Pre-training approaches (for example in NLP word embedding)
- SGD with momentum, decay, minibatch or Adam etc...
- ReLu activation in hidden units
- Use of max log-likelyhood (cross-entropy) loss with softmax for output to avoid saturation/small gradients effects
- Regularization tech like ealy stopping dropout and batch-normalization.

Lets discuss some of them like Gradient issues: what happens when we backpropagate gradients in our network with many layers? If the weights are small the gradient gets multiplied with very small weights being shrinked and this is called vanishing gradient issue: this bring our network to learn nothing with each step, instead if weights are big the gradient grows aka Exploding Gradients (overflow in our gradients for example)

Clipping gradient: repetitive multiplication trough layers can introduce cliffs in the cost function.
Clipping if : $||g|| > v$ then we clip gradient like this $g = vg/||g||$, where g is gradient and v is norm threshold

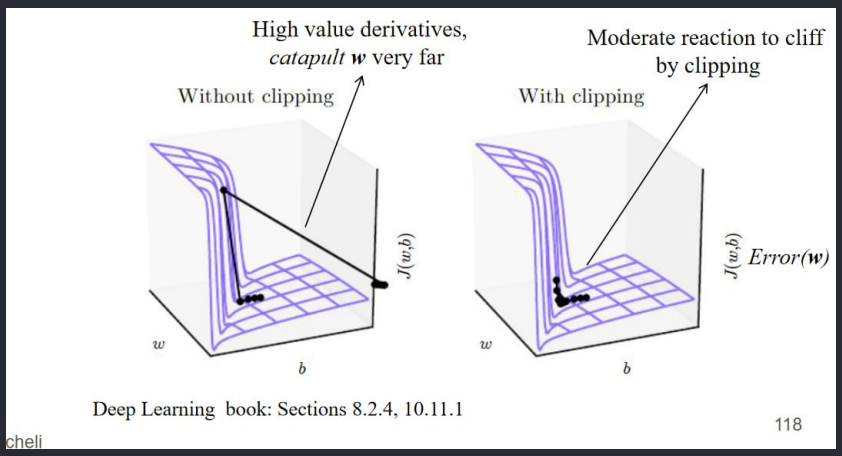

Figure 2.2: Gradient clipping

This can be a way to avoid gradients explosion, because we clip the gradient to a certain value, but this is not a good solution because we are losing information, in fact when we clip the gradient the "direction" of the descent will be different! A solution to this that mainteins direction is **Clipping by norm** but this brings other problems like making weights significantly smaller...
How can we implement it? Well: one option is to clip gradeint from a minibatch element-wise, just before parameter update, or clip the norm of the gradient before parameter update.

Instead when you have very small gradient (usually in the low layers), because of using activation function like tanh that gives gradient in range (-1,1) and when backpropagate computes gradients with chain rule multiplying a lot small numbers to compute the next gradeint of front layer descrease exponentiallywith a training very slow or even nullified by almost no changes.
Some approaches to deal are Rprop to short-cut connections or ReLu.

2.1.5 ReLu in DL

ReLu is a popular activation, with efficent gradient propagation wrt vanishing, efficient computation and faster and effective training in deep architectures but is non differentiable at zero

$$ReLU(x) = \begin{cases} x & x > 0 \\ 0 & x \leq 0 \end{cases} \tag{2.1}$$

Whis is useful in DL? Avoid saturation effect of sigmoidal function where gradient shrinks, but can't get gradient in zero, often in the implementation is assumed to be 0 or 1 that is a safe approximation, there are some tricks as well : start with positive nets eg. bias to a small, positive value like 0.1, this makes that relu units will be active for most inputs in traning set allowing derivative to go through.
Beyond ReLu: ELU or Leaky ReLu are other alternatives:

$$ELU(x) = \begin{cases} x & x > 0 \\ \alpha(e^x - 1) & x \leq 0 \end{cases} \tag{2.2}$$

$$LeakyReLu(x) = \begin{cases} x & x > 0 \\ \alpha x & x \leq 0 \end{cases} \tag{2.3}$$

Where neurons are not "turned off" but learns on the left part.

2.1.6 Batch Normalization

This technique is a method for NN optimization that normalizes each batch by calculating individual statistic such as mean and variance for each layer (adaptive reparametrization)
In pratice when we backpropagate we update all the layer simultaneously, when we make the update, in deep models, unexpected results can happen because many function composed togheter are changed, with Normalize each matrix [databatch x activation of units] with mean and variance then include it in backprop, normalizing input is standard approach and BN help making data flow intermediate layer of a

network, it has also regularization effect, achieve faster learning and accuracy for DL

2.2 Unsupervised Learning and Clustering

Unsupervised learning consist in a TR of an unlabled data $< x >$, clustering find natural grouping in set of data, we can represent our data in lower dimensional space with visualization, preprocessing like PCA, Multi-dimensional scaling and Independant Components Analysis.

2.2.1 Clustering

Clustering is a method of unsupervised learning, it is a method of grouping data in a way that data in the same group are more similar to each other than to data in other groups. We will mainly focus on clustering and vector quantization

Vector Quantization

Vector quantization encode data manifold eg. a submanifold $V \subseteq R^n$, utilizing only a fine set $w = (w_1, ..., w_k)$ of reference or codebook vectors $w_i \in R^n$, $i = 1, ...K$. A data vector $x \in V$ is described by the best matching or "winning" reference vector $w_{i^*(x)}$, for which the distortion error $d(x, w_{i^*(x)})$ is minimal.

This procedure divide the manifold V into a number of subregions.

$$V_i = \{x \in V s.t. ||x - w_i|| \leq ||x - w_j||, \forall j\} \tag{2.4}$$

This is called Voronoi polyhedra, out of which each data vector x is described by corresponding reference vector $w_{i^*(x)}$. Common usage of this problem: Optimal vector quantization is NP-Complete, this is popular in telecommunication to find the better position for an Antenna to cover better the space, also quantization in music. Example is Quantization in 1D, quantization in digital signal processing aprroximate a

continuous range of values or a large set of descrete one, by a relatively small set of values which can still take on continuous range discrete symbols so we have a quantization or distorsion error. We use VQ algorithms for clustering.

Our goal is to find optimal partitioning of unknown disribution in x-space into regions (clusters) approximated by a cluster center, a set of vector quantizers: $\mathbf{x} \mapsto c(x)$ or $w_{i^*(x)}$, we consider our squared distortion error and evaluate in the point x_i like this: $d(x_i, c(x_i) = ||x_i - c(x_i)||)$

$$E = \int f(d(x, w_{i^*(x)}))p(x)dx = \int ||x - w_{i^*(x)}||^2 p(x)dx \tag{2.5}$$

Where p(x) is the probability distribution

And discrete version is:

$$E = \sum_{i}^{l} \sum_{j}^{K} ||x_i - w_j||^2 \delta_{winner}(i, j) \tag{2.6}$$

$$\delta_{winner}(i, j) = \begin{cases} 1 & i^*(x_i) = j \\ 0 & otherwise \end{cases}$$

Now we want to minimize this E, vector quantization, we want the set of reference vectors that minimize E, this is our solution of VQ problem.

Note: The integrand of E is not continuously differentiable, our "winner" have discrete changes w.r.t x, however we computes the derivative locally fixing x on a voronoi cell changes w.r.t x, however we computes the derivative locally fixing x on a voronoi cell.

Online K-means

Taking derivatives of equation 2.6 w.r.t w_j we get the learning rule of VQ, LLoyd and MacQuee's well now K-means clustering algorithm (on line version) for any x_i

The learing rule will be:

$$\Delta w_{i^*} = \eta \delta_{winner}(i, i^*)(x_i - w_{i^*}) \tag{2.7}$$

Note that we only change winner, adapting to x

Proof. Show the derivation of the E wrt w_j

$Property : ||x - y||^2 = (x - y)^T(x - y) = x^T x - 2x^T y + y^T y$
(2.8) □

K-means batch algorithm

Also knows as LBG: Linde, Buzo, Gray

- Choose k cluster center to coincide with the k chosen pattern or k randomly defined points inside the hypervolume containing pattern
- Assign each pattern to the nearest cluster center (winner)
- Recompute the cluster center as the mean of the patterns assigned to it
- Repeat from step 2 until convergence, certain minimal decrease of error or no reassign of patterns to new cluster centers

For example the point two of the algorithms performs the assigniment to the winner in this way: $i*(x) = argmin||x - w_i||^2$ where computing the euclidean distance. And then the third step moves the centroid toward the mean in this way: $w_i = \frac{1}{|cluster_i|} \sum_{j:x_j \in cluster_i} x_j$. Note that we use the squared of norm 2 because we get rid of the root, so simple sum of distances

Now k-means is the simplest and most commonly used algorithm exploying squared error criteria, but you must provide K number of cluster and local minima of E make the method dependant on initialization so we must run more than once the algorithm, but is efficient so we can do it. Also it works well for compact and hyperspherical cluster but not for other type of cluster, an example could be two cluster that are divided in a spiral way, also K-means doesn't allow to project data in lower dimensional space, so no visual properties, in the next approach we will have unordered map and the indexing of these w can be done in arbitrary way, we will sovle this issue with next approach. We want to go in lower dimensional spaces in the case of unsupervised learning, usually 2D, going in higher dimension is useless and also make the work more hard!

Softmax

To avoid confinement to local minima a common approach is use "soft-max" adaptation rule, so instead of considering only the winner we consider all surrounding reference vectors, according to distance. Maybe with a step size that decrease with distance, maximum-entropy clustering (uses gaussian distance), a strategy used in NN is the **Kohonen self organizing maps** where proximity among reference vector is defined on a ordered map, aka Neural grid where reference vectors are arranged and associated to units. :

2.3 Recurrent Neural Networks

In the IEEE they called them "The nets that remembers" with short-term memory recurrent nets gain amazing abilities, basically a recurrent nets has some connection between hidden layer units that feedback on theirselves or with each other. Feedforward network flows information from input to output, instead RNN is based on a different architecture based on **Feedback loops** in the network topology, this loops provides dynamical properties to these systems, letting them to have "memory states" of past computation of the model so we can better represent or process sequenced data. These type of nets are more neurobiologically plausible due to the fact that biological nets are recurrent net.

The idea at the base is the parameters sharing, that we can also compare to the convolution that allows to share parameters but working on the Neighbouring memebers of the input, the idea of sharing is to use a kernel at each time step, but RNN share in different way.Each member of the output is a function of the previous members of the output. Each member of the output is produced using the same update rule applied to the previous outputs. This recurrent formulation results in the sharing of parameters through a very deep computational graph.

2.3.1 Why Sequantial Data are good?

Well when we want to work on data that are dependant on the history of their past inputs, like time series, or when the domain is composed by different varying sequences that is useful for Signal Processing, Language processing, Vision and Reasoning (temporal events like problems for self-driving cars), temporal series and Bionformatic (genomics and proteomics). Some nice application of RNN, can be: music composition and text/speech generation! Moreover we go from flat to structured data: Sequences, tree-like, multi relational, graphs and more.
Transduction is the process of mapping an input sequence to an output sequence, for example in speech recognition we map a sequence of audio to a sequence of words, in language translation we map a sequence of words to a sequence of words in another language, in music generation we map a sequence of notes to a sequence of notes, in text generation we map a sequence of words to a sequence of words. In all these cases we have a sequence of input and a sequence of output, so we have a transduction problem.
Let be $x^{(t)}$ our data with our time step t ranging from 1 to τ RNN operates over a "minibatch" of these sequences with difference τ for each member of this minibatch.

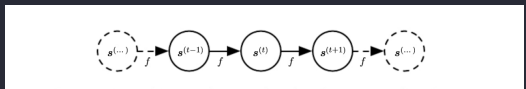

Figure 10.1: The classical dynamical system described by equation 10.1, illustrated as an unfolded computational graph. Each node represents the state at some time t, and the function f maps the state at t to the state at $t+1$. The same parameters (the same value of $\boldsymbol{\theta}$ used to parametrize f) are used for all time steps.

Figure 2.3: RNN diagram

2.3.2 Memory in RNN

When discussing about memory we want to make dependant our output from previous inputs, we can have Input Delay Neural Network (IDNN) since delays are in the NN connections and act as input memory, in these case we have a Finite-size shift register (a sliding window) in one dimension if you want, CNN extends this in 2D images

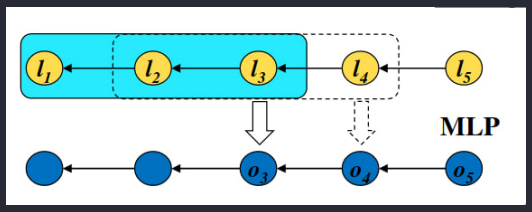

Figure 2.4: IDNN diagram

Recurrent Units

These unit use current input and state information to compute the next state, they get feedback loops

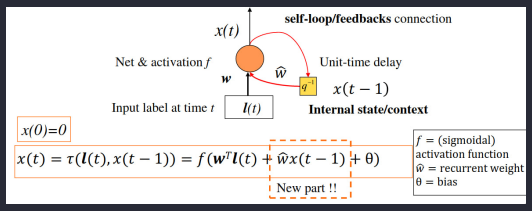

Figure 2.5: Recurrent Unit diagram

As exercise:

- Exercise: realize a 1-unit RNN (find the values for the 2 weights w e w-hat) which ouputs the sum of "1" received in input so far. Input stream: 1 and 0. Drawn the flow of input/state/ouput values time-by-time for an example (101101)

As we saw earlier our τ aka state transition function is realized by the NN, our state contains a summary of the past inputs, also notice that state have values $x \in \mathbb{R}$, and our $x(t)$ can be a set of states that compose a full network.

2.3.3 Turing Machine and RNN

Given this set of equation describing a RNN:

$$\begin{aligned} \mathbf{a}^{(t)} &= \mathbf{b} + \mathbf{W}\mathbf{x}^{(t)} + \mathbf{U}\mathbf{x}^{(t-1)} \\ \mathbf{h}^{(t)} &= tanh(\mathbf{a}^{(t)}) \\ \mathbf{o}^{(t)} &= \mathbf{c} + \mathbf{V}\mathbf{h}^{(t)} \hat{\mathbf{y}}^{(t)} = softmax(\mathbf{o}^{(t)}) \end{aligned} \tag{2.9}$$

where parameters are the bias vector $\mathbf{b}$, the weight matrices $\mathbf{W}$, $\mathbf{U}$, $\mathbf{V}$, respectively for input-to-hidden, hidden-to-hidden and hidden-to-output connections and the output bias vector $\mathbf{c}$. This is a RNN that maps an input sequence to an output sequence of same length. The RNN (finite net) can compute any computable function a Turing Machine can, the output can be read from the RNN after a number of time steps that is asymptotically linear in the number of time steps used by the Turing machine and asymptotically linear in the length of the input, RNN when used as turing machine take binary inputs and it's output must be discretized to provide binary output, the same net that can simulate a TM is sufficient for all sort of problems. The equation we've seen before are just how a RNN can do forward propagation.

So we have two main properties of RNN:

- RNN are Universal Approximators
- RNN are Turing Machines
- Another instresting point is: RNN seen as non-autonomous dynamical non-linear systems related to **Chaos Theory** and **Fractal theory**

2.3.4 Proving RNN and TM are equivalent

To prove the equivalence between the two it needs to be shown that the internal state of the network, or the contents of the network nodes, can be

identified with the program states, and the succession of the network states corresponds to the program flow.

Proof. □

2.3.5 More properties

As we saw RNN also simple ones are very powerful architectures and are based on these assumptions:

- **Causality**: a system is causal if the output at time t_0 or node v depends only on the input up to time $t < t_0$ and not on the input after time t. This is quite essential (necessary and sufficient) for internal state.
- **Stationarity**: time invariance after model training, i.e. the τ function is independent on node v of sequence and τ is the same in every t, this is useful for processing data with different lengths with a fixed size model.
- **Adaptivity**: transition function are realizied by NN, with weights as free parameters, so they are still learnt from data.

2.3.6 Unfolding

Unfolding is the process of unroll the model in the time dimension, one can build a Feedforward MLP called encoding network on the k given steps that is equivalent to the RNN, so can have 1 model for each step, these models share some weights and trought the econdiging network we can exploit this form of unfolding to apply our new backpropagation.

Backpropagation in RNN These learning algorithms must encode the transition developed by the model for each step, we have two main approaches:

- **Backpropagation through time (BPTT)**:
- **Real time recurrent learning (RTRL)**:

These two compute in different styles the gradients values of the output error across an unfolded net over time. A common problem in these cases is the **vanishing gradient problem** we talked some time ago. Storing long term dependancies in this graph is difficult.

2.3.7 Advanced models

Research is rapidly growing and there are different models: LSTM (Long short term memory) that try to solve gradient vanishing by using gated units capable of select past gradient GRU (gated recurrent units) basically simplified LSTM , BRNN (Bidirectional RNN) that consider left and right context, DRNN (Deep RNN), SRNN (Stacked RNN). Other way to address the vanishing problem are Hessian-Free optimizers and pre-training techniques.
More related approaches are SOM and HMM (Hidden Markov Models) that are used for modeling probability to transition from states, Randomized NN (ESN, LSM), Distance based models (string matching) Kernel for strings, Grammatical inference and IPL.
Rememeber the lesson about randomized networks? We can exploit the state machine to encode sequences and then use it for learning ouput mapping, using random machines (untrained) for random connected networks.
Reservoir computing and ESN or Liquid state machine class of RNN have very efficient (no hidden recurrent units) structures and capabilities to solve well task under some specific conditions, also the ability to intrinstically discriminate among different inputs sequences in a suffix-based fashion without recurrent parameters.

2.3.8 Echo state networks

This is an emergin paradigm for modeling recurrent NNs

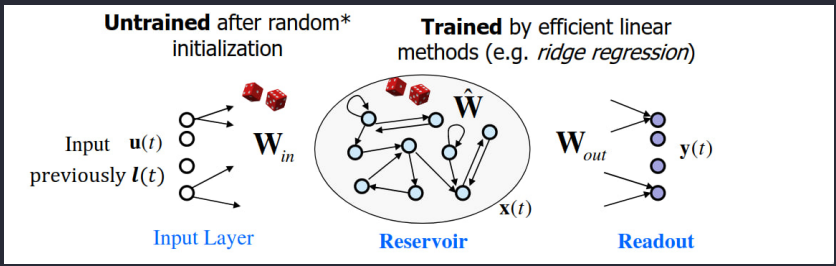

Figure 2.6: Echo state network

The idea is to have an input layer modeled after a reservoir of randomly connected recurrent units. And a readout part that is composed by feed-forward readout of linear units.
One way to think about these reservoir computing recurrent nets is that they are similar to kernel machines, mapping to an arbitrary lenght sequence (history of inputs until time t) to a fixed-length vector, the recurrent state $h^{(t)}$ on which a linear predictor (linear regressor) can be applied to easily solve the problem. The training may then be easily designed to be a convex function. For example if the output consist of linear regression from from hidden to output target and the training loss is MSE, then is convex and can be solved with simple methods.

Echo state property: contractivity of the state transition functions (limiting spectral radius of reservoir weights, i.e. stability of the dynamical system). Recurrent state asymptotically depend (echo) only on the history of the inputs

The important question is therefore: how do we set the input and recurrent weights so that a rich set of histories can be represented in the recurrent neural network state? The answer proposed in the reservoir computing literature is to view the recurrent net as a dynamical system, and set the input and recurrent weights such that the dynamical system is near the edge of stability.

2.3.9 Toward structured domains

Can we extend recurrent approach to rooted trees?
Well this is called recurrent neural networks, one can encode the unfolding process trought structure like tree, if we change tree structure we change the encoding as well.

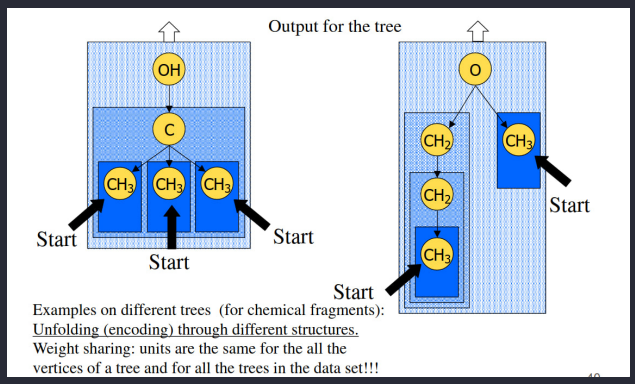

Figure 2.7: Tree RNN

2.4 Structured Domains

As we said earlier we can represent with graphs in this way:

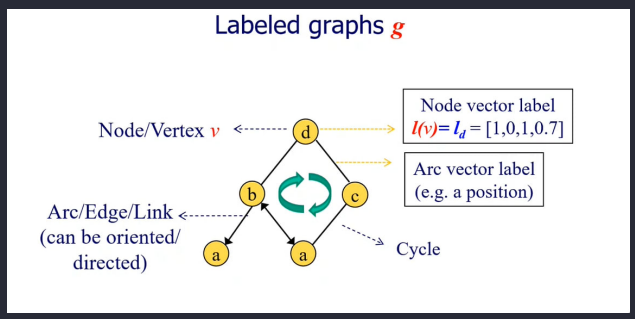

Figure 2.8: Graph RNN

The problem: There's no systematic way to extract features or metrics relationships from SD, we're talking about representational learning instance

and is difficult to say what is important in a certain domain, maybe how many connection a node has with others...
Feature based representation are incomplete (representation problem) But using adiacent matrix for represent something you have other issues, overdimension of matrix, if you have big and small graphs (padding), alignment problem on graphs: rotated graphs are different represented! (topological order)
The ability treat the proper inherent nature of input data is the key for goood ML applications

So instead to encode our data so give that to our model that compute it, we can bring "the model to the data", so we don't lose information that are lost in converting data in matrices/vectors

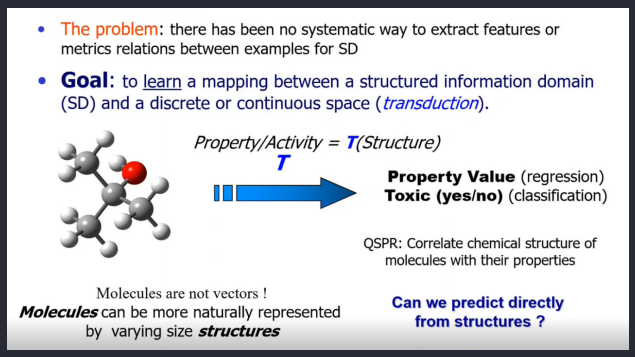

Figure 2.9: Molecules representation

So we want to learn a mapping from SD to discrete or continuous space via transduction. A more general transduction can be how we represent the input sd : Isomorphic transduction or not, for example structure-to-structure or structure-to-scalar.

2.4.1 Recurrent approach from sequence to trees

RNN can be seen as state transition systems with free-parameters, recursive NNs can be processed with trees from bottom up we visit the tree, for graphs this is an issue because they have to wait and can be troubles of cycles getting stuck!

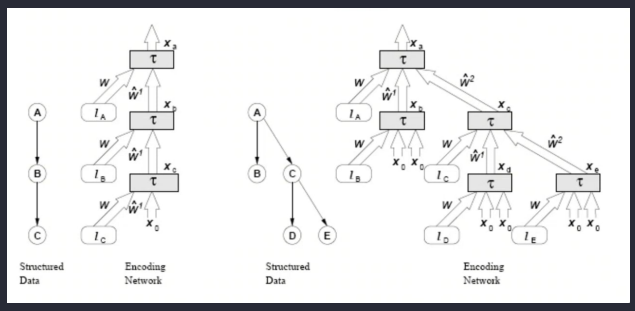

Figure 2.10: Tree and sequences

In recurrent nets the concepts of causality, Stationarity and Adaptivity are different, we don't work with time series, so transduction of vertex v depends only to the descendandts (the next recursive call)

2.4.2 Recursive Cascade Correlation

Adding a new layer for each training step, interleaving output and hidden units). The processed is the same but here you add a recursive unit that process the trees, you build a deep recursive model.

2.4.3 Reservoir computing and Trees

TreeESN is extremely efficient way to model RecNNs with randomized approaches and extending to structured data

Also DeepTreeESN have an hierarchical absraction both trought input and architectural layers, these store progressively more deep reservoir representations of trees.

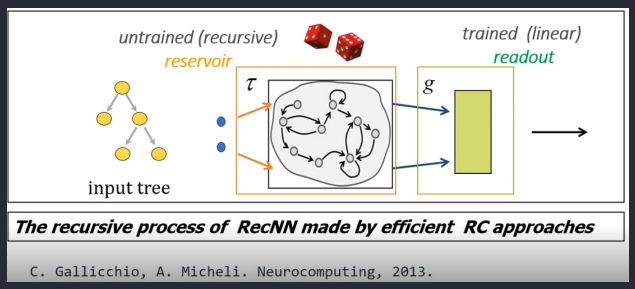

Figure 2.11: Tree ESN

2.4.4 Self-organizing maps

We can transfer the recursive idea to unsupervised learning with no preprocessing, with recursively embedding nodes on SOM, with bottom-up encoding process.

2.4.5 Hidden Tree Markov Models

Extends HMM to trees exploiting recursive approach, the main difference is: instead of modeling our transition by a neural network we model by a probability distribution of course we have the assumption that current change depends on the chain rule: decomposing a probability:

$$P(x_{t+1}|x_t, x_{t-1}, ..., x_1) = P(x_{t+1}|x_t)P(x_t|x_{t-1}, ..., x_1) \tag{2.10}$$