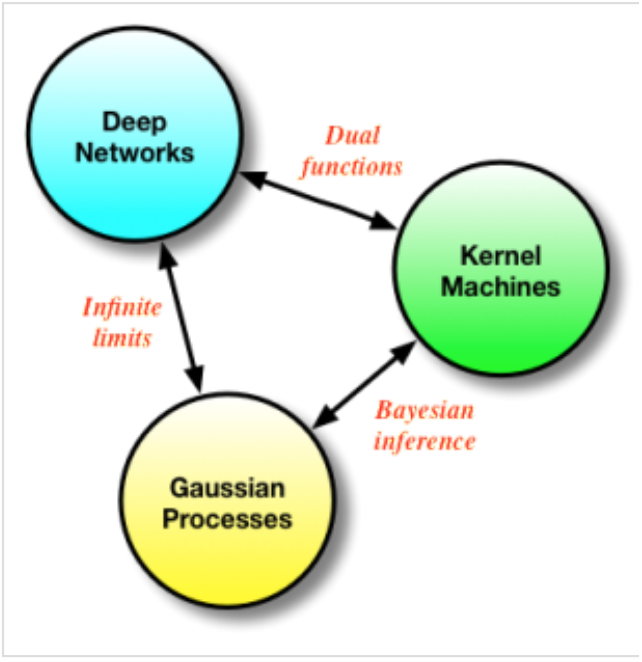
A Statistical View of Deep Learning (III): Memory and Kernels

9 Apr 2015 | Machine Learning and Statistics

Memory, *the ways in which we remember and recall past experiences and data to reason about future events*, is a term used frequently in current literature. All models in machine learning consist of a memory that is central to their usage. We have two principal types of memory mechanisms, most often addressed under the types of models they stem from: parametric and non-parametric (but also all the shades of grey in-between). Deep networks represent the archetypical parametric model, in which memory is implemented by distilling the statistical properties of observed data into a set of model parameters or weights. The poster-child for non-parametric models would be kernel machines (and nearest neighbours) that implement their memory mechanism by actually storing all the data explicitly. It is easy to think that these represent fundamentally different ways of reasoning about data, but the reality of how we derive these methods points to far deeper connections and a more fundamental similarity.

Deep networks, kernel methods and Gaussian processes form a continuum of approaches for solving the same problem - in their final form, these approaches might seem very different, but they are fundamentally related, and keeping this in mind can only be useful for future research. This connection is what I explore in this post.



Connecting machine learning methods for regression.

Basis Functions and Neural Networks

All the methods in this post look at regression: learning discriminative or input-output mappings. All such methods extend the humble linear model, where we assume that linear combinations of the input data ***x***, or transformations of it **φ(x),** explain the target values *y*. The **φ(x)** are basis functions that transform the data into a set of more interesting features. Features such as SIFT for images or MFCCs for audio have been popular in the past - in these cases, we still have a linear regression, since the basis functions are fixed. Neural networks give us the ability to use *adaptive basis functions,* allowing us to learn what the best features are from data instead of designing these by-hand, and allowing for a non-linear regression.

A useful probabilistic formulation separates the regression into systematic and random components: the systematic component is a function *f* we wish to learn, and the targets are noisy realisations of this function. To connect neural networks to the linear model, I'll explicitly separate the last linear layer of the neural network from the layers that appear before it. Thus for an *L*-layer deep neural network, I'll denote the first *L-1* layers by the mapping **φ(x; θ)** with parameters **θ,** and the final layer weights **w**; the set of all model parameters is **q** = **{θ, w}**.

$$\text{Sytematic:} \quad f = \mathbf{w}^\top \phi(\mathbf{x};\theta) \qquad \mathbf{q} \sim \mathcal{N}(0, \sigma_q^2 \mathbf{I}),$$

$$\text{Random:} \quad y = f(\mathbf{x}) + \epsilon \qquad \epsilon \sim \mathcal{N}(0, \sigma_y^2)$$

Once we have specified our probabilistic model, this implies an objective function for optimising the model parameters given by the negative log joint-probability. We can now apply back-propagation and learn all the parameters, performing MAP estimation in the neural network model. Memory in this model is maintained in the parametric modelling framework; we do not save the data but compactly represent it by the parameters of our model. This formulation has many nice properties: we can encode properties of the data into the function *f*, such as being a 2D image for which convolutions are sensible, and we can choose to do a stochastic approximation for scalability and perform gradient descent using mini-batches instead of the entire data set. The loss function for the output weights is of particular interest, since it will offers us a way to move from neural networks to other types of regression.

$$J(\mathbf{w}) = \frac{1}{2}\sum_{n=1}^{N}(y_n - \mathbf{w}^\top \phi(\mathbf{x}_n;\theta))^2 + \frac{\lambda}{2}\mathbf{w}^\top \mathbf{w}.$$

Kernel Methods

If you stare a bit longer at this last objective function, especially as formulated by explicitly representing the last linear layer, you'll very quickly be tempted to compute its dual function [1][pp. 293]. We'll do this by first setting the derivative w.r.t. **w** to zero and solving for it:

$$\nabla J(\mathbf{w}) = 0 \implies \mathbf{w} = \frac{1}{\lambda}\sum_n (y_n - \mathbf{w}^\top \phi(\mathbf{x}_n))\phi(\mathbf{x}_n)$$

$$\mathbf{w} = \sum_n \alpha_n \phi(\mathbf{x}_n) = \mathbf{\Phi}^\top \boldsymbol{\alpha} \qquad \alpha_n = -\frac{1}{\lambda}(\mathbf{w}^\top \phi(\mathbf{x}_n) - y_n)$$

We've combined all basis functions/features for the observations into the matrix **Φ**. By taking this optimal solution for the last layer weights and substituting it into the loss function, two things emerge: we obtain the dual loss function that is completely rewritten in terms of a new parameter **α,** and the computation involves the matrix product or Gram matrix **K=ΦΦ'**. We can repeat the process and solve the dual loss for the optimal parameter **α**, and obtain:

$$\nabla J(\boldsymbol{\alpha}) = 0 \implies \boldsymbol{\alpha} = (\mathbf{K} + \lambda \mathbf{I}_N)^{-1}\mathbf{y}$$

And this is where the kernel machines deviate from neural networks. Since we only need to consider inner products of the features **φ(x)** (implied by maintaining **K**), instead of parameterising them using a non-linear mapping given by a deep network, we can use *kernel substitution* (aka, the kernel trick) and get the same behaviour by choosing an appropriate and rich kernel function *k(**x, x'**). This highlights the deep relationship between deep networks and kernel machines: they are more than simply related, they are duals of each other*.

The memory mechanism has now been completely transformed into a non-parametric one - we explicitly represent all the data points (through the matrix **K**). The advantage of the kernel approach is that is often easier to encode properties of the functions we wish to represent e.g., functions that are up to *p-th* order differentiable or periodic functions, but stochastic approximation is now not possible. Predictions for a test point ***x**** can now be written in a few different ways:

$$f = \mathbf{w}_{MAP}^\top \phi(\mathbf{x}^*) = \boldsymbol{\alpha}^\top \boldsymbol{\Phi}(\mathbf{x})\phi(\mathbf{x}^*) = \sum_n \alpha_n k(\mathbf{x}^*, \mathbf{x}_n) = k(\mathbf{X}, \mathbf{x}^*)^\top (\mathbf{K} + \lambda \mathbf{I})^{-1}\mathbf{y}$$

The last equality is a form of solution implied by the Representer theorem and shows that we can instead think of a different formulation of our problem: one that *directly penalises the function* we are trying to estimate, subject to the constraint that the function lies within a Hilbert space (and providing a direct non-parametric view):

$$J(f) = \frac{1}{2}\sum_{n=1}^{N}(y_n - f(\mathbf{x}_n))^2 + \frac{\lambda}{2}\|f\|_{\mathcal{H}}^2.$$

Gaussian Processes

We can go even one step further and obtain not only a MAP estimate of the function *f*, but also its variance. We must now specify a probability model that yields the same loss function as this last objective function. This is possible since we now know what a suitable prior over functions is, and this probabilistic model corresponds to Gaussian process (GP) regression [2]:

$$p(f) = \mathcal{N}(\mathbf{0}, \mathbf{K}) \qquad p(y|f) = \mathcal{N}(y|f, \lambda)$$

We can now apply the standard rules for Gaussian conditioning to obtain a mean and variance for any predictions ***x****. What we obtain is:

$$p(f^*|\mathbf{X}, \mathbf{y}, \mathbf{x}^*) = \mathcal{N}(\mathbb{E}[f^*], \mathbb{V}[f^*])$$

$$\mathbb{E}[f^*] = k(\mathbf{X}, \mathbf{x}^*)^\top (\mathbf{K} + \lambda \mathbf{I})^{-1}\mathbf{y}$$

$$\mathbb{V}[f^*] = k(\mathbf{x}^*, \mathbf{x}^*) - k(\mathbf{X}, \mathbf{x}^*)^\top (\mathbf{K} + \lambda \mathbf{I})^{-1} k(\mathbf{X}, \mathbf{x}^*)$$

Conveniently, we obtain the same solution for the mean whether we use the kernel approach or the Gaussian conditioning approach. We now also have a way to compute the variance of the functions of interest, which is useful for many problems (such as active learning and optimistic exploration). Memory in the GP is also of the non-parametric flavour, since our problem is formulated in the same way as the kernel machines. GPs form another nice bridge between kernel methods and neural networks: we can see GPs as derived by Bayesian reasoning in kernel machines (which are themselves dual functions of neural nets), or we can obtain a GP by taking the number of hidden units in a one layer neural network to infinity [3].

Summary

Deep neural networks, kernel methods and Gaussian processes are all different ways of solving the same problem - how to learn the best regression functions possible. They are deeply connected: starting from one we can derive any of the other methods, and they expose the many interesting ways in which we can address and combine approaches that are ostensibly in competition. I think such connections are very interesting, and should prove important as we continue to build more powerful and faithful models for regression and classification.

Some References

[1] Christopher M Bishop, *Pattern recognition and machine learning*, , 2006

[2] Carl Edward Rasmussen, *Gaussian processes for machine learning*, , 2006

[3] Radford M Neal, *Bayesian Learning for Neural Networks*, , 1994