Algorithms-ML Reading Group

Convolutional Kernel Nets December 17, 2015

Lecturer: Zaid Harchaoui Scribe: Kiran Vodrahalli

Contents

1 Overview

The main point is Convolutional Kernel Nets are unsupervised versions of convolutional nets which outperform CNNs at some times on image retrieval tasks. The hope is you can characterize the RKHS norm. We can now hope to build a theory, which seems to be out of reach in the neural net literature. In fact you don't need to even do backprop (though you could do better that way) - we only report results of doing a feedforward computation! Of course, training one layer is NP-hard but you can approximate it well. All the code is online.

2 Introduction

Zaid Harchaoui is from Courant Institute at NYU. Convolutional Neural Nets give you features which look like filters; we can think of visualizing them as averages over all images; it's also kind of similar to nearest neighbor. The CNN-based features are in all computer vision pipelines, 1 billion images are input to CNNs per day. But compared to Facebook and Google, who are capable of optimizing over all hyperparameters (which includes the **architecture** - the ordering of the layers, number of neurons per layer, patch sizes etc). There is not a principled way of optimizing over these hyperparameters; and Google etc. do brute force. People have never proposed a scientific method for coming up with architecture. Similarly, they use a bunch of tricks to train neural networks. So training neural networks is much closer to exploring over all hyperparameters in the current day. This is a big concern! You need huge computational resources to explore over all of these networks. Well if you find it then life is nice. Even if you push the optimization really hard, you will still not be optimal! That very particular architecture made it; it is much closer to just brute force though (these things only started working after several years when we had computational power).

I will prove to you that this structure is actually incorrect.

Open Problems;

1. All apllications of CNNs are either pre-trained or re-optimized from previous ones.
2. Training CNNs is "black magic" and/or requires a bunch of computing resources. Beyond stochastic gradient descent; the way they do it. You start with step size magically, and change it magically.
3. Theoretical understanding of CNNs currently seems out of reach (are current architectures the right objects to study?)
4. Extentions of CNNs to structured data is unclear.
5. The deep-learning community very strongly believes that depth is important (circuit argument). I don't really agree with this. I can show that with shallow architectures and kernels, you can beat particular things. Do you really need deep networks?

I will give you code that you can download and be able to reproduce the results of something like AlexNet, without a ton of computer power.

3 Convolutional Kernel Nets

It's a successful blend of kernels with deep learning. This is just another way to define a function class on feature representations. What I really want to show is that kernels are more convenient. You can train these in a completely unsupervised manner, without using any labels. Here you have a very singular pipeline. Your image is a 2d lattice, you can take a patch, apply a convolution, and apply some kind of nonlinearity. Then you can perform pruning on the image, and redo this process again and again. We will show how to do that by solving ERM.

Definition 3.1. For any pixel location on the lowest layer of the net, I assume I can compute an image feature map. For instance, this can be the local gradients. The kernel is as follows: To compare two patches ϕ, ϕ' correspond to two subregions, you sum over all pixels in one image and over all in the other image, and of those low-level features you take the norm and multiply by a Gaussian kernel on pixel locations, and the other is a Gaussian kernel acting on those feature maps. If you start from the low-level feature maps, the norm is the l_2 norm. As you move to the next layer, you compute in an RKHS (potentially infinite dimensional). This allows you to enforce invariance. An image feature map is $\phi : \Omega \to \mathcal{H}$, where $\Omega \in [0,1]^2$ is a set of coordinates in the image and $\mathcal{H}$ is a Hilbert space. The kernel is defined as

$$K(\phi, \phi') = \sum_{x \in \Omega, x' \in \Omega} \|\phi(x)\|_{\mathcal{H}} \|\phi'(x)\|_{\mathcal{H}} e^{-(1/2\sigma^2)\|x-x'\|_2^2} e^{-(1/2\sigma^2)\|\phi(x)-\phi'(x)\|_{\mathcal{H}}^2}$$

You can generalize this to translation by re-writing $x = u + z, x' = u' + z'$. The low-level features we use are a gradient map (this is easy to compute), a patch map (extract a region from a pixel location).

You can define an RKHS at each layer which is different. You can consider coordinates Ω_{k-1} and Hilbert space $\mathcal{H}_{k-1}$. On the first patch in zeroth layer, compute a feature map to get to Ω_1 and $\mathcal{H}_1$. And then you can keep iterating through layers. We define a hierarchy of spatial maps, and we want to relate this to what the neural network is actually doing.

We motivate a cost function: Over all possible pairs of images, we want to take our convolutional kernel and approximate it by summing over a discrete subset of points (instead of using an integral), where patches are from the previous layer. If you plot the resulting nonlinearities, it looks like ReLu. We are taking the non-linear evaluations and replacing by linear combinations of the nonlinearities for a finite number of the values. We are effectively replacing with a low-dimensional dot product. So after learning, we have the mapping $x \to [\sqrt{\eta_1} e^{-1/\sigma^2 \|x-w\|_2^2}]_{i=1}^p \in \mathbb{R}^p$, where p is large. x is a patch and w is effectively a filter. We achieve this with cost function

$$\min_{\eta \in \mathbb{R}^p, W \in \mathbb{R}^{m \times p}} \frac{1}{n} \sum_{i=1}^n \left(e^{-1/2\sigma^2 \|x_i - x_i'\|_2^2} - \sum_{i=1}^p \eta_i e^{-1/\sigma^2 \|x_i - w\|_2^2} e^{-1/\sigma^2 \|x_i' - w_i\|_2^2} \right)^2$$

We are basically trying to find a sparse dictionary W that approximates the x well (summing over only a small portion). The Gaussian filter enforces the constrast (sharper edges, and so on). But you really have something else in mind. You took closeness to say p things (in the optimization), and now you're only doing the two largest ones - it's not exactly contrast. We don't solve even for optimally.

Now the reason you use this particular kernel - you're kind of emphasizing the peaks. You could theoretically do dictionary learning/sparse coding. I just didn't try it. There's just all these papers that say Gaussian kernel is universal, and so on. From the math point of view, you can take **any** divisible kernel (a kernel you can write as a convolution with itself with a finite bandwidth). So you could theoretically chose anything like that, it's just that Gaussian kernel is convenient. All the Gaussian kernel is doing is taking p sample patches and saying "who are you close to". This is a very nice interpretation. This point of view sheds more light than the neural network point of view. Kernels are just one way to view it. The vision people always use Gaussian kernels, but it's unclear why. Now it somewhat makes sense. It also turns out that this kernel gives loss plots (see the loss function above) that end up looking similar to ReLu.

Remark 3.2. Here's a question from Sanjeev: In the settings of real life, are the matrices we deal with actually low-rank? Everything seems to be less than rank 20 (not a scientific fact!) Why is any kernel low-rank? Empirically, the rank of Gram matrices for kernels tend to be ≤ 20 (clearly a casual statement). Related to why word embeddings exist. As an aside, in a bunch of Fortran code, they approximate the Hessian with a low-rank matrix, 20 is the largest size they can afford for λ^{20} (this is a bit different).

3.1 The Algorithm

Then, this unsupervised algorithm trains on the data, and actually works. How do you propagate feature maps from one map to the other.

First you take the input, do convolution and the nonlinearity, then space out, then do feature pooling (Gaussian filtering and subsampling). This is like wavelets, but in wavelets, there is theory which says you don't lose anything by subsampling in the correct hierarchical way. That is not the case here.

To calculate the kernel, you use a finite dimensional approximation of the Gaussian kernel. You learn a ζ_k representation for each layer k, and approximate the kernel. This trains relatively quickly, 1 hour on 10-CIFAR.

To do image retrieval, the pipeline is to do keypoint detection with a Hessian-affine detector, then encoded into descriptor space with the CKN, and then aggregate with VLAD. You take all points falling into a cluster, and then compute dominant direction in each cluster. Each blob is a patch descriptor. Fisher vectors were inspired by Fisher kernel, it corresponds to computing a Gaussian mixture modeling and then taking the opposite of the direction, this is called the Fisher vector. This is what is used in Google.

The test we do is retrieve the important patches of each image. You compare to ground truth how well it does; this task is hard. This is a pain for convolutional neural nets, since they don't know how to train without supervision. Something we might wonder is how come neural nets spend so much time generate images. Since they can't train in an unsupervised way, they just generate images to check consistency, which is rather unprincipled.

CKNs trained only on Holidays in an unsupervised fashion beats AlexNet on Holidays (dataset) which is finetuned on the images after being trained on the huge ImageNet dataset. Then, we do SVM to do classification after learning representations. To get our results, we don't use any of the tricks in the usual literature - no data pre-processing, and just full stochastic gradient descent. You don't even need to tune the kernel bandwidth. If you just set it to the $10^{th}\%$ of the statistics of the data, you get very close to the best you can get. We only use 2 GPUs. This whole project is "design neural nets for the poor". We are now working on audio (Sham Kakade) and video applications.