What is a Neural Network?

A neural network is a regression model which "learns" to provide some output given some input. A simple regression model is the linear equation $y = mx + b$, which gives an output y based on an input x using the free parameters m and b. The regression analysis problem consists of finding the values of m and b which return the desired output when given the input. Likewise, neural networks have inputs, outputs, and free parameters, but these take the form of *tensors* - vectors, matrices, and 3- or higher-dimensional arrays of numbers.

Forward Propagation

A neural network consists of *neurons* which send signals through *synapses*. Each neuron has an *activation* representing the strength of the signal it emits, and each synapse has a *weight* representing how it modifies the signal it carries. Neurons are grouped into *layers*. The prediction process begins by manually setting the activations of the first layer X. The activations of the second layer $a^{(2)}$ are calculated using the first weights $W^{(1)}$, additional values called the *bias* $b^{(1)}$, and the *activation function* f:

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

where

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

This process is repeated to find the activations of the next layers. The activations of the last layer $\hat{y}$ represent the output of the network.

Backpropagation

Neural networks learn by finding the W and b matrices which, when given an input X, provide the output $\hat{y}$ which closely matches the expected output y. The network calculates an *error* J to quantify how well the model fits the data. In this problem, J is the *cross entropy* between y and $\hat{y}$:

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

J is minimized through *gradient descent*. Gradient descent initializes the free parameters randomly, then finds the *partial derivative* of J with respect to each free parameter to ascertain how changing each parameter will affect J. The value of $\partial J/\partial \hat{y} = -y/\hat{y}$ and the Chain Rule are used to find the derivatives with respect to the previous layers' activations. For example, once $\partial J/\partial a^{(2)}$ is found, we can calculate

$$\frac{\partial J}{\partial W^{(1)}} = X^T \cdot (\frac{\partial J}{\partial a^{(2)}} \circ f'(z^{(2)}))$$

This result is used to adjust the values inside $W^{(1)}$ so that J decreases, making $\hat{y}$ closer to y. By making many incremental adjustments, the error J approaches a local minimum.

Convolutions

Matrix multiplication is performed by adding products of pairs of numbers and is used by *fully-connected* neural networks, whose neurons and weights each store one real number. In *convolutional* neural networks (CNNs), however, each neuron stores an *image*, a matrix of pixel values, and each synapse stores a *kernel*, a smaller matrix defining a "filter" on the image. Instead of multiplying the activations and weights for forward propagation, the images and kernels are *convoluted*, applying the filter to the image. Convolution can emulate the filters of blurring, sharpening, edge detection, exposure adjustment, etc., which are useful for image processing and feature recognition.

While backpropagation in fully-connected networks uses matrix multiplication just as in forward propagation, CNNs use *correlation* in backpropagation, which is equivalent to rotating the kernel by 180 degrees before convolution.

Why Activation Functions?

An activation function is a function applied between layers of the network. Since convolution is a linear function on pixel values, composing multiple convolutions will only result in another linear function. **The activation function f is thus needed to introduce *nonlinearity* into the regression model.** Commonly used functions include:

- The Rectified Linear Unit, ReLU - the most widely used function, the most efficient, and fairly accurate: $f(x) = max(0, x)$
- The softplus function, a smooth approximation of ReLU: $f(x) = ln(1 + e^x)$
- The logistic function, an improvement on the step function used in the earliest neural networks: $f(x) = 1/(1 + e^{-x})$

All functions differ in efficiency of computation and accuracy of results when used in a neural network.

The New Function

It would be nice to find a new function which

- smoothly approximates ReLU, like softplus, but
- together with its derivative, is more efficient to compute.

The derivative of the logistic function, for example, is $f'(x) = f(x)(1 - f(x))$ and computes more quickly compared to $f'(x) = e^x/(1 + e^{-x})^2$ because $f(x)$ has already been calculated. This project proposes the following function, where C is an adjustable value, to meet these requirements:

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

Question/Hypothesis

How does the proposed function compare with the ReLU, softplus, and logistic functions when used in a convolutional neural network to recognize images of handwritten digits in the MNIST dataset?

Since its graph appears similar to softplus, the new function should achieve a similar accuracy to softplus, and above that of ReLU and the logistic function. In addition, since it contains no exponentials or logarithms and is the solution to the differential equation:

$$f'(x) = \frac{f(x)}{2f(x) - x}$$

it should compute faster than either softplus or logistic.

Materials and Methods

The CNN was implemented in Python from scratch (!) without the aid of machine learning libraries such as Tensorflow, Theano, or Neon. Instead, the forward- and backpropagation were implemented using **Numpy**'s array functionality.

The instances of the CNN were trained on an Intel Core i5 with integrated GPU running Ubuntu. Three CNNs were trained for each function: new (C=1), ReLU, softplus, and logistic, for 50 epochs on the MNIST training dataset of 60000 images. After each epoch, the networks were evaluated on all 10000 images of the testing dataset to produce a percent error rate. These error rates were averaged across the trials, and the training time for each network was measured. **Additionally**, three CNNs were later trained for the new function with C=0.1 and for C=0.01.

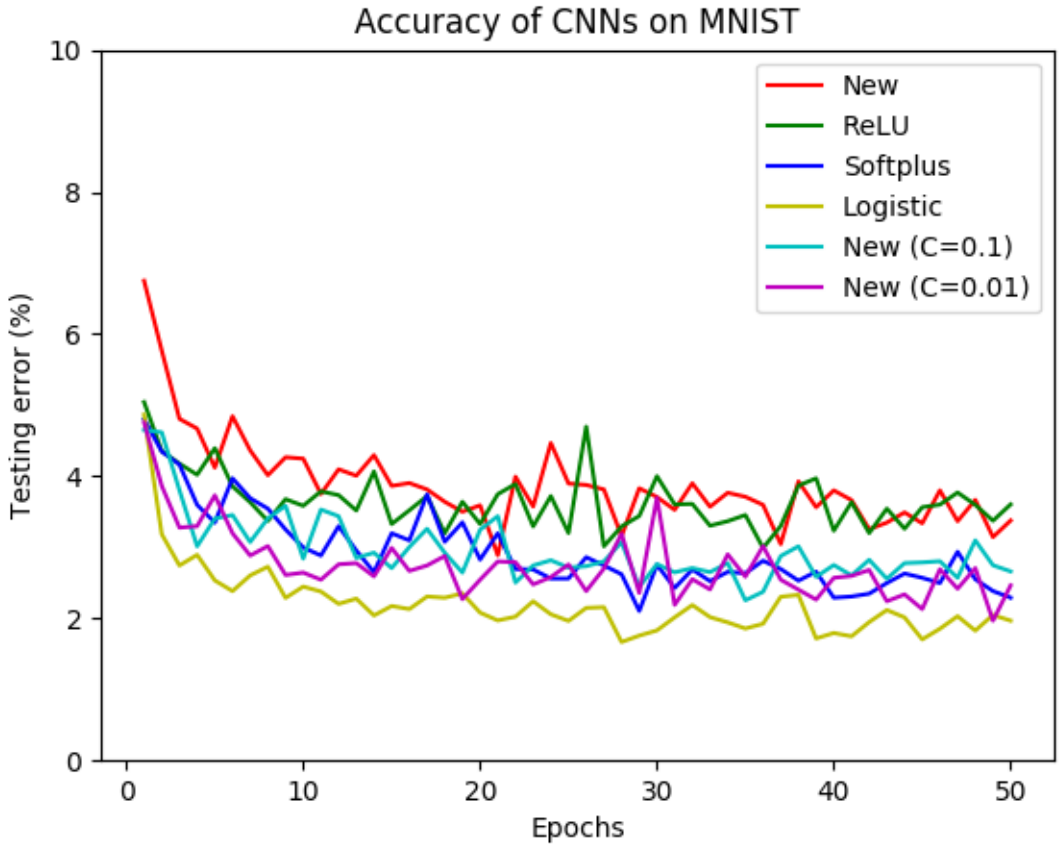
The layers of the CNN consist of the following tensors and operations:

$$(24, 24, 1) \xrightarrow{conv/act} (24, 24, 8) \xrightarrow{pool} (12, 12, 8) \xrightarrow{conv/act} (12, 12, 16) \xrightarrow{pool} (4, 4, 16) \xrightarrow{fc/softmax} (10)$$

The tuples of numbers represent the dimensions of the tensors containing the pixel values.

- "Conv/act" stands for a set of convolutions followed by the activation function.
- "Pool" stands for a pooling operation, scaling the image down to allow for distinguishing larger-scale features.
- "FC/softmax" stands for a fully-connected layer plus the softmax function to obtain ten probabilites, each representing the confidence of the CNN that the image depicts a particular digit.

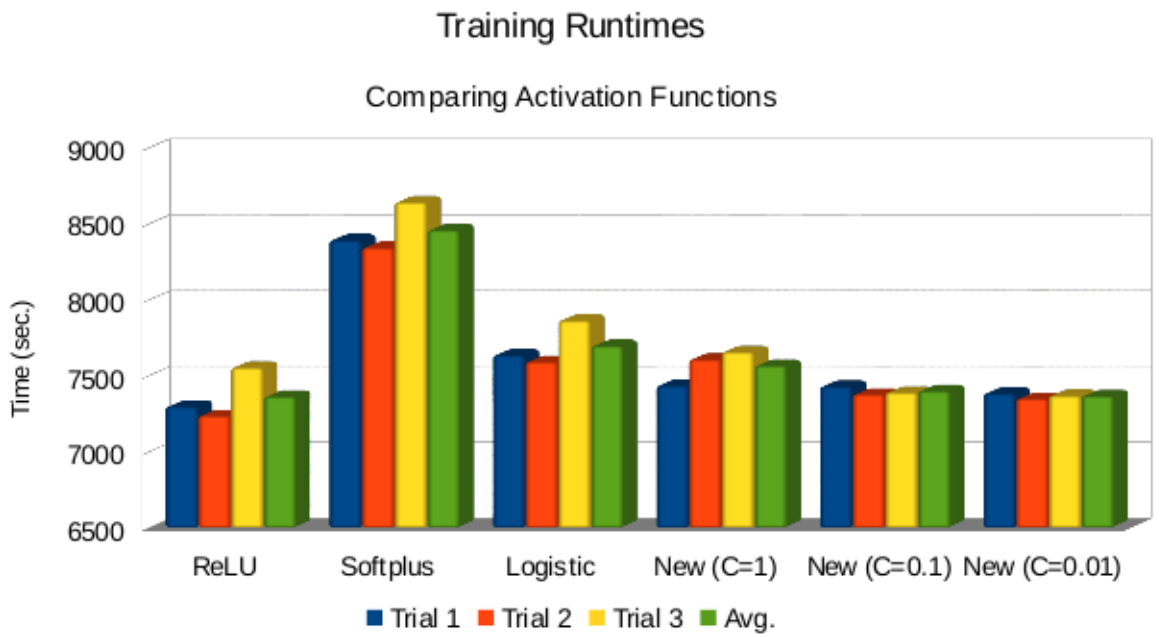
Results: Accuracy



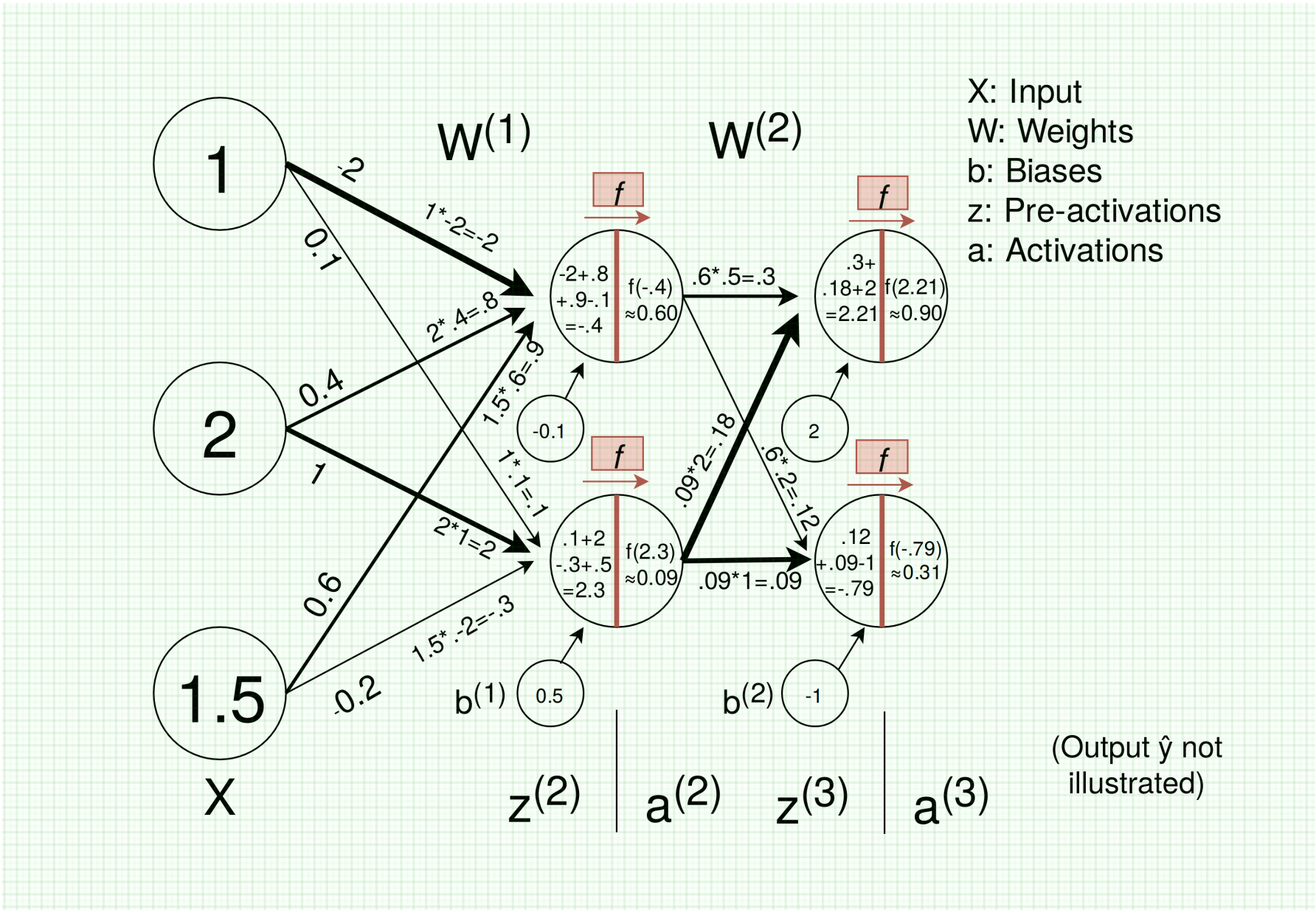
In the initial trials (C=1), the new function had an accuracy similar to ReLU (96%), but both softplus and logistic were more accurate. For C=0.1 and C=0.01, though, the new function **surpassed ReLU** in accuracy and closely matched softplus (97%), as hypothesized. However, the logistic function (98%) remained the most accurate.

Results: Speed

	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 networks' training times ranked as hypothesized from fastest to slowest: ReLU, the **proposed function**, logistic, softplus. Despite the running times of the functions alone spanning multiple orders of magnitude, the difference between networks is relatively small because most of the training time is spent performing convolutions.


X: Input
W: Weights
b: Biases
z: Pre-activations
a: Activations
W(1)
W(2)
1
2
1.5
X
-2
0.1
0.4
1
0.6
-0.2
1*-2=-2
2*.4=.8
1.5*.6=.9
1*.1=.1
2*1=2
1.5*-.2=-.3
-2+.8
+.9-.1
=-.4
f(-.4)
≈0.60
.1+2
-.3+.5
=2.3
f(2.3)
≈0.09
.6*.5=.3
.09*2=.18
.6*.2=.12
.09*1=.09
.3+
.18+2
=2.21
f(2.21)
≈0.90
.12
+.09-1
=-.79
f(-.79)
≈0.31
-0.1
2
0.5
-1
b(1)
b(2)
z(2)
a(2)
z(3)
a(3)
(Output ŷ not illustrated)

$$\left([1 \;\; 2 \;\; 1.5] \cdot \begin{bmatrix} 0.6 & 0.2 \\ 0.4 & 1 \\ -2 & 0.1 \end{bmatrix} \right) + [-0.1 \;\; 0.5] = [-0.4 \;\; 2.3] \qquad XW^{(1)} + b^{(1)} = z^{(2)}$$

$$[f(-0.4) \;\; f(2.3)] = [0.6 \;\; 0.09] \qquad f(z^{(2)}) = a^{(2)}$$

$$\left([0.6 \;\; 0.09] \cdot \begin{bmatrix} 0.5 & 0.2 \\ 2 & 1 \end{bmatrix} \right) + [2 \;\; -1] = [2.21 \;\; -0.79] \qquad a^{(2)}W^{(2)} + b^{(2)} = z^{(3)}$$

$$[f(2.21) \;\; f(-0.79)] = [0.9 \;\; 0.31] \qquad f(z^{(3)}) = a^{(3)}$$

$$\frac{\partial J}{\partial z^{(3)}} = \frac{\partial J}{\partial a^{(3)}} \circ f'(z^{(3)})$$

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

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

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

$$\frac{\partial J}{\partial W^{(1)}} = X^T \cdot \frac{\partial J}{\partial z^{(2)}}$$

Equations for forward and backward propagation in the example network illustrated above

Conclusion

The proposed activation function, being **more accurate** than ReLU and **faster** than logistic and softplus (for C=0.1 and C=0.01), has some advantage over every function against which it was tested, making it potentially useful for neural networks.

Discussion and Further Research

I propose to name this function the ***hyperbolic rectifier unit*** (or **HRU**) because its graph is a hyperbola with asymptotes $y = 0$ and $y = x$. In the future, the HRU could be tested with other values of C, within other neural networks for other problems, and against other activation functions, such as the arctangent, hyperbolic tangent, and exponential linear unit.

Bibliography/Acknowledgments

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.net2016/the-softmax-function-and-its-derivative/.
7. Matthew D. Zeiler. ADADELTA: An Adaptive Learning Rate Method. 22 Dec 2012.

I would like to thank Mr. Osei-Mensah for organizing Loomis's participation in CSEF.