Notes on Convolutional Neural Networks

Jake Bouvrie
Center for Biological and Computational Learning
Department of Brain and Cognitive Sciences
Massachusetts Institute of Technology
Cambridge, MA 02139
jvb@csail.mit.edu

Initial release date: November 22, 2006
Revision date: May 27, 2014

Abstract

We discuss the derivation and implementation of convolutional neural networks, followed by an extension which allows one to learn sparse combinations of feature maps. The derivation we present is specific to two-dimensional data and convolutions, but can be extended without much additional effort to an arbitrary number of dimensions. Throughout the discussion, we emphasize efficiency of the implementation, and give small snippets of MATLAB code to accompany the equations.

1 Introduction

This document discusses the derivation and implementation of convolutional neural networks (CNNs) [3, 4], followed by a few straightforward extensions. Convolutional neural networks involve many more connections than weights; the architecture itself realizes a form of regularization. In addition, a convolutional network automatically provides some degree of translation invariance. This particular kind of neural network assumes that we wish to learn *filters*, in a data-driven fashion, as a means to extract features describing the inputs. The derivation we present is specific to two-dimensional data and convolutions, but can be extended without much additional effort to an arbitrary number of dimensions.

We begin with a description of classical backpropagation in fully connected networks, followed by a derivation of the backpropagation updates for the filtering and subsampling layers in a 2D convolutional neural network. Throughout the discussion, we emphasize efficiency of the implementation, and give small snippets of MATLAB code to accompany the equations. The importance of writing efficient code when it comes to CNNs cannot be overstated. We then turn to the topic of learning how to combine feature maps from previous layers automatically, and consider in particular, learning sparse combinations of feature maps.

Disclaimer: This rough note could contain errors, exaggerations, and false claims.

2 Vanilla Back-propagation Through Fully Connected Networks

In typical convolutional neural networks you might find in the literature, the early analysis consists of alternating convolution and sub-sampling operations, while the last stage of the architecture consists of a generic multi-layer network: the last few layers (closest to the outputs) will be fully connected 1-dimensional layers. When you're ready to pass the final 2D feature maps as inputs to the fully connected 1-D network, it is often convenient to just concatenate all the features present in all the output maps into one long input vector, and we're back to vanilla backpropagation. The standard backprop algorithm will be described before going onto specializing the algorithm to the case of convolutional networks (see e.g. [1] for more details).

2.1 Feedforward Pass

In the derivation that follows, we will consider the squared-error loss function. For a multiclass problem with c classes and N training examples, this error is given by

$$E^N = \frac{1}{2}\sum_{n=1}^{N}\sum_{k=1}^{c}(t_k^n - y_k^n)^2.$$

Here t_k^n is the k-th dimension of the n-th pattern's corresponding target (label), and y_k^n is similarly the value of the k-th output layer unit in response to the n-th input pattern. For multiclass classification problems, the targets will typically be organized as a "one-of-c" code where the k-th element of $\mathbf{t}^n$ is positive if the pattern $\mathbf{x}^n$ belongs to class k. The rest of the entries of $\mathbf{t}^n$ will be either zero or negative depending on the choice of your output activation function (to be discussed below).

Because the error over the whole dataset is just a sum over the individual errors on each pattern, we will consider backpropagation with respect to a single pattern, say the n-th one:

$$E^n = \frac{1}{2}\sum_{k=1}^{c}(t_k^n - y_k^n)^2 = \frac{1}{2}\|\mathbf{t}^n - \mathbf{y}^n\|_2^2. \tag{1}$$

With ordinary fully connected layers, we can compute the derivatives of E with respect to the network weights using backpropagation rules of the following form. Let ℓ denote the current layer, with the output layer designated to be layer L and the input "layer" designated to be layer 1. Define the output of this layer to be

$$\mathbf{x}^\ell = f(\mathbf{u}^\ell), \quad \text{with} \quad \mathbf{u}^\ell = \mathbf{W}^\ell\mathbf{x}^{\ell-1} + \mathbf{b}^\ell \tag{2}$$

where the output activation function $f(\cdot)$ is commonly chosen to be the logistic (sigmoid) function $f(x) = (1 + e^{-\beta x})^{-1}$ or the hyperbolic tangent function $f(x) = a\tanh(bx)$. The logistic function maps $[-\infty, +\infty] \to [0, 1]$, while the hyperbolic tangent maps $[-\infty, +\infty] \to [-a, +a]$. Therefore while the outputs of the hyperbolic tangent function will typically be near zero, the outputs of a sigmoid will be non-zero on average. However, normalizing your training data to have mean 0 and variance 1 along the features can often improve convergence during gradient descent [5]. With a normalized dataset, the hyperbolic tangent function is thus preferrable. LeCun recommends $a = 1.7159$ and $b = 2/3$, so that the point of maximum nonlinearity occurs at $f(\pm 1) = \pm 1$ and will thus avoid saturation during training if the desired training targets are normalized to take on the values ± 1 [5].

2.2 Backpropagation Pass

The "errors" which we propagate backwards through the network can be thought of as "sensitivities" of each unit with respect to perturbations of the bias[1]. That is to say,

$$\frac{\partial E}{\partial b} = \frac{\partial E}{\partial u}\frac{\partial u}{\partial b} = \delta \tag{3}$$

since in this case $\frac{\partial u}{\partial b} = 1$. So the bias sensitivity and the derivative of the error with respect to a unit's total input is equivalent. It is this derivative that is backpropagated from higher layers to lower layers, using the following recurrence relation:

$$\boldsymbol{\delta}^\ell = (W^{\ell+1})^T\boldsymbol{\delta}^{\ell+1} \circ f'(\mathbf{u}^\ell) \tag{4}$$

where "$\circ$" denotes element-wise multiplication. For the error function (1), the sensitivities for the output layer neurons will take a slightly different form:

$$\boldsymbol{\delta}^L = f'(\mathbf{u}^L) \circ (\mathbf{y}^n - \mathbf{t}^n).$$

Finally, the delta rule for updating a weight assigned to a given neuron is just a copy of the inputs to that neuron, scaled by the neuron's delta. In vector form, this is computed as an outer product

[1] This nifty interpretation is due to Sebastian Seung

between the vector of inputs (which are the outputs from the previous layer) and the vector of sensitivities:

$$\frac{\partial E}{\partial \mathbf{W}^\ell} = \boldsymbol{\delta}^\ell (\mathbf{x}^{\ell-1})^T \tag{5}$$

$$\Delta \mathbf{W}^\ell = -\eta \frac{\partial E}{\partial \mathbf{W}^\ell} \tag{6}$$

with analogous expressions for the bias update given by (3). In practice there is often a learning rate parameter η_{ij} specific to each weight $(\mathbf{W})_{ij}$.

3 Convolutional Neural Networks

Typically convolutional layers are interspersed with sub-sampling layers to reduce computation time and to gradually build up further *spatial* and *configural* invariance. A small sub-sampling factor is desirable however in order to maintain *specificity* at the same time. Of course, this idea is not new, but the concept is both simple and powerful. The mammalian visual cortex and models thereof [12, 8, 7] draw heavily on these themes, and auditory neuroscience has revealed in the past ten years or so that these same design paradigms can be found in the primary and belt auditory areas of the cortex in a number of different animals [6, 11, 9]. Hierarchical analysis and learning architectures may yet be the key to success in the auditory domain.

3.1 Convolution Layers

Let's move forward with deriving the backpropagation updates for convolutional layers in a network. At a convolution layer, the previous layer's feature maps are convolved with learnable kernels and put through the activation function to form the output feature map. Each output map may combine convolutions with multiple input maps. In general, we have that

$$\mathbf{x}_j^\ell = f\left(\sum_{i \in M_j} \mathbf{x}_i^{\ell-1} * \mathbf{k}_{ij}^\ell + b_j^\ell\right),$$

where M_j represents a selection of input maps, and the convolution is of the "valid" border handling type when implemented in MATLAB. Some common choices of input maps include all-pairs or all-triplets, but we will discuss how one might learn combinations below. Each output map is given an additive bias b, however for a particular output map, the input maps will be convolved with distinct kernels. That is to say, if output map j and map k both sum over input map i, then the kernels applied to map i are different for output maps j and k.

3.1.1 Computing the Gradients

We assume that each convolution layer ℓ is followed by a downsampling layer $\ell+1$. The backpropagation algorithm says that in order to compute the sensitivity for a unit at layer ℓ, we should first sum over the next layer's sensitivies corresponding to units that are connected to the node of interest in the current layer ℓ, and multiply each of those connections by the associated weights defined at layer $\ell + 1$. We then multiply this quantity by the derivative of the activation function evaluated at the current layer's pre-activation inputs, u. In the case of a convolutional layer followed by a downsampling layer, one pixel in the next layer's associated sensitivity map $\boldsymbol{\delta}$ corresponds to a block of pixels in the convolutional layer's output map. Thus each unit in a map at layer ℓ connects to only one unit in the corresponding map at layer $\ell + 1$. To compute the sensitivities at layer ℓ efficiently, we can *upsample* the downsampling layer's sensitivity map to make it the same size as the convolutional layer's map and then just multiply the upsampled sensitivity map from layer $\ell+1$ with the activation derivative map at layer ℓ element-wise. The "weights" defined at a downsampling layer map are all equal to β (a constant, see section 3.2), so we just scale the previous step's result by β to finish the computation of $\boldsymbol{\delta}^\ell$. We can repeat the same computation for each map j in the convolutional layer, pairing it with the corresponding map in the subsampling layer:

$$\boldsymbol{\delta}_j^\ell = \beta_j^{\ell+1}\Big(f'(\mathbf{u}_j^\ell) \circ \mathrm{up}(\boldsymbol{\delta}_j^{\ell+1})\Big)$$

where $\mathrm{up}(\cdot)$ denotes an upsampling operation that simply tiles each pixel in the input horizontally and vertically n times in the output if the subsampling layer subsamples by a factor of n. As we will discuss below, one possible way to implement this function efficiently is to use the Kronecker product:

$$\mathrm{up}(\mathbf{x}) \equiv \mathbf{x} \otimes \mathbf{1}_{n \times n}.$$

Now that we have the sensitivities for a given map, we can immediately compute the bias gradient by simply summing over all the entries in $\boldsymbol{\delta}_j^\ell$:

$$\frac{\partial E}{\partial b_j} = \sum_{u,v} (\boldsymbol{\delta}_j^\ell)_{uv}.$$

Finally, the gradients for the kernel weights are computed using backpropagation, except in this case the same weights are shared across many connections. We'll therefore sum the gradients for a given weight over all the connections that mention this weight, just as we did for the bias term:

$$\frac{\partial E}{\partial \mathbf{k}_{ij}^\ell} = \sum_{u,v} (\boldsymbol{\delta}_j^\ell)_{uv} (\mathbf{p}_i^{\ell-1})_{uv} \tag{7}$$

where $(\mathbf{p}_i^{\ell-1})_{uv}$ is the *patch* in $\mathbf{x}_i^{\ell-1}$ that was multiplied elementwise by $\mathbf{k}_{ij}^\ell$ during convolution in order to compute the element at (u, v) in the output convolution map $\mathbf{x}_j^\ell$. At first glance it may appear that we need to painstakingly keep track of which patches in the input map correspond to which pixels in the output map (and its corresponding map of sensitivities), but equation (7) can be implemented in a single line of MATLAB using convolution over the valid region of overlap:

$$\frac{\partial E}{\partial \mathbf{k}_{ij}^\ell} = \mathrm{rot180}(\mathrm{conv2}(\mathbf{x}_i^{\ell-1},\ \mathrm{rot180}(\boldsymbol{\delta}_j^\ell),\ 'valid')).$$

Here we rotate the $\boldsymbol{\delta}$ image in order to perform cross-correlation rather than convolution, and rotate the output back so that when we perform convolution in the feed-forward pass, the kernel will have the expected orientation.

3.2 Sub-sampling Layers

A subsampling layer produces downsampled versions of the input maps. If there are N input maps, then there will be exactly N output maps, although the output maps will be smaller. More formally,

$$\mathbf{x}_j^\ell = f\Big(\beta_j^\ell \,\mathrm{down}(\mathbf{x}_j^{\ell-1}) + b_j^\ell\Big),$$

where $\mathrm{down}(\cdot)$ represents a sub-sampling function. Typically this function will sum over each distinct n-by-n block in the input image so that the output image is n-times smaller along both spatial dimensions. Each output map is given its own multiplicative bias β and an additive bias b. We can also simply throw away every other sample in the image [10].[2]

3.2.1 Computing the Gradients

The difficulty here lies in computing the sensitivity maps. One we've got them, the only learnable parameters we need to update are the bias parameters β and b. We will assume that the subsampling layers, are surrounded above and below by convolution layers. If the layer following the subsampling layer is a fully connected layer, then the sensitivity maps for the subsampling layer can be computed with the vanilla backpropagation equations introduced in section 2.

When we tried to compute the gradient of a kernel in section 3.1.1, we had to figure out which patch in the input corresponded to a given pixel in the output map. Here, we must figure out which patch in the current layer's sensitivity map corresponds to a given pixel in the next layer's sensitivity map in order to apply a delta recursion that looks something like equation (4). Of course, the weights

[2] Patrice Simard's "pulling" vs "pushing" appears to be unnecessary if you use `conv` with zero padding to compute the sensitivities and gradients.

multiplying the connections between the input patch and the output pixel are exactly the weights of the (rotated) convolution kernel. This is again efficiently implemented using convolution:

$$\begin{aligned}\boldsymbol{\delta}_j^\ell &= f'(\mathbf{u}_j^\ell) \circ \sum_{i \in M_j^{\ell+1}} \text{conv2}\big(\boldsymbol{\delta}_j^{\ell+1},\ \text{rot180}(\mathbf{k}_{ij}^{\ell+1}),\ \text{'full'}\big) \\ &= f'(\mathbf{u}_j^\ell) \circ \text{conv2}\Big(\boldsymbol{\delta}_j^{\ell+1},\ \text{rot180}\big(\sum_{i \in M_j^{\ell+1}} \mathbf{k}_{ij}^{\ell+1}\big),\ \text{'full'}\Big)\end{aligned} \tag{8}$$

where $M_j^{\ell+1}$ denotes the collection of maps at the next (higher) layer $(\ell+1)$ which receive input from subsampling layer map j at layer ℓ. In general, if a subsampling layer ouput map projects to multiple maps at the next layer, you'll need to sum over the convolutions with each kernel that filters the given subsampling map to compute its sensitivity. As before, we rotate the kernel to make the convolution function perform cross-correlation. Notice that in this case, however, we require the "full" convolution border handling, to borrow again from MATLAB's nomenclature. This small difference lets us deal with the border cases easily and efficiently, where the number of inputs to a unit at layer $\ell+1$ is not the full size of the $n \times n$ convolution kernel. In those cases, the "full" convolution will automatically pad the missing inputs with zeros.

At this point we're ready to compute the gradients for b and β. The additive bias is again just the sum over the elements of the sensitivity map:

$$\frac{\partial E}{\partial b_j} = \sum_{u,v} (\boldsymbol{\delta}_j^\ell)_{uv}.$$

The multiplicative bias β will of course involve the original down-sampled map computed at the current layer during the feedforward pass. For this reason, it is advantageous to save these maps during the feedforward computation, so we don't have to recompute them during backpropagation. Let's define

$$\mathbf{d}_j^\ell = \text{down}(\mathbf{x}_j^{\ell-1}).$$

Then the gradient for β is given by

$$\frac{\partial E}{\partial \beta_j} = \sum_{u,v} (\boldsymbol{\delta}_j^\ell \circ \mathbf{d}_j^\ell)_{uv}.$$

3.3 Learning Combinations of Feature Maps

Often times, it is advantageous to provide an output map that involves a sum over several convolutions of different input maps. In the literature, the input maps that are combined to form a given output map are typically chosen by hand. We can, however, attempt to learn such combinations during training. Let α_{ij} denote the weight given to input map i when forming output map j. Then output map j is given by

$$\mathbf{x}_j^\ell = f\left(\sum_{i=1}^{N_{in}} \alpha_{ij}(\mathbf{x}_i^{\ell-1} * \mathbf{k}_i^\ell) + b_j^\ell\right),$$

subject to the constraints

$$\sum_i \alpha_{ij} = 1, \quad \text{and} \quad 0 \le \alpha_{ij} \le 1.$$

These constraints can be enforced by setting the α_{ij} variables equal to the softmax over a set of unconstrained, underlying weights c_{ij}:

$$\alpha_{ij} = \frac{\exp(c_{ij})}{\sum_k \exp(c_{kj})}.$$

Because each set of weights c_{ij} for fixed j are independent of all other such sets for any other j, we can consider the updates for a single map and drop the subscript j. Each map is updated in the same way, except with different j indices.

The derivative of the softmax function is given by

$$\frac{\partial \alpha_k}{\partial c_i} = \delta_{ki}\alpha_i - \alpha_i\alpha_k \tag{9}$$

(where here δ is used as the Kronecker delta), while the derivative of (1) with respect to the α_i variables at layer ℓ is

$$\frac{\partial E}{\partial \alpha_i} = \frac{\partial E}{\partial u^\ell}\frac{\partial u^\ell}{\partial \alpha_i} = \sum_{u,v}\left(\boldsymbol{\delta}^\ell \circ (\mathbf{x}_i^{\ell-1} * \mathbf{k}_i^\ell)\right)_{uv}.$$

Here, $\boldsymbol{\delta}^\ell$ is the sensitivity map corresponding to an output map with inputs $\mathbf{u}$. Again, the convolution is the "valid" type so that the result will match the size of the sensitivity map. We can now use the chain rule to compute the gradients of the error function (1) with respect to the underlying weights c_i:

$$\frac{\partial E}{\partial c_i} = \sum_k \frac{\partial E}{\partial \alpha_k}\frac{\partial \alpha_k}{\partial c_i} \tag{10}$$

$$= \alpha_i\left(\frac{\partial E}{\partial \alpha_i} - \sum_k \frac{\partial E}{\partial \alpha_k}\alpha_k\right). \tag{11}$$

3.3.1 Enforcing Sparse Combinations

We can also try to impose sparseness constraints on the distribution of weights α_i for a given map by adding a regularization penalty $\Omega(\boldsymbol{\alpha})$ to the final error function. In doing so, we'll encourage some of the weights to go to zero. In that case, only a few input maps would contribute significantly to a given output map, as opposed to all of them. Let's write the error for a single pattern as

$$\tilde{E}^n = E^n + \lambda \sum_{i,j} |(\boldsymbol{\alpha})_{ij}| \tag{12}$$

and find the contribution of the regularization term to the gradient for the weights c_i. The user-defined parameter λ controls the tradeoff between minimizing the fit of the network to the training data, and ensuring that the weights mentioned in the regularization term are small according to the 1-norm. We will again consider only the weights α_i for a given output map and drop the subscript j. First, we need that

$$\frac{\partial \Omega}{\partial \alpha_i} = \lambda \operatorname{sign}(\alpha_i) \tag{13}$$

everywhere except at the origin. Combining this result with (9) will allow us to derive the contribution:

$$\frac{\partial \Omega}{\partial c_i} = \sum_k \frac{\partial \Omega}{\partial \alpha_k}\frac{\partial \alpha_k}{\partial c_i} \tag{14}$$

$$= \lambda\left(|\alpha_i| - \alpha_i \sum_k |\alpha_k|\right). \tag{15}$$

The final gradients for the weights c_i when using the penalized error function (12) can be computed using (14) and (10):

$$\frac{\partial \tilde{E}^n}{\partial c_i} = \frac{\partial E^n}{\partial c_i} + \frac{\partial \Omega}{\partial c_i}.$$

3.4 Making it Fast with MATLAB

In a network with alternating sub-sampling and convolution layers the main computational bottlenecks are:

1. During the feedforward pass: downsampling the convolutional layer's output maps
2. During backpropagation: upsampling of a higher sub-sampling layer's delta's to match the size of the lower convolutional layer's output maps.
3. Application of the sigmoid and it's derivative.

Performing the convolutions during both the feedforward and backproagation stages are also computational bottlenecks of course, but assuming the 2D convolution routine is efficiently implemented, there isn't much we can do about it.

One might be tempted however to use MATLAB's built-in image processing routines to handle the up- and down-sampling operations. For up-sampling, `imresize` will do the job, but with significant overhead. A faster alternative is to use the Kronecker product function `kron`, with the matrix to be upsampled, and a matrix of ones. This can be an order of magnitude faster. When it comes to the down-sampling step during the feedforward pass, `imresize` does not provide the option to downsample by summing over distinct n-by-n blocks. The "nearest-neighbor" method will replace a block of pixels by only one of the original pixels in the block. An alternative is to apply `blkproc` to each distinct block, or some combination of `im2col` and `colfilt`. While both of these options only computes what's necessary and nothing more, repeated calls to the user-defined block-processing function imposes significant overhead. A much faster alternative in this case is to convolve the image with a matrix of ones, and then simply take every-other entry using standard indexing (i.e. `y=x(1:2:end,1:2:end)`). Although convolution in this case actually computes four times as many outputs (assuming 2x downsampling) as we really need, this method is still (empirically) an order of magnitude or so faster than the previously mentioned approaches.

Most authors, it seems, implement the sigmoid activation function and it's derivative using `inline` function definitions. At the time of this writing, "inline" MATLAB function definitions are *not* at all like C macros, and take a huge of amount of time to evaluate. Thus, it is often worth it to simply replace all references to f and f' with the actual code. There is of course a tradeoff between optimizing your code and maintaining readability.

4 Practical Training Issues (Incomplete)

4.1 Batch vs. Online Updates

Stochastic descent vs. batch learning.

4.2 Learning Rates

LeCun's stochastic online method (diagonal approx to the hessian). Is it worth it? Viren's idea: at least have a different rate for each layer, because gradients at the lower layers are smaller and less reliable. LeCun makes similar statements in [5].

4.3 Choice of Error Function

Squared error (MLE), vs. cross-entropy. The latter can be more effective for some classification tasks [2].

4.4 Checking Your Work with Finite-Differences

Finite-differences can be an indispensable tool when it comes time to verify that you've got your backpropagation imeplementation (or derivation) correct. It is remarkably easy to make many mistakes and still have a network that appears to be learning something. Checking the gradients your code produces against finite difference estimates is one way to make sure you don't have any errors:

For a single input pattern, estimate the gradients using second-order finite differences

$$\frac{\partial E}{\partial w_i} \approx \frac{E(w_i + \epsilon) - E(w_i - \epsilon)}{2\epsilon}$$

and check against the gradients your backpropagation code is returning. Epsilon should be small, but not too small to cause numerical precision problems. Something like $\epsilon = 10^{-8}$ could be ok. Note that using finite differences to train the network is wildly inefficient (i.e. $O(W^2)$ for W weights in the network); the $O(W)$ speed advantage of backpropagation is well worth the hassle.

Acknowledgments

The author thanks Wen Zhou, Rasmus Berg Palm, Ming Yuan, and Vibhu Agarwal for pointing out typos in the previous version of these notes.

References

[1] C.M. Bishop,"Neural Networks for Pattern Recognition", Oxford University Press, New York, 1995.

[2] F.J. Huang and Y. LeCun. "Large-scale Learning with SVM and Convolutional for Generic Object Categorization", In: *Proc. 2006 IEEE Computer Society Conference on Computer Vision and Pattern Recognition*, vol. 1, pp. 284-291, 2006.

[3] Y. LeCun, B. Boser, J. Denker, D. Henderson, R. Howard, W. Hubbard, and L. Jackel. "Backpropagation applied to handwritten zip code recognition", *Neural Computation*, 1(4), 1989.

[4] Y. LeCun, L. Bottou, Y. Bengio, and P. Haffner."Gradient-based learning applied to document recognition", *Proceedings of the IEEE*, vol. 86, pp. 2278–2324, November 1998.

[5] Y. LeCun, L. Bottou, G. Orr, and K. Muller. "Efficient BackProp", in: *Neural Networks: Tricks of the trade*, G. Orr and K. Muller (eds.), "Springer", 1998.

[6] J.P. Rauschecker and B. Tian. "Mechanisms and streams for processing of 'what' and 'where' in auditory cortex," *Proc. Natl. Acad. Sci. USA*, 97 (22), 11800–11806, 2000.

[7] T. Serre, M. Kouh, C. Cadieu, U. Knoblich, G. Kreiman and T. Poggio. "A Theory of Object Recognition: Computations and Circuits in the Feedforward Path of the Ventral Stream in Primate Visual Cortex", CBCL Paper #259/AI Memo #2005-036, Massachusetts Institute of Technology, October, 2005.

[8] T. Serre, A. Oliva and T. Poggio. "A Feedforward Architecture Accounts for Rapid Categorization", *Proc. Natl. Acad. Sci. USA*, (104)15, pp.6424-6429, 2007.

[9] S. Shamma. "On the role of space and time in auditory processing," *TRENDS in Cognitive Sciences*, Vol. 5 No. 8, 2001.

[10] P.Y. Simard, Dave Steinkraus, and John C. Platt. "Best Practices for Convolutional Neural Networks Applied to Visual Document Analysis", *Proceedings of the International Conference on Document Analysis and Recognition*, pp. 958-962, 2003.

[11] F.E. Theunissen, K. Sen, and A. Doupe, "Spectral-temporal receptive fields of nonlinear auditory neurons obtained using natural sounds," *J. Neuro.*, Vol. 20, pp.2315–2331, 2000.

[12] D. Zoccolan, M. Kouh, J. DiCarlo and T. Poggio. "Tradeoff between selectivity and tolerance in monkey anterior inferior temporal cortex", *J. Neurosci.*, 2007.