A Novel Activation Function for a Convolutional Neural Network Implemented Without Machine Learning Libraries

Aresh Pourkavoos
Loomis Chaffee: Grade 10

March 10, 2019

Abstract

Artificial neural networks apply a nonlinear activation function to all numbers in a given layer of neurons, represented by a tensor, before the affine transformation in the next layer. By alternating nonlinear and linear functions, they introduce nonlinearity into what otherwise would be only a linear regression model. This project proposes a new activation function for convolutional neural networks (CNNs) to better balance training speed and classification accuracy than three of the currently-used functions. The new function was tested to classify images of handwritten digits in the MNIST dataset. The general structure of the CNN emulates that of Karpathy's ConvNetJS demo. I coded this entire convolutional neural network in Python using only NumPy's array functionality without the aid of machine learning libraries. Four different activation functions were compared. The rectified linear unit (ReLU), $max(0, x)$, computes quickly but has derivative zero over negative numbers. The softplus function, $ln(1 + e^x)$, approximates ReLU and, although less efficient, has a positive gradient everywhere. The logistic function, $1/(1 + e^{-x})$, produces less accurate results than ReLU and was mainly included to test the validity of this experiment. I hypothesized that the new function, $(x + \sqrt{1 + x^2})/2$, which resembles softplus, would have a comparable accuracy to the same while computing more quickly. Twelve CNNs, three using each activation function, were trained on MNIST for 50 epochs using the ADADELTA adaptive rate method. This took 25.9 hours of IGPU time. Although slightly less accurate than the others, the new function, as hypothesized, ran faster than all but ReLU.

1 Introduction

1.1 What is a neural network?

Artificial neural networks (ANNs) are mathematical models meant to mimic the processes of neurons manipulating data in the brains of humans and other

animals. They have shown use in a wide variety of tasks, from voice recognition to self-driving cars to generation of realistic images of human faces. There are many types of ANNs specialized for different problems, but virtually all of them perform regression analysis; given a set of inputs and outputs, the network needs to "learn" to produce the given outputs when fed the given inputs.

It is able to tweak the results it produces using various free parameters. In linear regression, for example, in order to produce a desired y given some x, it has the freedom to adjust the slope m and y-intercept b of the fitting line, $y = mx+b$. The free parameters of an ANN are analogous to m and b, but ANNs operate on vectors, matrices, or higher-dimensional *tensors*, not just individual numbers. Additionally, to write the full equation expressing the output in terms of the input and free parameters would be cumbersome in most cases.

As in linear regression, an ANN defines the "best fit" as the set of free parameters which minimizes some *error* - the difference between the expected output y and the output of the model $\hat{y}$. Linear regression usually defines the error as the sum of squared differences between y and $\hat{y}$, but ANNs have other options as well. The main difference, though, lies in how the optimal free parameters are found. In linear regression with least-squares error, there is an equation which will simply return the best values of m and b. In ANNs, on the other hand, the process is not so simple and will be discussed later under "Backpropagation".

1.2 Making a prediction

Before backpropagation, though, we must first understand *forward propagation*, the process the network uses to output a $\hat{y}$ given an X. This project uses a convolutional neural network (CNN), but in order to fully grasp its inner workings, one should look at a simpler type of ANN, a *fully-connected* neural network.

Fully-connected ANNs, like animal brains, are made of *neurons*, which store information, and *synapses*, which relay signals between neurons. Each neuron has an *activation*, the strength of the signal it emits, and each synapse has a *weight*, the factor by which it dampens, amplifies, or even inverts the signal it carries. Both activations and weights are real numbers.

Neurons are grouped into *layers*: an *input* layer to store X, some number of *hidden* layers to act as intermediate stages, and an *output* layer for $\hat{y}$. Because every neuron stores a real number, *every layer stores a vector* whose number of components equals the number of neurons in that layer. The activations of the input layer are set according to the components of the vector X, and the data move through the hidden layers to the output layer. (These components are usually normalized to put them in or near the range [0, 1], especially for large numbers like years. Because the this project's problem does not require normalization, though, it will not be discussed here.)

1.3 A specific example

The iris dataset is a sample of 150 irises collected by biologist Edgar Anderson in the mid-1930s. For each iris, the dataset contains measurements of its sepal length, sepal width, petal length, petal width, and classification into one of three species, *Setosa*, *Virginica*, or *Versicolor*. The goal of the network trained on this dataset is to identify a flower based on its dimensions. The input layer would thus have four neurons, one for each measurement. How many neurons should the output layer have, though?

1.3.1 Formatting output with softmax

In a "normal" regression problem, where the network would need to output not a choice but some number of independent traits, the output layer would be fairly straightforward. If the network were to predict the height of the stem and depth of the roots from the four characteristics above, there would need to be two output neurons, one for each output trait.

It takes a bit more thought to arrive at a good way to represent solutions to a classification problem. For a start, the output layer might need just one neuron. If every species were assigned a number - say, 0 for *Setosa*, 1 for *Virginica*, and 2 for *Versicolor* - the output could be interpreted as the number to which it is closest. There are two main issues with this approach, however. The first is inherent bias against classifying as the central number. *Setosa* and *Versicolor* would be classified for any output in the infinite ranges $(-\infty, 0.5)$ and $(1.5, \infty)$, respectively, but the network would need to hit $(0.5, 1.5)$ in order to output *Virginica*. The second is that a one-dimensional output does not capture all types of uncertainty between three choices. For example, what if a certain flower looked somewhere in between a *Setosa* and a *Versicolor*? The output should be between the two corresponding ranges, but that would put it right in the *Virginica* range, even if the input looked nothing like the latter.

What we need instead is for the network to return three *probabilities*, each representing the confidence that the flower is of one species. How can we ensure that the numbers meet the definition of "probabilities," i.e. they are nonnegative and sum to 1? A normal "regression-type" network can return any three real numbers in the output layer.

Enter the softmax function σ. It meets both of the above requirements in turn, first by taking the exponential of each number to ensure positivity, then by dividing each by the sum of all exponentials to reach a sum of 1. In mathematical notation, applying softmax to a vector v with n components,

$$\sigma(v)_j = e^{v_j} / \sum_i e^{v_i}$$

1.4 Forward propagation

The names used in this paper for all variables will be:
Input activations: X

Weight matrix: W
Bias vector: b
Pre-activation state: z
Activation function: f
Activation state: a
Output activations: $\hat{y}$
Expected output: y
Error: J

Some of these variables have multiple instances in a single network and will be indexed by a superscripted number in parentheses (e.g. the first weight matrix is $W^{(1)}$).

To summarize the process of one layer, every neuron in a given layer (except for the output layer) is connected to every other neuron in the next layer by a weight, making the network "fully-connected". Each weight takes the activation of the neuron at its input, multiplies the activation by its strength, and feeds the result into the neuron at its output. The neurons in the next layer take all of the numbers that the weights feed it and add them, along with one more value, called a *bias* b. This paper will call this sum the *pre-activation state* of the neuron. (Although it may be called by other names, if such names exist, the researcher was unable to find them.) Each neuron then applies an *activation function* to the pre-activation state to get the activation of the neuron in the next layer.

The network to classify irises consists of three layers: an input layer with four neurons, one hidden layer with five, and an output layer with three. (Neural networks often have multiple hidden layers, and those which do are called *deep* networks. The CNN in this project is deep.) X is the activation of the input layer (or, equivalently, the first layer $a^{(1)}$). Because the next layer has five neurons, the weights of the synapses between them $W^{(1)}$ are represented as a 4-by-5 matrix.

The pre-activation states of the hidden layer $z^{(2)}$ are determined by X, $W^{(1)}$, and the bias $b^{(1)}$. Let's look at the pre-activation of a particular hidden neuron, $z_j^{(2)}$. For every neuron in X X_i, there is a synapse $W_{ij}^{(1)}$ connecting it to $z_j^{(2)}$. The weight of this synapse multiplied by the activation X_i is added to $z_j^{(2)}$ for all i, along with the corresponding bias $b_j^{(1)}$. Altogether,

$$z_j^{(2)} = \sum_i (X_i W_{ij}^{(1)}) + b_j^{(1)}$$

for all j. The sum term is that of pairwise products, with X_i for all i making up a row vector and $W_{ij}^{(1)}$ for all i making a column of the matrix. This sum of pairwise products is the same as a dot product, and the dot product of a row and a column is exactly the process of matrix multiplication:

$$z^{(2)} = XW^{(1)} + b^{(1)}$$

This is an *affine transformation* - a linear transformation (represented by matrix multiplication) followed by a translation (when the bias is added). Skipping

the activation function would be no better than having just one network layer because composing linear functions alone results only in another linear function. The activation function is applied to each element individually in order to introduce *nonlinearity* into the regression model and to give neural networks their power:

$$a^{(2)} = f(z^{(2)})$$

1.4.1 Choosing an activation function

The earliest neural networks used *perceptrons*, neurons whose activation function was 0 if $x \leq 0$ and 1 otherwise. The problem with these neurons was that the function is discontinuous, so making small changes to weights could have unpredictable effects. It was soon replaced with a smoother variant, the *logistic function*, a *sigmoid* or S-shaped curve with the equation

$$f(x) = 1/(1 + e^{-x})$$

The logistic function was the go-to for neural networks until 2011, when a new function was shown to outperform it while being much easier to calculate. This function is the *rectified linear unit*, or $ReLU$, defined as

$$f(x) = max(0, x), \text{ or equivalently, } 0 \text{ if } x \leq 0 \text{ and } x \text{ otherwise.}$$

The same paper also proposed a smooth approximation to ReLU, the *softplus* function, with equation

$$f(x) = ln(e^x + 1).$$

The author of the paper noted that ReLU's ability to easily return an activation of 0 was more biologically accurate than previous functions, as relatively few neurons are firing in a human brain at any moment. (This sparsity does not, however, support to any degree the myth that we only use ten percent of our brains, because virtually all neurons are used over longer periods.)

1.5 Forward propagation (cont.)

To restate the two equations of the iris network involved in forward propagation so far,

$$z^{(2)} = XW^{(1)} + b^{(1)}$$
$$a^{(2)} = f(z^{(2)})$$

f could be any of the activation functions mentioned. To get the pre-activation states of the neurons in the output layer, another affine transformation is needed with new free parameters:

$$z^{(3)} = a^{(2)}W^{(2)} + b^{(2)}$$

Finally, the softmax function is applied to give a classification:

$$\hat{y} = \sigma(z^{(3)})$$

These four equations encapsulate the full "$y = mx + b$" for this network. The question remains, though: How do we find the free parameters?

1.6 Backpropagation

Neural networks learn via *gradient descent*, a technique used to find local minima of a function, and they apply it to find a minimum of error J. Gradient descent begins by setting the free parameters randomly and then incrementally changes them to decrease the output of the function being minimized. To know whether to tune each parameter up or down, and by how much, gradient descent needs the *partial derivatives* of J, one with respect to *each free parameter*. The challenge of training a neural network lies in finding these partial derivatives, and the algorithm which does so is called backward propagation, *backpropagation*, or backprop.

To begin, we need to define J. A method often used to compare two probability distributions is the *cross entropy* between them,

$$J = -\sum_i y_i ln(\hat{y}_i)$$

Neural networks are essentially a large nested function from input to output, and taking the derivative of such a function requires *repeated application of the Chain Rule.* Our goal for this network is to find

$$\frac{\partial J}{\partial W^{(1)}},\ \frac{\partial J}{\partial b^{(1)}},\ \frac{\partial J}{\partial W^{(2)}}, \text{ and } \frac{\partial J}{\partial b^{(2)}}$$

so that we can modify each of the free parameters to decrease J and thus improve the network's accuracy. The equation for J, though, doesn't depend directly on any of these free parameters, but it does depend on $\hat{y}$, which in turn depends indirectly on all of the weights and biases. Let's begin by finding $\frac{\partial J}{\partial \hat{y}}$, which will have the same shape as $\hat{y}$ itself. Because each element of $\hat{y}$, $\hat{y}_i$ appears in only one term of the sum, the i-th term, when taking $\frac{\partial J}{\partial \hat{y}_i}$, we need only focus on the i-th term because everything else is considered constant. $\frac{d}{d\hat{y}_i} y_i ln(\hat{y}_i)$ is, by the derivative of the natural logarithm and the constant multiple rule, $y_i/\hat{y}_i$. Applying this for all i and remembering the negative sign,

$$\frac{\partial J}{\partial \hat{y}} = -\frac{y}{\hat{y}}$$

with division being performed elementwise. The equation for $\hat{y}$ depends only on $z^{(3)}$, to which the softmax function is applied. Taking the derivative here and applying the Chain Rule to find $\frac{\partial J}{\partial z^{(3)}}$ is tricky because changing one element of $z^{(3)}$ will affect all elements of $\hat{y}$. To learn more about the derivative of softmax, see "The Softmax Function and its Derivative."

Instead, let's assume we know $\frac{\partial J}{\partial z^{(3)}}$ and try to find the gradients we need, $\frac{\partial J}{\partial b^{(2)}}$ and $\frac{\partial J}{\partial W^{(2)}}$. Because $b^{(2)}$ is simply added to $z^{(3)}$ with no other processing, its derivative is the same as that of $\frac{\partial J}{\partial z^{(3)}}$.

$$\frac{\partial J}{\partial b^{(2)}} = \frac{\partial J}{\partial z^{(3)}}$$

One can think of it as that whether some value is added to $z^{(3)}$ directly or the same value is added to $b^{(2)}$ and then forward-propagates to affect $z^{(3)}$, the change

in J from the operation will be the same. To find $\frac{\partial J}{\partial a^{(2)}}$, look at a particular element $a_i^{(2)}$, which feeds all elements of $z^{(3)}$ through $W^{(2)}$. $a_i^{(2)}$'s influence on J through some $z_j^{(3)}$ is equal to $\partial z_j^{(3)}/\partial a_i^{(2)} = W_{ij}^{(2)}$ times $\frac{\partial J}{\partial z^{(3)}}_j$. However, $a_i^{(2)}$ influences J through *all* elements of $z^{(3)}$, so its cumulative effect is the sum of all of these:

$$\frac{\partial J}{\partial a_i^{(2)}} = \sum_j \frac{\partial J}{\partial z_j^{(3)}} W_{ij}^{(2)}$$

This sum is the dot product of a row (the gradients of $z^{(3)}$) and another row (the i-th row of $W^{(2)}$). To simplify the equation and write it in terms of matrix multiplication, this can be turned into the dot product of a row and a column by transposing $W^{(2)}$:

$$\frac{\partial J}{\partial a^{(2)}} = \frac{\partial J}{\partial z^{(3)}} W^{(2)^T}$$

$a^{(2)}$ does not depend directly on any free parameters, but it depends on $z^{(2)}$, which depends on $W^{(1)}$ and $b^{(1)}$ in turn. Thus, to find $\frac{\partial J}{\partial W^{(1)}}$ and $\frac{\partial J}{\partial b^{(1)}}$, we must first find $\frac{\partial J}{\partial z^{(2)}}$. $a_i^{(2)} = f(z_i^{(2)})$ for all i, so $da_i^{(2)}/dz_i^{(2)} = f'(z_i^{(2)})$. By the Chain Rule,

$$\frac{\partial J}{\partial z_i^{(2)}} = \frac{\partial J}{\partial a_i^{(2)}} \frac{da_i^{(2)}}{dz_i^{(2)}} = \frac{\partial J}{\partial a_i^{(2)}} f'(z_i^{(2)})$$

Stated without indices,

$$\frac{\partial J}{\partial z^{(2)}} = \frac{\partial J}{\partial a^{(2)}} \circ f'(z^{(2)})$$

where $\circ$ denotes the elementwise product or Hadamard product (as opposed to the dot product or matrix product). The equations for $\frac{\partial J}{\partial W^{(1)}}$ and $\frac{\partial J}{\partial b^{(1)}}$ are virtually the same as those for $\frac{\partial J}{\partial W^{(2)}}$ and $\frac{\partial J}{\partial b^{(2)}}$ after replacing $z^{(3)}$ with $z^{(2)}$ and $a^{(2)}$ with X, essentially shifting everything back one layer.

1.6.1 For CNNs

In a fully-connected ANN like the one just discussed, the pre-activation states are decided as a sum of products of weights and activations of the previous layer, all of these being real numbers. In a convolutional network, however, neurons do not store real numbers but *images*, matrices of pixel values. Every layer is thus not a vector, as in a fully-connected network, but a three-dimensional tensor (a vector of matrices). Similarly, synapses do not store individual numbers but *kernels*, small matrices (5x5 in this project). This makes every array storing them *four*-dimensional - two numbers to index which neurons are connected (as before) and two to index a particular pixel within the matrix. Instead of multiplying activations with weights, images and kernels are *convoluted* and then added to get the pre-activation values in the next layer. Kernels can define

"filters" on the image or detect particular features. For example, a 3-by-3 kernel which blurs the image with which it is convoluted is

$$\begin{bmatrix} 1/9 & 1/9 & 1/9 \\ 1/9 & 1/9 & 1/9 \\ 1/9 & 1/9 & 1/9 \end{bmatrix}$$

by averaging the values of the 9 neighbors of every pixel. Additionally, after applying the activation function, an extra step of *pooling* is required, which scales the image down so that large-scale features can be detected. Convolutions have a *receptive field* only as big as the matrices themselves, so to be able to distinguish things like loops or stems in the context of digit recognition, the image must be scaled down so that the convolutional kernel covers a wider area of the image.

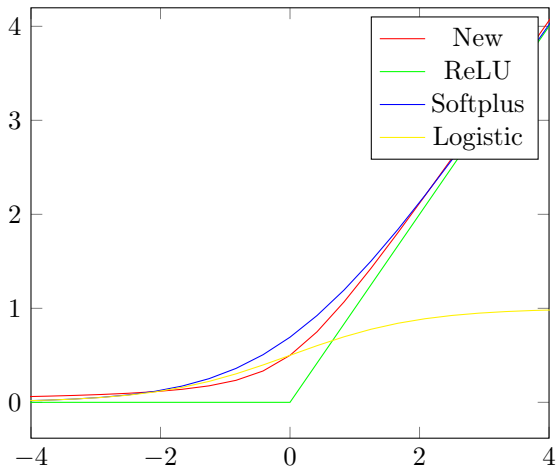
In CNNs, a bit of symmetry which exists in fully-connected networks is broken in that forward- and backpropagation do not both use convolution like fully-connected ANNs use matrix multiplication both ways. Instead, backpropagation in CNNs relies on *correlation*, which rotates the kernel 180 degrees before convoluting. Backpropagation for pooling is performed by *scaling up the images* of the gradients of a post-pooling layer.

2 Question and hypothesis

How does the new activation function

$$f(x) = \frac{x + \sqrt{x^2 + 1}}{2}$$

compare to the ReLU, softplus, and logistic functions in terms of training efficiency and classification accuracy when used in a CNN to recognize digits in MNIST?



It should run faster than softplus and logistic for two reasons. First, it does not involve logarithms or exponentials, which are computationally intensive compared to square roots. The logistic function, though, is the solution to the differential equation

$$f'(x) = f(x)(1 - f(x)),$$

making its derivative, which is necessary for backpropagation, quick to compute, given that $f(x)$ is already known. The new function, however, solves a differential equation too,

$$f'(x) = f(x)/(2f(x) - x)$$

keeping its total computation time ahead of softplus.

Also, because it looks similar to softplus, their accuracy should be roughly similar. As noted before, previous research has demonstrated that ReLU should improve upon sigmoids like the logistic function, so the latter was included to test the validity of the experiment.

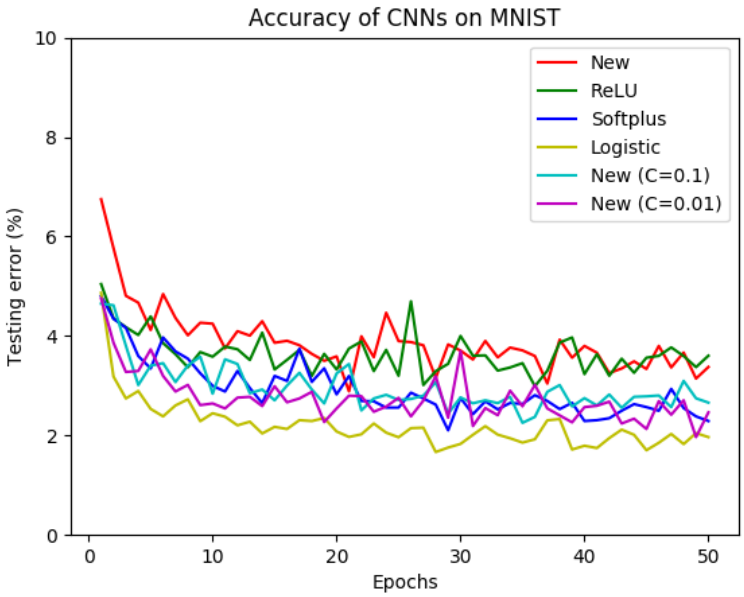
3 Materials and Methods

The network was implemented in Python and did not use any machine learning libraries for forward or backpropagation. All of the math was implemented with NumPy's array functions, including but not limited to matrix multiplication for fully-connected layers, Einstein summation for convolutional layers, and Kronecker multiplication for pooling layers.

The network's input layer is a NumPy array with shape (24, 24, 1, 20), representing the 24-by-24 pixel input image accepted, the existence of only 1 image (neuron) in that layer, and the network's batch size of 20. The first weight tensor has shape (5, 5, 1, 8), with 5-by-5 kernels connecting the single input image to each of the eight hidden neurons. Because this changes neither the image dimension nor the batch size, the first hidden layer consists of a (24, 24, 8, 20) array, to which f is applied. This layer is then average-pooled by a factor of 2 to get a (12, 12, 8, 20) array. The next operation is another convolution with (5, 5, 8, 16) weights to reach a (12, 12, 16, 20) hidden layer, followed by another activation and then a pooling by 3. The pooled layer, with a shape of (4, 4, 16, 20), has each of its 256 pixels (per example) connected by a (256, 10) weight matrix to each neuron in the output (10, 20).

Twelve instances of the network were initialized, three with each of the four activation functions, and they were trained one by one on an Intel Core i5 processor with an HD Graphics 530 integrated GPU. Each was trained for 50 epochs of the MNIST training data, recording its accuracy on the testing data after each epoch. The total training time was measured for each network as well.

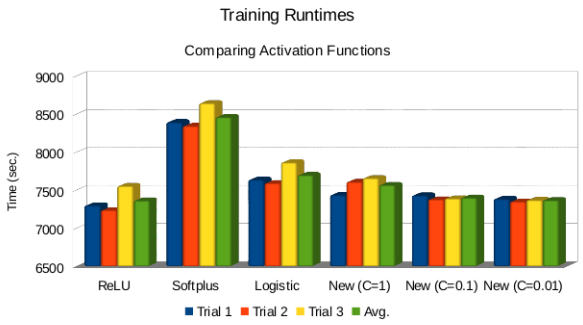
4 Results



The above graph depicts the performance of the networks over time, measured by the percent of testing images classified incorrectly after each epoch. The scatterplots represent the three trials, and the lines graph the average error for each function.

Surprisingly, the logistic function achieved the highest accuracy out of all four. The new function with C=1 achieved approximately the same accuracy as ReLU. However, setting C to 0.1 or 0.01 improved the new function, putting it on par with softplus.

	Trial 1	Trial 2	Trial 3	Avg.
ReLU	7297	7237	7554	7363
Softplus	8392	8342	8642	8459
Logistic	7636	7591	7863	7697
New (C=1)	7435	7608	7657	7567
New (C=0.1)	7430	7378	7390	7399
New (C=0.01)	7384	7349	7369	7367



The proposed function beat all but ReLU in terms of training speed on MNIST.

5 Conclusion and Discussion

The relatively small time difference between trials reflects the fact that most of the training time is spent calculating the convolutions. The number of convolutions per layer increases proportionally to the *square* of the neurons in each layer, whereas the number of activations only grows linearly.

Nonetheless, the new function presented some advantage over each of the three functions for C=0.1 and C=0.01: an accuracy advantage over ReLU and a speed advantage over softplus and logistic. This makes it potentially useful for neural networks.

I propose to name this function the *hyperbolic rectifier unit* or HRU, because its graph is that of a hyperbola, and its asymptotes, $y = 0$ and $y = x$, are the two lines which define ReLU. The HRU could be tested on other problems or against other activation functions, like the arctangent, hyperbolic tangent, or exponential linear unit (ELU).

References

[1] LeCun, Yann; Corinna Cortes; Christopher J.C. Burges. "MNIST handwritten digit database". 2010, yann.lecun.com/exdb/mnist/.

[2] Karpathy, Andrej. "ConvNetJS MNIST demo". ConvNetJS, Stanford University, cs.stanford.edu/people/karpathy/convnetjs/.

[3] Welch Labs. "Neural Networks Demystified". Online video playlist. YouTube. 4 Nov 2014, youtu.be/bxe2T-V8XRs.

[4] The GIMP Development Team, "8.2. Convolution Matrix". 2015, docs.gimp.org/2.8/en/plug-in-convmatrix.html.

[5] Xavier Glorot, Antoine Bordes and Yoshua Bengio. "Deep sparse rectifier neural networks". AISTATS. 2011.

[6] Eli Bendersky. “The Softmax function and its derivative”. 18 Oct 2016, eli.thegreenplace.net/2016/the-softmax-function-and-its-derivative/.

[7] Matthew D. Zeiler. “ADADELTA: An Adaptive Learning Rate Method”. 22 Dec 2012.