Statistical Learning, Fall 2024-5

# Class notes 12

## Kernel methods

The general paradigm we have discussed, given modeling problem with $x \in \mathbb{R}^p$ low dimensional:

- Embed $x \to h(x) \in \mathbb{R}^q$ with $q >> p$.
- Fit a (possible linear model) in the high dimension $\hat{f}(x) = \sum_{j=1}^{q} h_j(x)\hat{\beta}_j$.
- Challenges:
  - Computational: how to fit in high dimension
  - Statistical: how to regularize in high dimension

**Examples:**

- Boosting:
  - Model space: all trees of given size
  - Computational trick: coordinate descent via gradient boosting
  - Regularization: sort of lasso (not discussed in class)
- DNN:
  - Model space: Not a linear model but linear combination of non-linear transformation of linear combinations...
  - Computational tricks: (stochastic) gradient descent,
  - Regularization: sort of ridge (gradient descent $\approx$ ridge, similarly dropout

Now we will discuss perhaps the primary example of this thinking, which was hugely important in ML in the past, lost some of its glamour: Kernel methods including (but not limited to) kernel SVM. We can think of the basic idea the same way, except now $x \to h(x)$ where $h_1, ...h_q$ (possibly $q = \infty$) is a basis of a Reproducing kernel Hilbert functional space (RKHS) $\mathcal{H}_K$. The space is defined indirectly through the kernel function

$$K(\cdot,\cdot) : \mathbb{R}^p \times \mathbb{R}^p \to \mathbb{R} \text{ such that: } K(x,y) = \langle h(x), h(y) \rangle = \sum_{j=1}^{q} h_j(x)h_j(y).$$

We also naturally define for a function in $\mathcal{H}_K$, $f = \sum_j \beta_j h_j$, we naturally define $\|f\|^2_{\mathcal{H}_K} = \sum_j \beta_j^2$.

**Kernel examples:**

1. Linear Kernel ($q = p$): $K(x, y) = \langle x, y \rangle$. Here $\mathcal{H}_K$ is simply linear functions.

2. Polynomial kernel: $K_d(x, y) = (1 + x^t y)^d$. Here $q = \binom{p+d}{p}$ all polynomials in $x_j, y_j$ up to degree $d$.

3. RBF (Gaussian) kernel: $K_\sigma(x, y) = \exp\left(-\|x - y\|^2/(2\sigma^2)\right)$. Here $q = \infty$ and we usually don't think about $h_1, ...$ explicitly, only about the kernel as measuring distance:

   - When $\sigma$ is small, the kernel $K(x, \cdot)$ is very tight around $x$
   - When $\sigma$ is big, the kernel $K(x, \cdot)$ becomes very spread and $K(x, y)$ remains big for $\|x - y\|$ big

   Since $q = \infty$ the function space $\mathcal{H}_K$ contains all nicely behaved functions regardless of $\sigma$, however we will see that the different nature of the kernel will play a role in model building (i.e. selecting among the functions in $\mathcal{H}_K$) through regularization.

## Kernel machines

The Hilbert space comes with a norm attached and therefore a natural regularization term that controls that norm. Given a loss function our problem is:

$$\hat{f}_\lambda = \arg\min_{f \in \mathcal{H}_K} \sum_{i=1}^{n} L(y_i, f(x_i)) + \lambda \|f\|^2_{\mathcal{H}_K}.$$

We see here that the regularization term is where the specific kernel plays an important role: how functions in $\mathcal{H}_K$ are prioritized for fitting.

The most important result in this area is *the Representer theorem* (Kimmeldorf and Wahba 1970):
The optimal solution to the kernel regression problem above has the form:

$$\hat{f}_\lambda = \sum_{i=1}^{n} \alpha_i K(x_i, \cdot)\ , \quad \|\hat{f}_\lambda\|^2_{\mathcal{H}_K} = \alpha^T K \alpha, \quad \text{where: } K_{ij} = K(x_i, x_j).$$

Thus we get that we can solve the problem in the $n$ dimensional basis of the columns of $K$:

$$\hat{f}_\lambda = \arg\min_{\alpha} \sum_{i=1}^{n} L(y_i, \sum_{j=1}^{n} \alpha_j K(x_i, x_j)) + \lambda \alpha^T K \alpha.$$

For squared loss this *Kernal linear regression* problem can be nicely written:

$$\hat{f}_\lambda = \sum \hat{\alpha}_i K(x_i, \cdot) \quad \text{where: } \hat{\alpha} = \arg\min_{\alpha} \|\mathbb{Y} - K\alpha\|^2 + \lambda \alpha^T K \alpha,$$

a "generalized ridge regression" problem, with an algebraic solution:

$$\hat{\alpha} = (K + \lambda I_n)^{-1} \mathbb{Y}.$$

Now we can interpret what some of our kernels do in this context:

- Linear kernel: $K = \mathbb{X}\mathbb{X}^T$ and therefore $\hat{\alpha} = (\mathbb{X}\mathbb{X}^T + \lambda I_n)^{-1}\mathbb{Y}$. In this case we can easily show:

$$K\hat{\alpha} = \mathbb{X}\mathbb{X}^T(\mathbb{X}\mathbb{X}^T + \lambda I_n)^{-1}\mathbb{Y} = \mathbb{X}(\mathbb{X}^T\mathbb{X} + \lambda I_p)^{-1}\mathbb{X}^T\mathbb{Y} = \mathbb{X}\hat{\beta}_\lambda,$$

  the solution is the same as regular ridge regression!

- RBF Kernel with small $\sigma$:

$$K(x_i, x_j) = \exp\left(-\|x_i - x_j\|^2/(2\sigma^2)\right) \approx 0 \text{ when } x_i \neq x_j \ , \text{ while } K(x_i, x_i) = 1.$$

  Therefore the kernel regression problem is very much like penalized k-NN:

$$\|\mathbb{Y} - K\alpha\|^2 + \lambda\alpha^T K\alpha \approx \|\mathbb{Y} - \alpha\|^2 + \lambda\alpha^T\alpha,$$

  the solution to this approximate problem is easily seen to be simple shrinkage of $1 - NN$: $\hat{\alpha}_i = \frac{Y_i}{1+\lambda}$.

The most important kernel machine was the one using the hinge loss (kernel SVM):

$$L(y, \hat{y}) = (1 - y\hat{y})_+,$$

and recall that we discussed how the sparsity of the solution $\hat{\alpha}$ (support vectors) helps in computing and finding solution.

For regression, the ML crowd who like loss functions that zero many $\hat{\alpha}$ came up with the $\epsilon$-support vector regression loss, which is absolute loss with a *dontcare* region in the middle:

$$L(y, \hat{y}) = (|y - \hat{y}| - \epsilon)_+,$$

Now we can also describe kernel methods in the high dimensional modeling framework:

- Model space: all functions in the RKHS
- Computational tricks: representer theorem, giving a problem of dimension $n$; Sparsity of solution $\hat{\alpha}$ ("support vectors") when selecting appropriate loss functions, like hinge loss of SVM for classification or $\epsilon$-SVR for regression
- Regularization: RKHS norm, sort of ridge

## Neural networks and deep learning

**1-Hidden Layer Neural net:** A single layer network (from Wikipedia)
Each arc contains a parameter $w_{ij}$
Each node sums its inputs
Each of $K$ hidden nodes (here $K = 4$) applies a non-linear function $\sigma$ :

$$\hat{f}_l(x) = \sum_{k=1}^{K} w_{kl}^{(2)}\sigma(\sum_{j=1}^{p} w_{jk}^{(1)}x_j),$$

where $l$ is the index of the output nodes (in regression or two-class classification usually a single output node).

Designing a simple Neural net:

- How many hidden layers, how many nodes in each?
- Which non-linearity $\sigma$ to use?
- One or more output nodes? For regression or 2-class classification, usually only one
- Given training data $T$, how do we learn the weights/parameters $W$?

**Gradient descent for parameter learning:** As in other methods, we start from defining a loss function $L(y, \hat{y})$ for learning, for example

- Squared loss (RSS) for regression, like in OLS or our boosting example
- Bernoulli log-likelihood (AKA cross-entropy) for classification, like in logistic regression

Since $\hat{y}_i = \hat{f}(x_i)$ is a function of the parameters $W$, we can write the loss as function of $W$, in our simple example with one hidden layer, one output node, and squared loss:

$$\sum_{i=1}^{n} L(y_i, \hat{y}_i) = \sum_{i=1}^{n} \left( y_i - \sum_{k=1}^{K} w_k^{(2)} \sigma(\sum_{j=1}^{p} w_{jk}^{(1)} x_{ij}) \right)^2$$

Given $W^{(t-1)}$, we calculate the derivative (gradient) of the loss function relative to every parameter $\nabla L^{(t)}$. Dimension of $\nabla L$ is the number of parameters in our network — can be millions or billions. We then take a *downward* step to decrease the loss

$$W^{(t)} = W^{(t-1)} - \epsilon \nabla L^{(t)}$$

We repeat that for many iterations

## Two important notions: back-propagation and stochastic gradient descent (SGD)

A big part of the Neural Nets knowledge-base deals with ways of updating the weights based on gradient which are:

- Statistically efficient (including regularization, implicitly or explicitly)
- Computationally efficient (how do we get somewhere in such a huge parameter space, with non-convex functions of parameters?)

Back-prop is an efficient approach for taking advantage of the graphical model structure for calculating elements of $\nabla L^{(t)}$ of the form:

$$\frac{\partial \sum_{i=1}^{n} L(y_i, \hat{y}_i^{(t-1)})}{\partial w_{jk}^{(l)}}$$

In mathematical terms, back-prop is simply using the chain rule to differentiate for weights that are "deep" in the function, and uses the layer structure to calculate it efficiently. More generally, the important idea is that of a *computation graph* that allows to efficiently calculate the gradient relative to all parameters.

SGD uses the idea that:

$$\frac{\partial \sum_{i=1}^{n} L(y_i, \hat{y}_i)}{\partial w_{jk}^{(l)}} = \sum_{i=1}^{n} \frac{\partial L(y_i, \hat{y}_i)}{\partial w_{jk}^{(l)}}$$

SGD calculates the term inside the sum for one observation (or small group of observations) every time, and updates the weights. Because the number of weights can be very large, backprop + SGD allow updating all of them with relatively little computation. In practice, this converges much faster than calculating $\nabla L$ over all observations before updating.

## From Neural Nets to deep learning

Neural nets have been around for decades, initially proposed as "imitating the brain". They went in and out of fashion, not the leading choice for predictive modeling as of 2005 (Boosting was, including for image processing!) Starting around 2005: the deep learning (DL) revolution.

With the explosion of computing power, could fit *much* bigger neural nets, with millions or billions of parameters. In parallel, architectures got more imaginative, like:

- Convolutional NNets for images (we will discuss)
- Recurrent NNs
- GANS

These new architectures can address a wide range of problems beyond standard predictive modeling, a few examples:

- Reinforcement learning: learning to play games, the "reward" is only observed at the end
- Auto-encoding: find structure in data (non-linear PCA)
- Transfer learning: Learn networks on problems where you have lots of data, adjust to problems with a little data

They also involve more modern statistical elements, such as:

- Dropout: occasionally zero some of the weights in the network to *get out of local minimum*
- Momentum: Instead of just using the gradient (first derivative) remember previous gradients (sort-of second derivative)

Starting around 2005 we observe amazing accuracy gains in many domains from using DL models, fields dominated by DL today: image recognition, speech recognition, natural language processing,...

## Convolutional neural nets: the original DL revolution for images

Motivating example: CIFAR-10
An image is a $K \times K \times L$ object (for example $K = 100, L = 3$ layers of color). For simplicity assume here one layer $L = 1$. Denote an image by $A$, the $i, j$ pixel is $A(i, j)$ A *local area* in the image is a $q \times q$ region (say $q = 3$ or $q = 5$).

**Filters as detectors:** A *filter* $f$ is a $q \times q$ mask $f(k,l),\ k,l = 1,\dots q$. Now $f$ can be applied to every $q \times q$ area in $A$:

$$(f * A)(i,j) = \sum_{k=1}^{q} \sum_{l=1}^{q} f(k,l) \times A(i+k, j+l)$$

Gives a new "image" of size $(K-q+1) \times (K-q+1)$, where each pixel is a *convolution* of a neighborhood with the filter.

**Filter example: finding a corner or a line** Consider the following two $3 \times 3$ filters:

$$f_1 = \begin{pmatrix} 0 & 0 & 0 \\ 1 & 1 & 0 \\ 0 & 1 & 0 \end{pmatrix}, \quad f_2 = \begin{pmatrix} 0 & 1 & 0 \\ 0 & 1 & 0 \\ 0 & 1 & 0 \end{pmatrix}$$

$(f_1 * A)$ gives high values where $A$ has a corner, while $(f_2 * A)$ gives high values where $A$ has a vertical line Example: Sobel filter finds edges

Filters can find "features" in the image that can help identify it:

- A house has corners
- A snake has a repeating pattern on its back
- A cat has pointy ears

But we don't know what features will allow us to solve a specific problem:

- Corners, lines, edges, patterns?
- What scale? A small cat has small pointy ears, a big cat has LARGE pointy ears — they are both cats

So, we want the model to "learn" filters of different types and different scales that will help it solve the specific problem

**A CNN architecture:**

- We don't decide on filters, we "learn" them: a $q \times q$ filter has $q^2$ parameters.
- Each filter is applied to the entire image to create a new "image".
- Images are scaled ("pooling"), then more filters are applied, to find features at different scales
- The learned features are then used in a dense feed-forward network to classify images

Example

Some details:

- Number of parameters of convolutional layers is relatively small: assume input is $K \times K \times L$ and we have $C$ convolutions of size $q \times q \times L$, then the number of parameters is $C \times q^2 \times L$ regardless of image size $K$
- Pooling: decreasing image size from $K \times K$ usually to $K/2 \times K/2$ by aggregating every $2 \times 2$ region to a single pixel
- Most common pooling operator: max pooling (are there pointy ears anywhere in this region?)
- By alternating convolutions and pooling we learn features at different scales
- Important: all parameters (filters, dense layers weights,...) are learned *together* using SGD