Make RBF Networks Fast Again- Exploiting Multi-Threaded Computing to Speed Up RBF Networks

Chris Dusold
cd@vt.edu

Arijit Ray
ray93@vt.edu

Abstract

When learning ideas and concepts, humans often cluster similar ideas together, breaking up concepts into sub-concepts after learning the concepts. A way to model this idea is to train neural networks using soft-assignment clustering algorithms as non-linear activation layers. Concepts can be broken up through adding centroids to the output vocabulary after initial training and using transfer learning to fine tune the new sub-concepts. This can be used to grow vocabularies in neural networks with minimal training. However, extensive training times of these layers still pose a hindrance for fast training of networks with huge amounts of high dimensional data. This paper addresses the above problem by exploiting the high degree of parallelism that exists in the computation of these layers. We outline the translation of a CPU version of Radial Basis Function Layers (RBF) into a General Purpose Graphical Processing Unit (GPGPU) version for speed and for practical use in deep learning. We compare the forward and backward inference times of a single-threaded CPU, multi-threaded CPU, and a GPGPU implementation of an RBF layer, and show that we obtain considerable speed-ups when multi-tasking. We also publicly release all back-end and bench-marking code for ready integration with TensorFlow for use in the scientific community.

1 Introduction

Deep Neural Networks have been used to create impressive strides in a wide variety of Artificial Intelligence (AI) tasks over the past decade. In computer vision, they have achieved state-of-the-art performances on image classification and recognition.

When used for classifying images, neural networks make decisions about the classification based on where the input data point lies in a projected space. Often this projected space captures some semantic meaning from the task and dataset, and is sometimes referred to a "semantic space." This projection into a virtual semantic space is done by multiplying the input with a set of learned weights at each layer. The last few layers are usually affine fully-connected layers that create virtual non-linear boundaries in order to classify the data point depending on which territory it falls into in that projected space. However, this means that an input, which just crosses a boundary into a "class's" territory gets classified as that "class" even if it is far away from the region where most data points generally lie in that territory. This makes the networks vulnerable to adversarial examples that fool the network into believing they are of a class when in fact they are not. Radial basis functions (RBFs), on the other hand, take a calculated exemplar cluster center for each of the classes, and calculates how far away an input is from each cluster center in order to classify it. With clusters, a data point has to to be close to the region where the most common data points of that class are in order to get classified as that "class". Hence, it will be somewhat robust to adversarial examples.

Multiprocessing Final Project Report. Fall 2016, Virginia Tech.

A part of the success of Deep Neural Networks in computer vision can be attributed to their ability to transfer learn new tasks using layers that were previously trained for a related task. Transfer learning is the method of retraining the weights of the last n (where n is generally 1 or 2) layers of a neural network for the new task at hand. For a fully-connected layer as the last layer, this involves retraining the entire weights of that layer for the new task. This method removes a large amount of the training with the already trained previous layers. However, with clusters, we can simply add a new centroid while keeping the old centroids and merely fine-tune them in order to adjust for the distinction of the new class added in. Since we do not re-train the entire layer, this should result in a considerable additional speed-up. This is somewhat similar to an adult human learning the distinction between Fiji apples and Gala apples. The adult human does not try to re-learn the distinctions between Fiji apples, Gala apples, dogs, cats, cars and every other class that it learned since childhood, but simply learns the distinction in the class of apples that makes it Fiji vs. Gala.

In spite of these advantages of RBF Layers, they are still often slow to train, despite their original praise of being quick to train (Tetteh et al., 1996). Training involves a forward pass which computes a penalty based on how well the network is doing at the task, and then a backward pass which updates the weights in the network in order to hopefully make it better. This involves computationally expensive matrix operations, especially when dealing with huge amounts of high dimensional data. Fortunately, these calculations involve a high degree of parallelism. In this paper, we focus on making the computation of RBF layers faster by exploiting the parallelism using multi-threaded computing. We implement and benchmark the forward and backward inference times of a network run using a single RBF layer on a single-threaded CPU, a multi-threaded CPU, and a GPU implementation.

The rest of the paper is organized as follows- we discuss some of the prior work done on RBF layers, then discuss the math behind the parallelism that we exploit, and then, we finally show the benchmarking results and illustrate that we obtain considerable speed-ups when using multi-processing.

2 Prior Work

What is commonly referred to as an RBF Network was designed by Broomhead and Lowe (1988) to be just a RBFs followed by an affine layer. (Derks et al., 1995; Tetteh et al., 1996) That is, the distances to each centroid calculated by the RBF is multiplied by a matrix, then a bias is added to the results. Other arrangements are rarely explored, as this version has been shown to be a universal approximator. (Park and Sandberg, 1991; Liao et al., 2003) Because the output of the RBF is simply a distance metric, a gradient is fairly easy to compute, and because backpropagation is very general due to the chain rule, the RBF can be used in many different ways.

Although the backpropagation algorithm, made famous by Rumelhart et al. (1986), has grown to be the most widely used training method for neural networks, and RBF Networks were thought to be more robust than multi-layer perceptrons (MLPs) (Derks et al., 1995; Tetteh et al., 1996; Goodfellow et al., 2014), the use of radial basis functions in backpropagation networks hasn't been common enough to even warrant implementation in popular libraries such as Torch or TensorFlow. An argument that could be made is that they take too long to train (Derks et al., 1995), but others claim they can be faster than MLPs (Tetteh et al., 1996). Another possible argument is that their excessive non-linearities make the objective space unfavorable for deep learning, but recent works in adversarial analysis may show that to be a favorable feature. (Goodfellow et al., 2014)

The addition of new clusters in semantic spaces requires a robust semantic space that can identify like objects for many variances within the object type, and have enough detail to tell differences too. The new clusters need some measure of differences possible to separate them from the old ideas. This gap in the semantic space shares features explored by papers looking at adversarial examples, as initially introduced by Szegedy et al. (2013) as robustness in the gap between the decision boundary can also thwart adversarial examples.

Adversarial examples come in a couple forms. One form takes versions of the images in the dataset with minimal perturbations in a calculated manner to force the model to misclassify the data with high confidence. Another version uses images nothing like anything the model has seen before which are classified with high confidence. In both cases, the high confidence is the biggest problem. Often, as they show, these changes do not need to even be human perceptable, and the neural network

will still be completely fooled and confident in its prediction. Szegedy et al. (2013) argue that this is because non-linearities in the neural networks cause cliffs in the reasoning which cause calculated little changes to cascade into large results. This however was disputed by Goodfellow et al. (2014), with their claim that the linear nature caused the direction of maximum change, which aligns with the gradient, to cause the maximum perturbation, which carries through the whole network unimpeded because of its linear nature. Goodfellow et al. (2014) claim that RBF network models can only learn the dataset with so much accuracy, but the amount it learns seems to be robust when faced with adversarial examples. This ignores the seminal paper of MNIST (LeCun et al., 1998) which used a multi-layer RBF to achieve 96.4% accuracy, which is already significant enough to test this effect.

3 Euclidean Distance RBF Layer

K-Means yields a cluster model where the sum of the distances in euclidean space for all datapoints to the centroid their cluster is assigned to is minimized. This is typically done through Lloyd's Algorithm, which is just a hard-assigned constrained expectation maximization. In fact, Lloyd's Algorithm is so popular as a solution to the K-Mean problem, that most people refer to Lloyd's Algorithm as "K-Means" or the 'K-Means Algorithm." (Liberty et al., 2014) The assignment generally hard assigns the cluster to the datapoint, which would be equivalent to a one-hot assignment. That assignment creates an output function which has a gradient of zero everywhere except at cluster transitions, where it is non-differentiable. In order to correct this, a value based upon the distance could be used, such as a Radial Basis Function. The assignment of such values would make more sense if the centroids closest to the data point is highest in value, indicating a higher likelihood of the data point residing in that cluster. The easiest way to do this metric would be by using the negative distance, which would rise to a maximum of zero, indicating the point overlaps the centroid exactly. Since the distance function is frequently differentiable everywhere (except at a distance of zero for some distance types), this provides a good starting point to develop a naive clustering algorithm to use.

The distance from any point, x_i, to any centroid, μ_k, is simply $||x_i - \mu_k||$ for some norm function. Distance gets larger the farther the point is away from the centroid, but it makes more sense for larger values to be closer to align with non-linear classifiers like Softmax, where the max is what the data is classified as. A simple way to do so would be to use the negative distance as the output from this layer. Thus, the output y_k for input x_i will be

$$y_{i,k} = -||x_i - \mu_k||_2 \tag{1}$$

To backpropagate through this layer, a derivative for the distance metric has to be calculated with respect to the input, x_i.

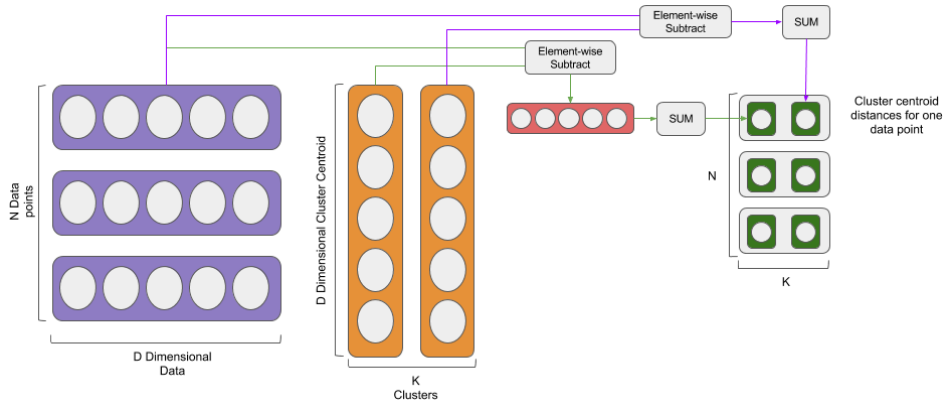
$$\begin{aligned}
y_k &= -||x_i - \mu_k||_2 \\
&= -\sqrt{(x_i - \mu_k)^T (x_i - \mu_k)} \\
u &= (x_i - \mu_k)^T (x_i - \mu_k) \\
\frac{du}{dx_i} &= 2(x_i - \mu_k) \\
y_k &= -\sqrt{u} \\
\frac{dy_k}{dx_i} &= -\frac{du}{2\sqrt{u}} \\
&= -\frac{2(x_i - \mu_k)}{2\sqrt{(x_i - \mu_k)^T (x_i - \mu_k)}} \\
&= -\frac{(x_i - \mu_k)}{||x_i - \mu_k||_2} \\
&= \frac{(x_i - \mu_k)}{y_k}
\end{aligned} \tag{2}$$

Somewhat intuitively, this shows the gradient, or the direction of steepest ascent, is in the direction towards the centroid, with a magnitude corresponding to one, as one unit step in that direction decreases the distance by one. Thus, using chain rule, the backpropagation merely is

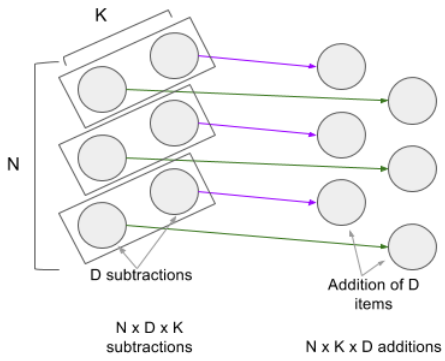
$$\frac{d\text{loss}}{dx_i} = \sum_{k=1}^{K} \frac{d\text{loss}}{dy_k} \frac{(x_i - \mu_k)}{y_k} \tag{3}$$

The gradient with respect to the centroids is equivalently calculated by substituting x_i with μ_k in the equation, and summing over all inputs ($\sum_{i=1}^{n}$) instead of all outputs.

3.1 Exploiting the Parallelism in this Computation



The above figure shows the computation for mapping an input batch of n data-points with d dimensions to k cluster centroids. Ideally, every computation can be done in parallel since the computations are almost independent of each other. The state diagram for the forward pass looks somewhat like the figure below.



Since it is customary to have a larger number of data points than the data dimension or number of cluster centroids, we split the computation among the threads available by the N dimension for the CPU implementation. Since we have a limited number of threads on the CPU to work reasonably with, we only split computation across the number of datapoints.

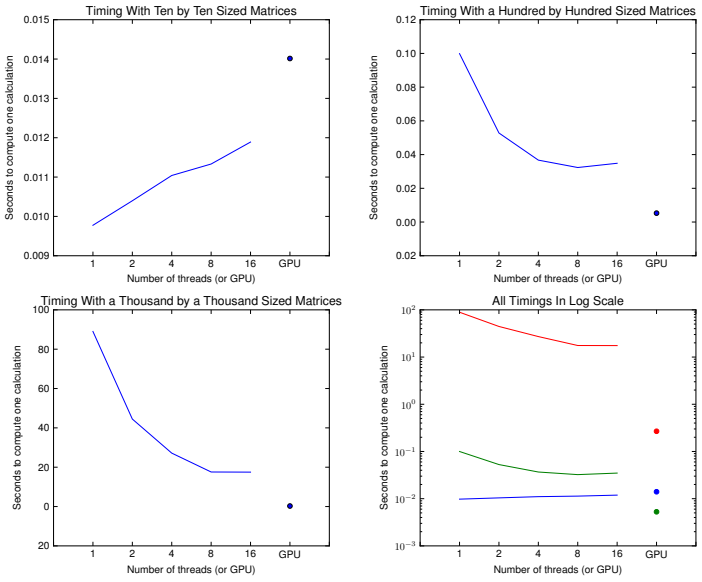
On the GPU, there are many thousands more threads to work with, so we may often be able to uniquely assign each datapoint and cluster center pairing its own thread on the GPU depending on the batch size used in training. Every single computation in feedforward has no shared computation, but it does have shared memory access. The bus in GPU tends to speed up accesses to memory that are adjacent for adjacent threads. The main optimization from this project which was performed for the GPGPU section was making the memory accesses coalesce such as mentioned above. This forced us to duplicate some efforts as we could no longer align the gradient for the clusters coming

from the same output with the gradient for the input coming from the same output. Also, the choice of data shape to directly align with the traditional affine layers in MLPs forced the memory accesses of the datapoints to be transposed from those of the cluster centers. This was primarily because of future work planned outside of this class. Since the summation across the data dimension in feed forward all accessed the same output position, direct multiprocessing for the element-wise subtraction was not possible without a sizable intermediate, but utilizing reduction would be a way to solve this in the future. The intermediate for the test examples below, for instance, would get up to one thousand cubed times the size of the data, or 4 Gigabytes just for one layer, since we're using 32 bit float types.

4 Benchmarking RBFs

We implemented the forward pass and the backward pass for our custom RBF Layer in C++11 for integration with TensorFlow. We use the C++11 Thread Package for implementing the CPU multi-threaded versions. The code we compiled was tested on a computer with an i7-4790k with four physical cores and eight virtual threads in total[1] and a GTX 980 Ti[2] GPU. We tested this with one, two, four, eight, and sixteen threads on the CPU (even though we only have eight virtual cores) and thirty two by two hundred fifty six (32 x 256) threads on the GPU. To test against different loads, we ran the code with k data samples of k dimension each versus k cluster centers, with k values of ten, one hundred, and one thousand. Each of these computations had their timings calculated from averaging over ten runs. We release all back-end code and bench-marking code publicly under an MIT Creative Common License.[3]

4.1 Feedforward

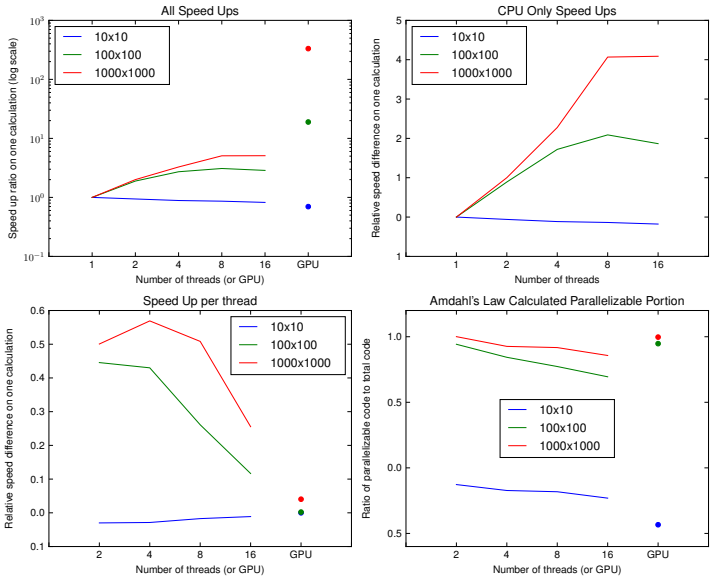


Unsurprisingly, ten by ten matrices is too little work to actually have demonstrable parallelism, so the overhead caused by thread launching causes the timings to increase. That is fine, as this demonstrates that there's a lower bound somewhere in the order of magnitude between ten and a hundred. The results show a good (almost ideal) speed up for increasing the number of threads for larger and more realistic data sizes and dimensions. To compare more accurately, we utilized the single-threaded implementation timing to compute speed up ratios and relative speed differences below, as well as what the amount of parallelism theoretically with Amdahl's Law.

[1] 4.0 GHz, 4.4 GHz single core with Turbo Boost. Details: `http://ark.intel.com/products/80807/Intel-Core-i7-4790K-Processor-8M-Cache-up-to-4_40-GHz`

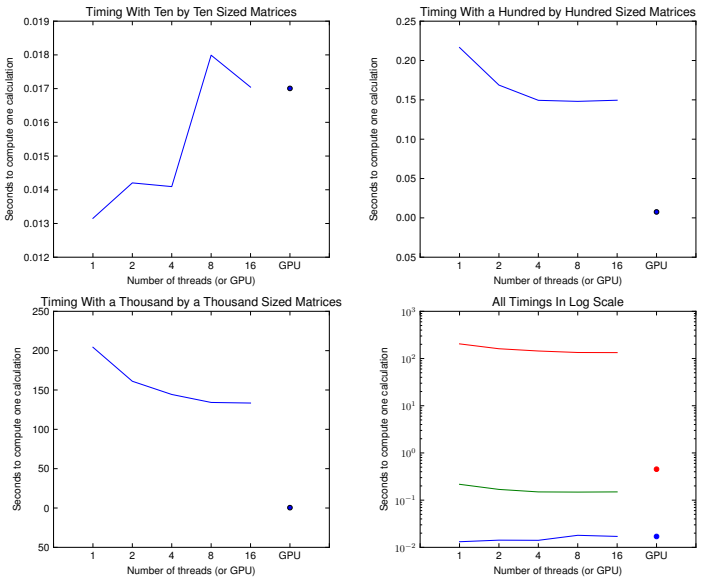
[2] Details: `http://www.evga.com/products/product.aspx?pn=06G-P4-4998-RX`

[3] Available at: `https://github.com/cdusold/TensorFlowRBF`

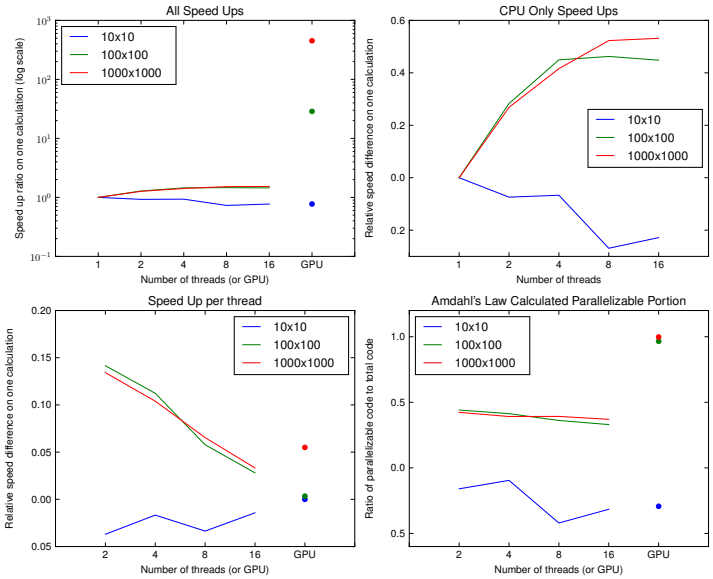


The results for ten by ten show that parallelizing the code resulted in more sequential code and thus a negative parallelizable portion (due to thread launching). As expected, the others were much more applicable, and showed very high parallel sections, with increasing parallelization with higher dimensional computations. Ignoring the sixteen thread case, as it wasn't fully supported by our hardware, the amount of theoretical parallelism on average for one hundred by one hundred sized data was 85.3% for the CPU implementation and 94.7% on GPUs. The average for one thousand by one thousand sized data was 94.8% on CPU and 99.7% on GPU.

4.2 Backpropagation



Again, ten by ten was too small for good parallelism. The speed ups were a lot less applicable for this computation on CPU, but still performed rather well on GPU



There was very little difference this time between the speed up for one hundred by one hundred sized data and one thousand by one thousand sized data. Continuing to ignore the sixteen thread case, the amount of theoretical parallelism on average for one hundred by one hundred sized data was 40.5% for the CPU implementation and 96.5% on GPUs. The average for one thousand by one thousand sized data was 40.2% on CPU and 99.8% on GPU.

5 Conclusions and Future Work

Overall, we obtained almost ideal speed-ups while using multi-thread processes on both CPU and GPU for feedforward, and in backprop on GPU. Interestingly, whereas the CPU version did much worse in backpropagation, the GPU implementation actually did better. In order to discover the cause in this disparity, more research will have to be done. Still, the speed ups are significant and show extreme, almost embarrassing, parallelism in the RBF layer computation.

In the future, the GPU implementation should be made even more optimized. The code we developed only parallelized along the number of samples and the number of clusters, but a parallel reduction can be made over the number of dimensions for a theoretical gain of $O(log(n)/n)$. Additionally, using templating to break out computation into kernels of appropriate block and grid sizes for computation is a common practice for optimization on GPUs, and we just use the same 32 blocks by 256 threads for every computation, without having profiled it outside of speed up.

Additionally, the value

$$\frac{d\text{loss}}{dx_i} = \sum_{k=1}^{K} \frac{d\text{loss}}{dy_k} \frac{(x_i - \mu_k)}{y_k}$$

has a shared computation intermediate value of $\frac{d\text{loss}}{dy_k} \frac{1}{y_k}$ for each of the x_i and that part should be precomputed by Tensorflow's already very optimized element-wise division before being passed into our code. We recognized this and excluded it from our paper as it seemed to be out of the intention of this class project, and the parallelization speed up will be skewed by the better implementation provided by Google, but when it gets finalized for TensorFlow over the winter, that optimization will be included.

The CPU version can also get some improvements, not just from the above precomputation in TensorFlow, but from changing the number of threads from specified to controlled by a threadpool implementation in either Boost or TensorFlow (as I believe they offer one as well). Again, this was outside the scope of testing thread count based speed up of our computation so it was left out.

References

D. S. Broomhead and D. Lowe. Radial basis functions, multi-variable functional interpolation and adaptive networks. Technical report, DTIC Document, 1988.

E. Derks, M. Pastor, and L. Buydens. Robustness analysis of radial base function and multi-layered feed-forward neural network models. *Chemometrics and Intelligent Laboratory Systems*, 28(1): 49 – 60, 1995. ISSN 0169-7439. doi: http://dx.doi.org/10.1016/0169-7439(95)80039-C. URL `http://www.sciencedirect.com/science/article/pii/016974399580039C`.

I. J. Goodfellow, J. Shlens, and C. Szegedy. Explaining and harnessing adversarial examples. *arXiv preprint arXiv:1412.6572*, 2014.

Y. LeCun, L. Bottou, Y. Bengio, and P. Haffner. Gradient-based learning applied to document recognition. *Proceedings of the IEEE*, 86(11):2278–2324, 1998.

Y. Liao, S.-C. Fang, and H. L. Nuttle. Relaxed conditions for radial-basis function networks to be universal approximators. *Neural Networks*, 16(7):1019–1028, 2003.

E. Liberty, R. Sriharsha, and M. Sviridenko. An algorithm for online k-means clustering. *arXiv preprint arXiv:1412.5721*, 2014.

J. Park and I. W. Sandberg. Universal approximation using radial-basis-function networks. *Neural computation*, 3(2):246–257, 1991.

D. E. Rumelhart, G. E. Hinton, and R. J. Williams. Learning representations by back-propagating errors. *Nature*, 323(Oct):533–536+, 1986.

C. Szegedy, W. Zaremba, I. Sutskever, J. Bruna, D. Erhan, I. Goodfellow, and R. Fergus. Intriguing properties of neural networks. *arXiv preprint arXiv:1312.6199*, 2013.

J. Tetteh, E. Metcalfe, and S. L. Howells. Optimisation of radial basis and backpropagation neural networks for modelling auto-ignition temperature by quantitative-structure property relationships. *Chemometrics and Intelligent Laboratory Systems*, 32(2):177 – 191, 1996. ISSN 0169-7439. doi: http://dx.doi.org/10.1016/0169-7439(95)00088-7. URL `http://www.sciencedirect.com/science/article/pii/0169743995000887`.