DS-GA 1008 - Deep Learning, Spring 2017

Assignment 1

Dhruv Madeka (dm4087), Sean D Rosario (sdr375), and Osvaldo Bulos (obr214)

New York University

1 Backprop

1.1 Nonlinear Activation Functions

$$\frac{\partial E_{\text{out}}}{\partial x_{\text{in}}} = \frac{\partial E_{\text{out}}}{\partial x_{\text{out}}} \frac{\partial x_{\text{out}}}{\partial x_{\text{in}}} \qquad \text{(By the chain rule)}$$

sigmoid :
See, Figure 18 in Appendix A for a graphical illustration.

$$\begin{aligned}
\frac{\partial x_{\text{out}}}{\partial x_{\text{in}}} &= \frac{e^{-x_{\text{in}}}}{(1+e^{-x_{\text{in}}})^2} \\
&= \frac{(1+e^{-x_{\text{in}}})-1}{(1+e^{-x_{\text{in}}})^2} \\
&= \frac{1}{(1+e^{-x_{\text{in}}})} - \frac{1}{(1+e^{-x_{\text{in}}})^2} \\
&= \frac{1}{1+e^{-x_{\text{in}}}}\left(1 - \frac{1}{1+e^{-x_{\text{in}}}}\right) \\
&= \sigma(x_{\text{in}})(1-\sigma(x_{\text{in}}))
\end{aligned}$$

$$\implies \frac{\partial E_{\text{out}}}{\partial x_{\text{in}}} = \frac{\partial x_{\text{out}}}{\partial x_{\text{in}}} \frac{e^{-x_{\text{in}}}}{(1+e^{-x_{\text{in}}})^2} \tag{1}$$

$$\implies \frac{\partial E_{\text{out}}}{\partial x_{\text{in}}} = \frac{\partial x_{\text{out}}}{\partial x_{\text{in}}} \sigma(x_{\text{in}})(1-\sigma(x_{\text{in}})) \tag{2}$$

tanh :
See, Figure 19 in Appendix A for a graphical illustration.

$$\begin{aligned}
x_{\text{out}} &= 1 - \frac{2}{e^{2x_{\text{in}}}+1} \\
\implies \frac{\partial x_{\text{out}}}{\partial x_{\text{in}}} &= \frac{4e^{2x_{\text{in}}}}{(e^{2x_{\text{in}}}+1)^2} \\
\implies \frac{\partial E_{\text{out}}}{\partial x_{\text{in}}} &= \frac{\partial x_{\text{out}}}{\partial x_{\text{in}}} \frac{4e^{2x_{\text{in}}}}{(e^{2x_{\text{in}}}+1)^2}
\end{aligned} \tag{3}$$

ReLU :
See, Figure 20 in Appendix A for a graphical illustration.

$$\begin{aligned}
\frac{\partial x_{\text{out}}}{\partial x_{\text{in}}} &= \mathbb{1}_{x_{\text{in}} \geq 0} \\
\implies \frac{\partial E_{\text{out}}}{\partial x_{\text{in}}} &= \frac{\partial x_{\text{out}}}{\partial x_{\text{in}}} \mathbb{1}_{x_{\text{in}} \geq 0}
\end{aligned} \tag{4}$$

1.2 Softmax

Here, we need to consider two cases. When $i = j$ and when $i \neq j$.

When $i = j$:

$$\begin{aligned}\frac{\partial (X_{\text{out}})_i}{\partial (X_{\text{in}})_i} &= \frac{-\beta e^{-\beta (X_{\text{in}})_i} (\sum_k e^{-\beta (X_{\text{in}})_k}) + \beta e^{-2\beta (X_{\text{in}})_i}}{(\sum_k e^{-\beta (X_{\text{in}})_k})^2} \\ &= \frac{\beta e^{-\beta (X_{\text{in}})_i} (1 - \sum_{k \neq i} e^{-\beta (X_{\text{in}})_k})}{(\sum_k e^{-\beta (X_{\text{in}})_k})^2}\end{aligned}$$

When $i \neq j$:

$$\begin{aligned}\frac{\partial (X_{\text{out}})_j}{\partial (X_{\text{in}})_i} &= \frac{-\beta e^{-\beta (X_{\text{in}})_i} e^{-\beta (X_{\text{in}})_j}}{(\sum_k e^{-\beta (X_{\text{in}})_k})^2} \\ \implies \frac{\partial (X_{\text{out}})_j}{\partial (X_{\text{in}})_i} &= \frac{-\beta e^{-\beta [(X_{\text{in}})_i + (X_{\text{in}})_j]}}{(\sum_k e^{-\beta (X_{\text{in}})_k})^2}\end{aligned}$$

2 Techniques

2.1 Optimization

Momentum The momentum method [4] for Gradient Descent takes the following form. Given a function $f(\theta)$ to be minimized, each step in the iteration is given by:

$$v_{t+1} = \mu v_t - \varepsilon \nabla f(\theta_t) \tag{5}$$
$$\theta_{t+1} = \theta_t + v_{t+1} \tag{6}$$

Here, ε is the learning rate, $\mu \in [0, 1]$ is the momentum coefficient and $\nabla f(\theta_t)$ is the gradient of f at the value of the parameters θ_t.

The intuition behind why this method is called momentum can be seen by recursively substituting v_t in the above equation. Doing this, we get:

$$\begin{aligned}\theta_{t+1} &= \theta_t + \mu v_t - \varepsilon \nabla f(\theta_t) \\ &= \theta_t + \mu(\mu v_{t-1} - \varepsilon \nabla f(\theta_{t-1})) - \varepsilon \nabla f(\theta_t) \\ \implies \theta_{t+1} &= \theta_t + \mu \left(\mu^t v_0 - \sum_{i=0}^{t-1} \mu^{t-1-i} \varepsilon \nabla f(\theta_i) \right) - \varepsilon \nabla f(\theta_t) \\ \implies \theta_{t+1} &= \theta_t + \mu^{t+1} v_0 - \varepsilon \sum_{i=0}^{t} \mu^{t-i} \nabla f(\theta_i)\end{aligned}$$

What this means is that the gradient from the previous steps of the descent have an effect on the update of the parameters at the current step. Essentially, the update for the current step is the gradient at the function value plus (with exponential decay) the gradients of the function at the previous steps. Figure 1 shows the weight decay for $\mu = 0.5$. In order to understand the intuition of momentum, we consider Figure 2 which shows how the update vector is moved for a simple function. For a function such as the one above, the gradient at 0 is zero, even though a local minimum is not obtained there, such a saddle point is overcome by momentum methods because the gradient of the previous iterations pushes the descent forward as shown by the blue arrows.

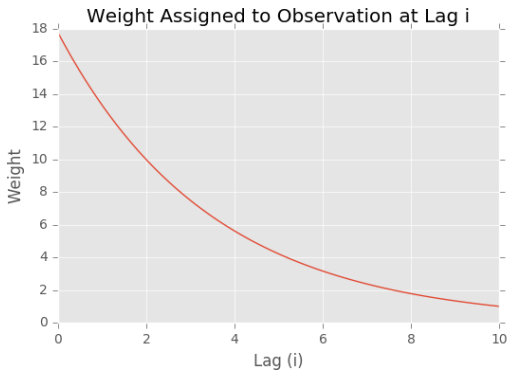

Fig. 1: The weight assigned to each previous iteration of the descent (a lag) decays as an exponential

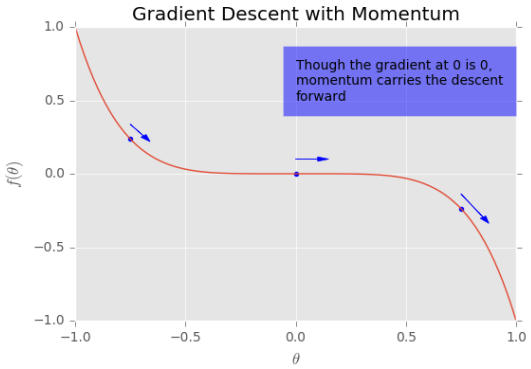

Fig. 2: Momentum carries the descent past the saddle point as shown by the blue arrows

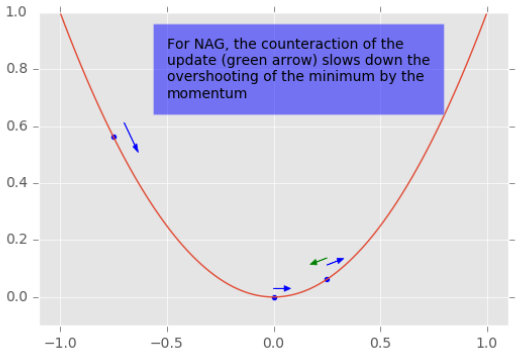

Fig. 3: The gradient correction by the NAG descent (green arrow) provides a better update than classical momentum (blue arrows).

Nesterov's Accelerated Gradient Descent Following [6], the form for the NAG descent is given by:

$$v_{t+1} = \mu v_t - \varepsilon \nabla f(\theta_t + \mu v_t) \tag{7}$$
$$\theta_{t+1} = \theta_t + v_{t+1} \tag{8}$$

A first inspection of this method seems to say that the core idea is not to use the gradient at the current step, but rather to use the gradient of the destination of the step in order to move in the direction of convergence. This means that if the gradient at our destination is of the opposite sign as our gradient at our current position, NAG will nullify some effects of the momentum. The benign looking difference in the equations allows NAG to change v quicker than the classical momentum (CM).

If we consider an update which moves the objective f in an undesirable direction, for NAG the gradient correction to the velocity v_t is computed at position $\theta + \mu v_t$ and if μv_t is indeed a poor update,then $\nabla f(\theta + \mu v_t)$ will point θ_t more strongly than $\nabla f(\theta_t)$ does, thus providing a better correction than CM. The effect of this small improvements compounds over steps as the algorithm iterates. Consider Figure 4 and 5, while Momentum (blue arrows) consistently overshoots the minimum, NAG updates in the correct direction (green arrow is the gradient correction provided by NAG) to provide a better update.

A geometric intuition was provided by [6], which can be seen in Figure 5. Essentially, Classical Momentum uses the gradient at the current step along with the momentum factor to update the

parameter, while NAG performs a partial update through the gradient at $\theta_t + \mu v_t$ to provide a correction which is as yet unknown to CM.

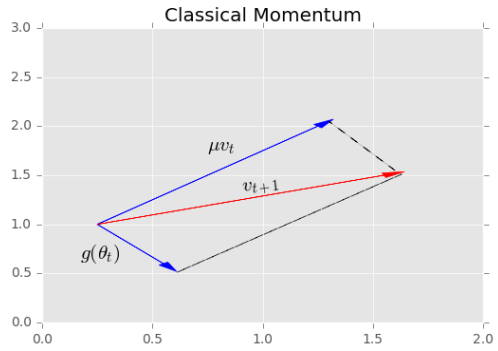

Fig. 4: Classical momentum uses the previous gradients as well as the current gradient value to compute the parameter update.

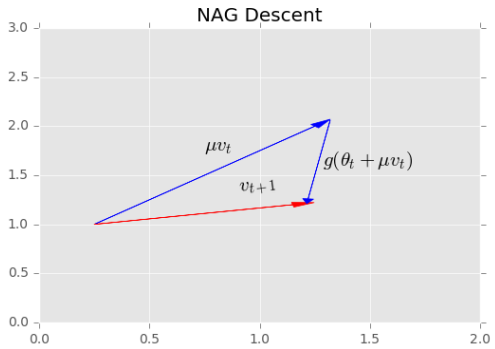

Fig. 5: NAG utilizes the gradient of the destination to compute the parameter update.

2.2 Reducing Overfitting

1 Dropout as an Ensembling Method In order to understand Dropout as an Ensembling method, it helps to remember that an Ensemble method is one in which different models are trained over different datasets (which are sampled from the original with replacement). The reason that this works is that not different models will usually make different errors on the training set. This can be seen as training different models on the same dataset, and with high probability removing some of the input values - so that most datasets have different examples, but can share some of the input values. Recall that to learn with bagging, we define k different models, construct k different datasets by sampling from the training set with replacement, and then train model i on dataset i.

This is usually impractical when training a model as large as a neural net. However, a neural network, which are based on a series of affine transformations and non-linearities, we can effectively remove a unit from the network by multiplying its output by zero.

Dropout aims to approximate k-different models by a different binary mask to apply to all of the input and hidden units in the network for each mini batch. Each time we load an example into a mini-batch, we randomly sample a different binary mask to apply to all of the input and hidden units in the network. This can be seen as sampling and training a thinned network for each training case. Thus, as shown in Figure 6, dropout amounts to sampling over a thinned network (since equivalently a neural net with n units can be seen as a collection of 2^n thinned networks). The mask for each unit is sampled independently from all others.

Specifically, for a mask probability η and a set of parameters θ, dropout consists of minimizing the expected loss L as $\mathop{\mathbb{E}}_{\eta}[L(\eta, \theta)]$

Yet, dropout has some differences from ensemble methods. In the case of ensemble methods, each model is independent of the other and is trained to convergence on it's training set. For dropout, a tiny subset of the 2^n models are each trained for a single step and parameter sharing causes the remaining sub-networks to arrive at good settings.

2 Weight Scaling for Dropout At each pass of the training data, for dropout, each node is retained with probability p and dropped. Since dropping out a network is equivalent to setting its weight to 0, we can say that for every weight at layer l and pass i w_i^l the expectation is $\mathbb{E}[W] = pW_i^l + (1-p) * 0$. What this means is that for every data set, for some passes, we increase the weight by retaining the node, and others we drop it. As a result, at test time (since we want

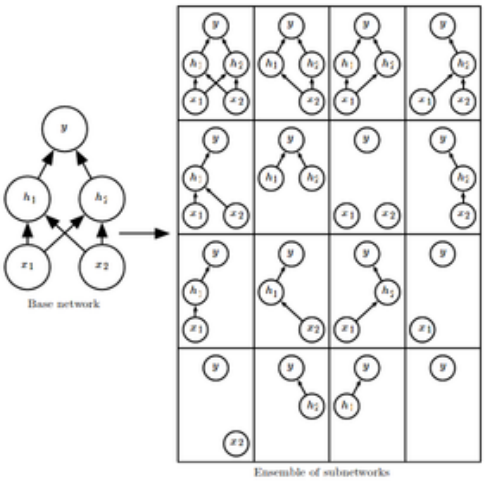

Fig. 6: Following [2], we consider a base network with 2 input and 2 hidden layers. We show the 16 possible networks that might be considered by randomly dropping out an input or hidden layer. The problem of having networks that don't connect the input to the output layer is minimized for broader networks.

approximately $\mathbb{E}[W]$) we can scale by pW_i^l or if we are doing this during training, it is equivalent to diving the weight by the probability of retention $\frac{1}{p}W_i^l$.

Either way, the goal is to make sure that the expected total input to a unit at test time is roughly the same as the expected total input to that unit at train time, even though half the units at train time are missing on average.

3 Data Augmentation for MNIST Two kinds of transformations can be applied to the MNIST dataset to deal with the scarcity problem.
The first method would be simple distortions (translation, rotation and skewing) by applying affine displacement fields. This method is done by computing a new target location for every pixel with respect to its original location. As an example: having Δx(x,y)=1, and Δy(x,y)=0, the new target location of the pixel will be a shift of 1 pixel to the right and none in the vertical axis. By having the displacement field Δx(x,y)= αx, and Δy(x,y)= αy, the image would be scaled by a factor of α. However, since α could be a non-integer value, an interpolation would be required.
The second method is the use of elastic deformations. The idea behind this method is to simulate the uncontrolled oscillations of the hand muscles dampened by inertia. These deformations are created by generating a random displacement field: Δx(x,y) = rand(-1,+1) and Δx(x,y)=rand(-1,+1), where rand(-1,+1) is a random number between -1 and +1 with a uniform distribution. The fields Δx and Δy are then convolved with a Gaussian of standard deviation σ given in pixels. The displacement fields are then multiplied by a scaling factor α that controls the intensity of the deformation.

2.3 Initialization

He et. al initialization vs Xavier initialization The method proposed by He et al [3] deals with the difficulty of deep models (more than 8 convolutional layers) to converge when the weights are initialized randomly using a Gaussian distribution with a fixed standard deviation (usually 0.01). This problem in convergence is due that if the values of the activation are too small, the variance will drop in each layer, while if the activations become bigger and bigger, the values will saturate and become meaningless, with gradients approaching to zero. The Xavier method, proposed by Glorot and Bengio [1], uses a scaled uniform distribution for initialization, which takes the assumption that the activations inside the model are linear. This method is totally invalid when ReLUs or PReLUs are used. The He et al method deals with the rectifier nonlinearities.

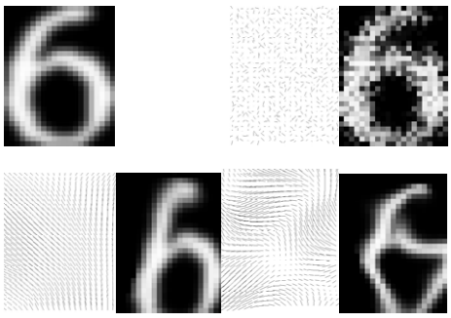

Fig. 7: Examples of Elastic Deformations using different σ: (Top Right) Pure random field (σ=0.01). (Bottom Left) Smoothed random field resembling the properties of the hand (σ=8). (Bottom Right) Smoothed random field with too much variability (σ=4)

To analyze more deeply the difference between these two methods, a good start is the variance formula for the responses in each layer.

$$Var(Y) = Var(W_1X_1 + W_2X_2 + ... + W_nX_n) = nVar(W_i)Var(X_i) \tag{9}$$

Here we can see that the variance of the output is the variance of the input just scaled by $nVar(W_i)$. In order to keep the variance of the input the same as the one of the output, then $nVar(W_i)$ should be 1, giving us:

$$Var(W_i) = \frac{1}{n} = \frac{1}{n_{in}} \tag{10}$$

Following the same steps for the backpropagation signal, we need

$$Var(W_i) = \frac{1}{n} = \frac{1}{n_{out}} \tag{11}$$

to keep the variance of the input and output gradient the same. These constraints can be satisfied if $n_{in} = n_{out}$, so Glorot and Bengio, in their Xavier initialization take the average of the two:

$$Var(W_i) = \frac{2}{n_{in} + n_{out}} \tag{12}$$

An important assumption here is the "linear neuron" bit, that Glorot and Bengio specifies that just after initialization, the parts of the nonlinearities (tanh and sigm) that are explored are the bits close to zero and where the gradient is close to 1.

However this does not hold with the rectifying nonlinearities. He et al. re-did the derivations for ReLU and PReLU and they found that the conditions were the same just up to a factor of 2. A rectifying linear unit is zero for half of its input, so the size of the weight variance must be doubled to keep the variance constant.

They got the equation:

$$\frac{1}{2}(1 + a^2)n_{in}Var(W) = 1 \tag{13}$$

where a is the initialized value of the coefficients. When $a = 0$, it becomes the ReLU case;

$$\frac{1}{2}n_{in}Var(W) = 1 \tag{14}$$

and if $a = 1$, it becomes the linear case [Eq. 7].

Pretraining To prevent the learning from stalling, the VGG team [5] pre-train a shallow network , shallow enough to be trained with random initialization. The weights learned were used to initialize a much deeper model. More specifically, the first four convolutional layers and the last three fully connected layers were initialized with the weights of the shallow pre-trained network. All the weights of intermediate layers that were not initialized with pretrained weights were initialized randomly, sampled from a normal distribution with the zero mean and 10^2 variance.

Unsupervised feature extraction Coate et. al discuss methods to extract features from an image using several simple unsupervised method, which are off-the-shelf feature learning algorithms. The objective is to learn a function f that can learn the transformation of an input $x \in \mathbb{R}^N$ to a vector representation $y \in \mathbb{R}^K$, where N is a vector of pixel intensity values from a w-by-w patch of an image, and K is the number of features to extract. Some of the methods are as follows:

Sparse auto-encoder : An auto-encoder with K hidden nodes is trained using backprop to learn weights W and biases b for the feature mapping $f(x) = g(Wx + b)$, where $g(z)$ is the logistic sigmoid function, for a vector z.

Sparse restricted Boltzmann machine : A Sparse RBM with K binary hidden variables is trained to learn the weights W and biases b using the contrastive divergence approximation, for the same feature mapping as an auto-encoder

K-means clustering :K centroids are learned for two choices of feature mapping f The standard 1-of-K coding scheme is represented as $f_k(x) = \mathbb{1}(k = argmin_j \|c^j - x\|_2^2)$. Since this is maximally sparse, another non-linear feature mapping can be used for softer encoding while maintaining some sparsity: $f_k(x) = max\{0, \mu(z) - z_k\}$, where

Gausian mixture models : GMMs can represent the input data as K Gaussian distributions. GMMs is trained for one iteration using the Expectation-Maximation (EM) algorithm. The feature mapping f maps it into posterior probabilities.

Using the function f, a representation y is then computed for each w-by-w subpatch of the input image. The dimension can be reduced by pooling. The paper specifies pooling by splitting the $y^{(ij)}$'s into four equally sized quadrants, and summing the values in each quadrant. This reduces the dimension from (n-w+1)-by-(n-w+1)-by-K to $4K$, where K is the number of channels. SVM classification is then applied on the 4K dimensional feature vectors.

3 PyTorch (MNIST Handwrittern Digit Recognition)

3.1 Exploring the MNIST data

The dataset is the MNIST hand recognition dataset, available at http://yann.lecun.com/exdb/mnist/. For this assignment, we had access to 3000 labeled examples and 47000 unlabeled examples.

By doing a t-sne plot, we observe that the different classes, for the most part, are linearly separable. T-sne (t-Distributed Stochastic Neighbor Embedding) is a dimension reduction method proposed by Laurens van der Maaten and Geoffrey Hinton [7], that is commonly used to visualize high dimensional data, on a 2D dimensional surface, where the distance between points in the 2 dimensional space is representative of the distance in the higher dimensional space. The locations of the points y_i in the map are determined by minimizing the (non-symmetric) KullbackLeibler divergence of the distribution Q from the distribution P, that is:

$$KL(P||Q) = \sum_{i \neq j} p_{ij} \log \frac{p_{ij}}{q_{ij}}$$

where Given a set of N high-dimensional objects x_i.... x_N , $p_{ij} = \frac{p_{j|i} + p_{i|j}}{2N}$ and

$$p_{j|i} = \frac{\exp(-\|\mathbf{x}_i - \mathbf{x}_j\|^2/2\sigma_i^2)}{\sum_{k \neq i} \exp(-\|\mathbf{x}_i - \mathbf{x}_k\|^2/2\sigma_i^2)},$$

where σ_i is the bandwidth of the Gaussian kernels defines the perplexity of the conditional distribution, and

$$q_{ij} = \frac{(1 + \|\mathbf{y}_i - \mathbf{y}_j\|^2)^{-1}}{\sum_{k \neq m} (1 + \|\mathbf{y}_k - \mathbf{y}_m\|^2)^{-1}}$$

where n a d-dimensional map y_i.... y_N is a d-dimensional mapping that is learnt to reflect similarities $p_i j$

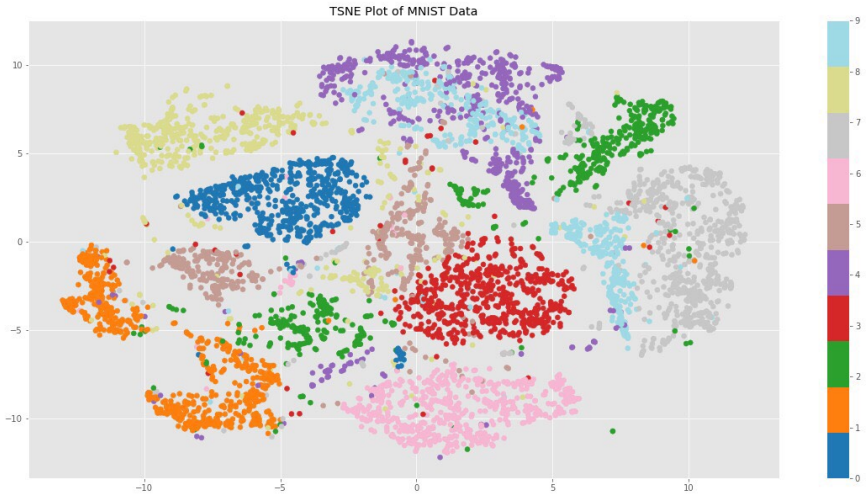

Fig. 8: T-sne plot of the 3000 labeled data points

3.2 Baseline model

The default model is a simple convolutional neural network, that trains on the dataset of 3000 labeled data points. The network has the following architecture:

- Convolution with a 5 x 5 kernel
- Max Pool by 2
- Rectified linear Unit activation
- Convolution with a 5 x 5 kernel
- Dropout
- Max Pool by 2
- Rectified linear Unit activation
- Fully connected layer with dropout

We tried variations of the default model, to see what factors would improve performance. We played with the network in the following ways:

Adding another layer: We tried adding another layer to the network. The performance of the 3-layer model after 200 epochs did not go beyond 95%. Hence we decided not to add more layers, and stick with a 2 layer network.

Number of epochs: With the default configuration of the network, we trained the model for 200 epochs and recorded its accuracy on the validation set, after each training epoch. The network is able to reach close to 97% accuracy, given enough epochs, after which it seems to plateau.

The VGG model The next model used for this task was the VGG with the following architectures in Table 2. The VGG configurations are very large networks. We weren't able to get good results with the VGG network

VGG represented a challenge for the amount of layer involved.

3.3 Data Augmentation

To create a larger training set, a series of transformations were applied to the labeled dataset. Table 3 show the selected transformations with their parameters range.

Model number	No. of epochs	Learning rate	Momentum	Validation Accuracy
1	100	0.1	0.5	96.37%
2	100	0.05	0.5	96.76%
3	100	0.01	0.5	96.18%
4	100	0.1	0.95	10.0%
5	100	0.05	0.95	10.0%
6	100	0.005	0.5	95.53%
7	100	0.005	0.95	97.08%

Table 1: Table of Accuracies for the different base models

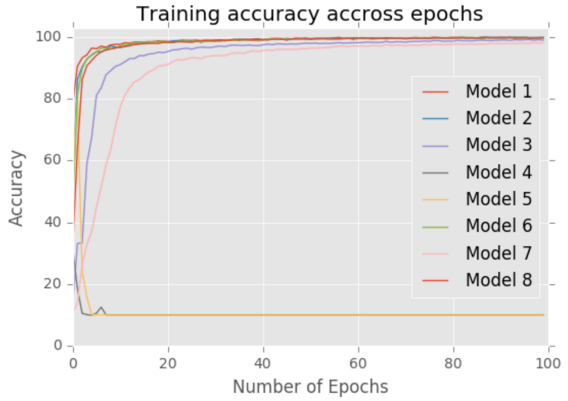

Fig. 9: Accuracy curves on the training set

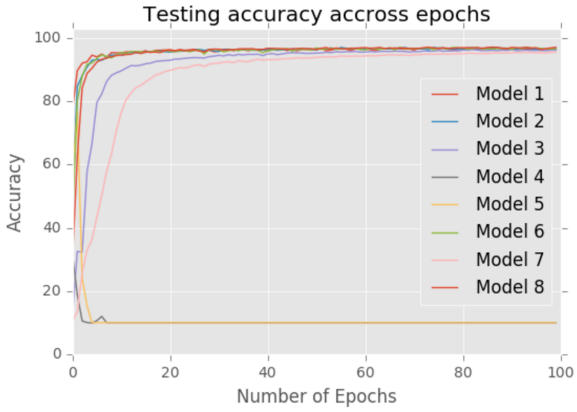

Fig. 10: Accuracy curves on the validation set

VGG	?? Architecture
VGG11:	64, 'MaxPool' (M), 128, 'M', 256, 256, 'M', 512, 512, 'M', 512, 512, 'M'
VGG13	64, 64, 'M', 128, 128, 'M', 256, 256, 'M', 512, 512, 'M', 512, 512, 'M'
VGG16	64, 64, 'M', 128, 128, 'M', 256, 256, 256, 'M', 512, 512, 512, 'M', 512, 512, 512, 'M'
VGG19	64, 64, 'M', 128, 128, 'M', 256, 256, 256, 256, 'M', 512, 512, 512, 512, 'M', 512, 512, 512, 512, 'M'

Table 2: VGG Architectures

A certain number of parameter variations were applied to each transformation to create several images. For example: if the parameter variations number was set to 2, two different rotated images were created; if it was set to 10, then ten different rotated images were created. This does not apply for Random Gaussian Noise and Dilation transformations. Table 4 shows the different number of

ConvNet Configuration					
A	A-LRN	B	C	D	E
11 weight layers	11 weight layers	13 weight layers	16 weight layers	16 weight layers	19 weight layers
input (224 × 224 RGB image)					
conv3-64	conv3-64 **LRN**	conv3-64 **conv3-64**	conv3-64 conv3-64	conv3-64 conv3-64	conv3-64 conv3-64
maxpool					
conv3-128	conv3-128	conv3-128 **conv3-128**	conv3-128 conv3-128	conv3-128 conv3-128	conv3-128 conv3-128
maxpool					
conv3-256 conv3-256	conv3-256 conv3-256	conv3-256 conv3-256	conv3-256 conv3-256 **conv1-256**	conv3-256 conv3-256 **conv3-256**	conv3-256 conv3-256 conv3-256 **conv3-256**
maxpool					
conv3-512 conv3-512	conv3-512 conv3-512	conv3-512 conv3-512	conv3-512 conv3-512 **conv1-512**	conv3-512 conv3-512 **conv3-512**	conv3-512 conv3-512 conv3-512 **conv3-512**
maxpool					
conv3-512 conv3-512	conv3-512 conv3-512	conv3-512 conv3-512	conv3-512 conv3-512 **conv1-512**	conv3-512 conv3-512 **conv3-512**	conv3-512 conv3-512 conv3-512 **conv3-512**
maxpool					
FC-4096					
FC-4096					
FC-1000					
soft-max					

Fig. 11: The Convolutional network configurations used in the VGG model

Transformation	Parameters	Description
Random Gaussian Noise	Default	Adds Noise with a Gaussian Dist.
Dilation	Default	Makes the lines of the number bolder
Elastic Transformation	$\alpha = 1.8$	Simulates oscillations
Rotation	[-10, 10]	Rotates the image
X Padding	[-5, 5]	Pads the image in the X axis
Y Padding	[-5, 5]	Pads the image in the Y axis
X Scaling	[0.8, 1.3]	Scales the image in the X axis
Y Scaling	[0.8, 1.3]	Scales the image in the Y axis
Padding + Scale	[-3, 3], [0.9, 1.2]	Combination of Padding and Scale

Table 3: Table of Transformations

No. of Variations	Size of Dataset	Accuracy
2	51,000	90%
3	72,000	83%
5	114,000	90%
7	156,000	89%
10	219,000	94%

Table 4: Table of Data Augmentation

parameter variation with the resultant dataset and the accuracy obtained by running the baseline model.

After running the first series of Data Augmentations, the dataset with 219,000 images obtained the best accuracy with 94%. A considerable decrease with respect to the baseline model. It is reasonable to think that some of these transformations are introducing noise to the model, making the accuracy to drop. A further analysis of each of the transformations would be required to identify the ones that helps the model. A tunning of the parameters is another aspect to consider in order

to create significant images.
Figure 12 shows the different transformations applied to the dataset.

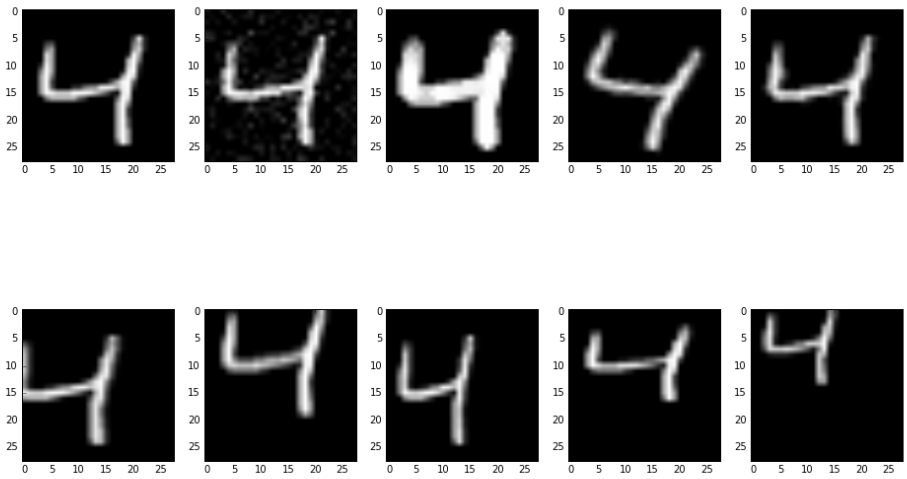

Fig. 12: Transformations applied to the dataset.
From Top Row: Original. Gaussian Noise. Dilation. Rotation. Elastic Transformations.
From Bottom Row: X Padding. Y Padding. X Scaling. Y Scaling. Padding + Scale

3.4 Final Model

Model architecture The final model architecture that we adopted was a modification of the initial base model. The tools that enabled us to obtain a higher test accuracy were primarily additional convolutional layers and better activation functions as well as a much wider network. The structure of the network was as follows:
Structure of Final Model:

- Convolution Layer
- Max-Pooling Layer
- UpSampling Layer
- Convolution Layer with Dropout
- Max-Pooling Layer
- UpSampling Layer
- Convolution Layer
- Max-Pooling Layer
- UpSampling Layer
- Linear Layer with 500 Output Neurons
- Linear Layer with 50 Output Neurons
- Linear Layer with 10 Output Neurons
- Log Softmax Activation

All layers use a special type of Activation function called the Leaky Recitified Linear Unit (Leaky-ReLU).

Leaky-ReLU: Following **??**, we noticed that ReLU activation functions suffer from the is that a large gradient flowing through the ReLU could cause weights to update in such a way that the neuron would not activate for any datapoint again. That is, the ReLU units can irreversibly die during training since they can get knocked off the data manifold. Leaky ReLU's (see Figure 13 below) are an attempt by **??** to fix this "dying ReLU" problem. Instead of being zero when $x < 0$, the functions have the form $f(x) = \mathbb{1}_{x<0}\alpha x + \mathbb{1}_{x\geq 0}x$. The negative slope α can also be a parameter creating the Parametric Rectified Linear Unit (PReLU).

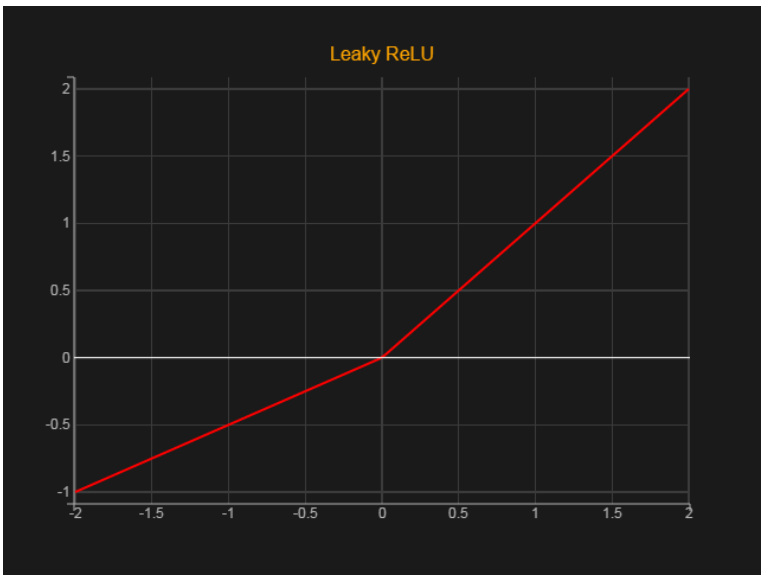

Fig. 13: Leaky ReLU

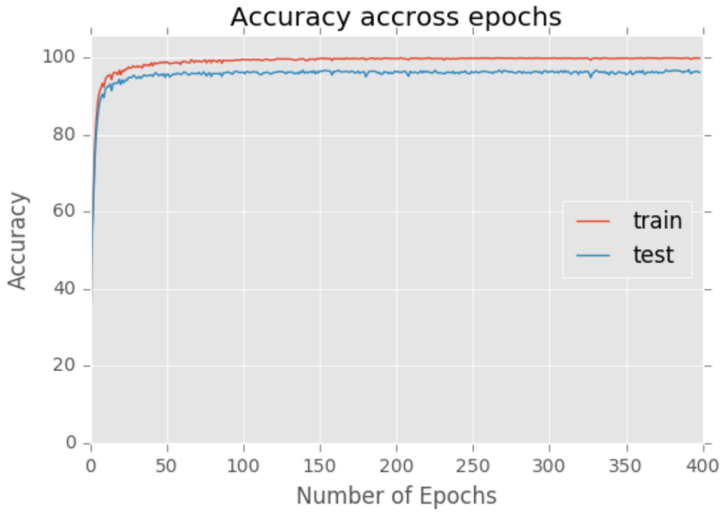

Fig. 14: final curves

UpSampling In order to ensure that the repeated convolutions and pooling operations did not shrink the image too far and allowed for higher level features to be extracted, we used the technique

of upsampling. This can be done in two ways, either by cloning the pixels as seen in Figure 15 below or alternately, by bi-linear interpolation between the pixels.

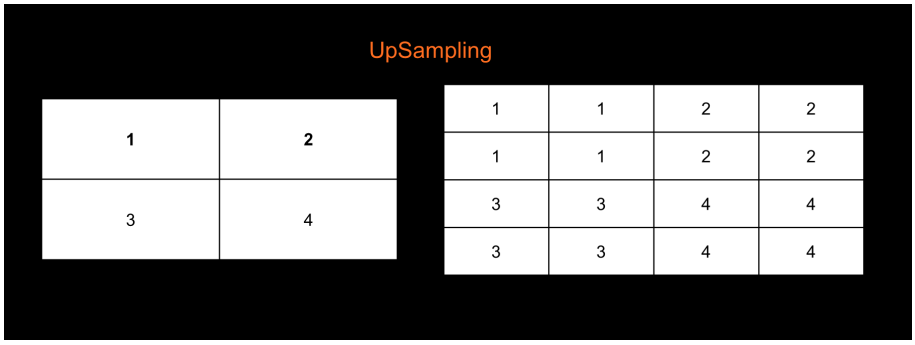

Fig. 15: UpSampling Illustration

3.5 Semi-supervised technique: Pseudo-Labels

To incorporate the unlabeled train set, it was decided to implement Pseudo-Labels.
This semi-supervised method uses target classes in the unlabeled data as if they were the true labels. For each unlabeled batch, the class with maximum predicted probability is set as the pseudo label. The training is done simultaneously using the labeled and unlabeled data.
The combined loss for the the labeled and unlabeled data is

$$L = \frac{1}{n}\sum_{m=1}^{n}\sum_{i=1}^{C} L(y_i^m, f_i^m,) + \alpha(t)\frac{1}{n'}\sum_{m=1}^{n'}\sum_{i=1}^{C} L(y'^m_i, f'^m_i,) \tag{15}$$

where n is the number of the labeled batch, n' the number of the unlabeled batch, f_i^m the output units of m's sample in labeled data, y_i^m its label, f'^m_i the output unlabeled units, y'^m_i the pseudo-label and $\alpha(t)$ is a balancing coefficient.
To set $\alpha(t)$:

$$\alpha(t) = \left\{ \begin{array}{ll} 0, & t \leq T_1 \\ \frac{t-T_1}{T_2-T_1}\alpha_f, & T_1 \leq t \leq T_2 \\ \alpha_f, & T_2 \leq t \end{array} \right\}$$

The model used to train the Pseudo-Labels was the convolutional baseline model, with the parameters in Table 5.

Parameter	Value
Learning Rate	0.05
Momentum	0.5
Optimizer	SGD
Labeled Batch Size	64
Unlabeled Batch Size	700
α_f	3
Epochs	500
T_1	100
T_2	400

Table 5: Table of Pseudo Labels Parameters

This Pseudo Labels model gave an Accuracy of **97%**. No improvement so far with respect with the baseline model.

The loss for Train and Validation are shown in the Figure 16. The Validation Accuracy is shown in Figure 17.

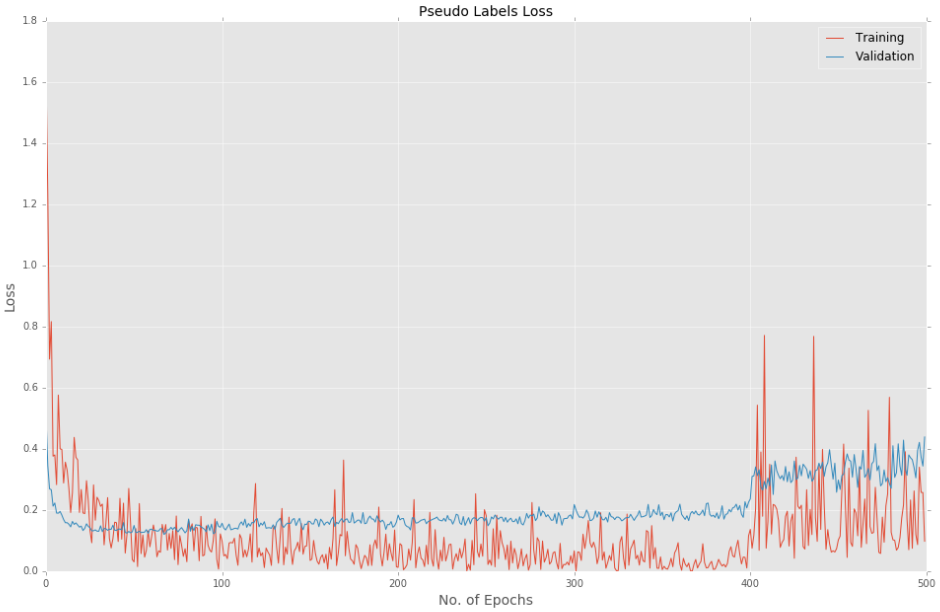

Fig. 16: Pseudo Labels Model - Train and Validation Loss

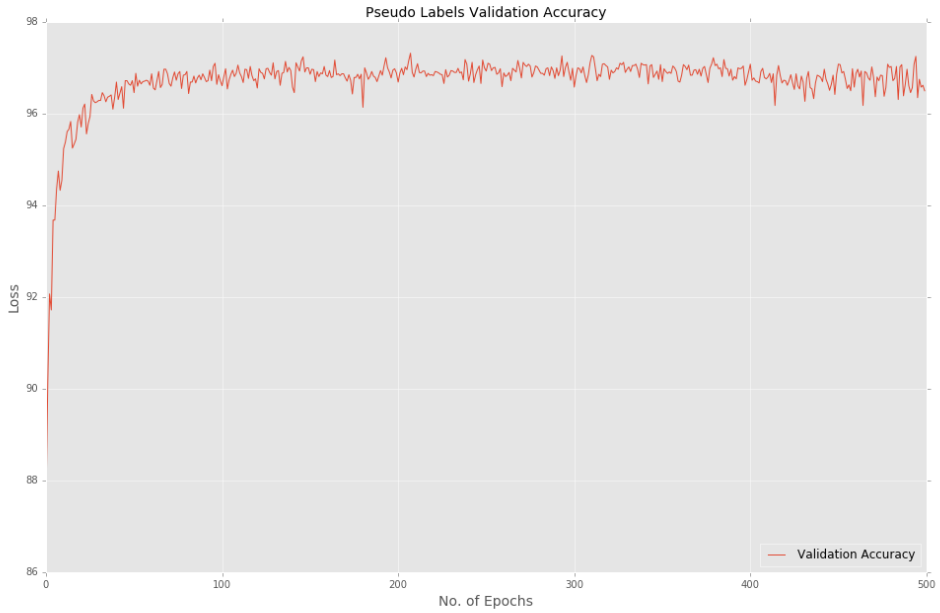

Fig. 17: Pseudo Labels Model - Accuracy

References

1. X. Glorot and Y. Bengio. Understanding the difficulty of training deep feedforward neural networks. 2010.
2. I. Goodfellow, Y. Bengio, and A. Courville. *Deep learning*. MIT Press, 2016.
3. K. He, X. Zhang, S. Ren, and J. Sun. Delving deep into rectifiers: Surpassing human-level performance on imagenet classification. *CoRR*, abs/1502.01852, 2015.
4. B. T. Polyak. Some methods of speeding up the convergence of iteration methods. *USSR Computational Mathematics and Mathematical Physics*, 4(5):1–17, 1964.
5. K. Simonyan and A. Zisserman. Very deep convolutional networks for large-scale image recognition. *CoRR*, abs/1409.1556, 2014.
6. I. Sutskever, J. Martens, G. E. Dahl, and G. E. Hinton. On the importance of initialization and momentum in deep learning.
7. L. van der Maaten and G. E. Hinton. Visualizing high-dimensional data using t-sne. *Journal of Machine Learning Research*, 9:2579–2605, 2008.

A Backprop

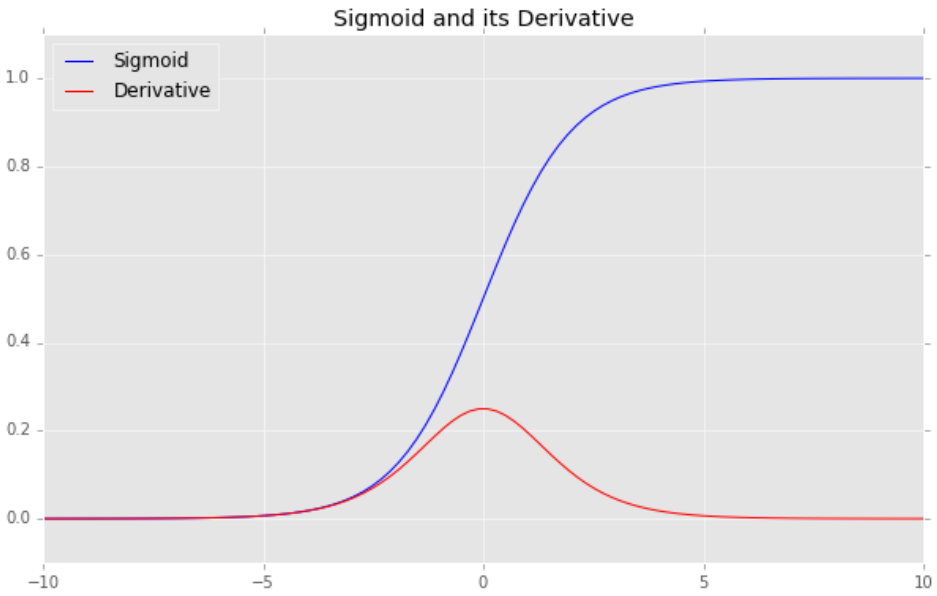

Fig. 18: The Sigmoid function and its derivative resemble the CDF of a Gaussian and it's derivative the PDF of a Gaussian.

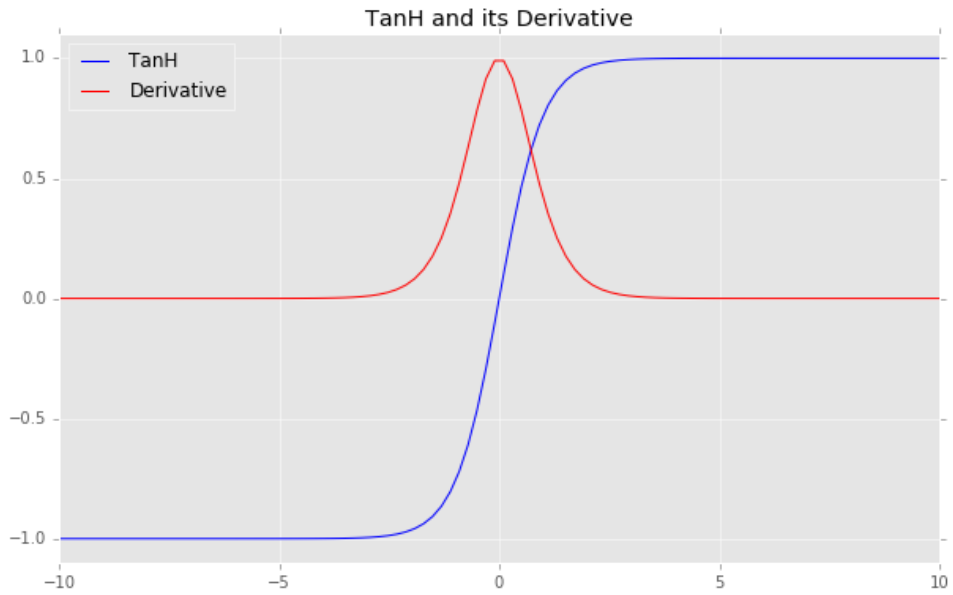

Fig. 19: The difference between the sigmoid and tanh is that the tanh function is just a scaled and shifted sigmoid.

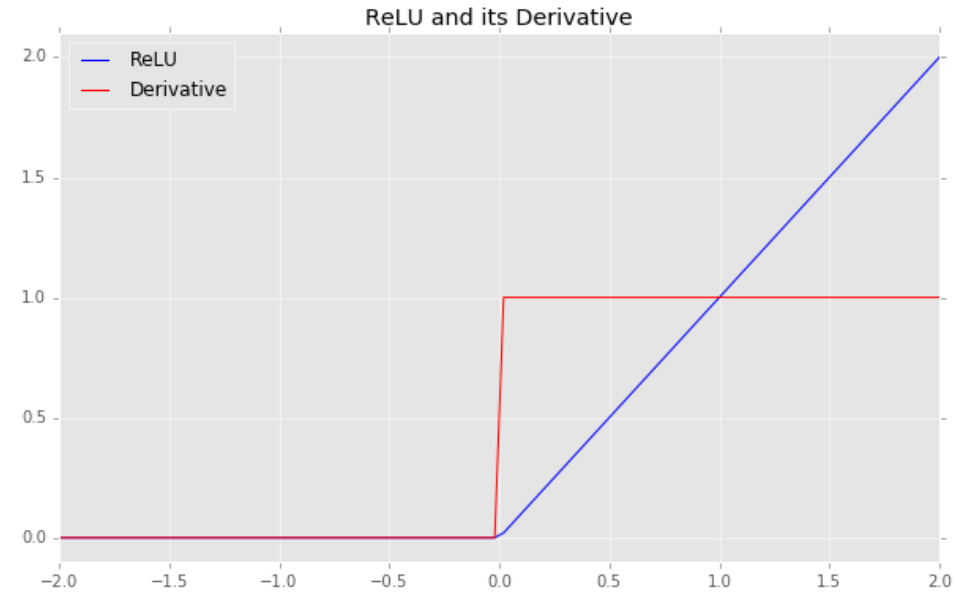

Fig. 20: The ReLU function and its derivative both have a discontinuity at the pivot point.