

# Improving the way neural networks learn

Srikumar Ramalingam

School of Computing

University of Utah

# Reference

Most of the slides are taken from the third chapter of the online book by Michael Nielson:

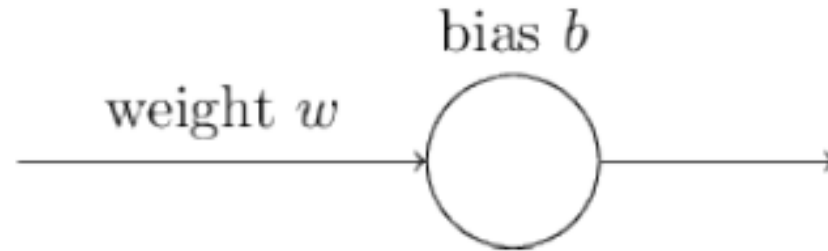
- [neuralnetworksanddeeplearning.com](http://neuralnetworksanddeeplearning.com)

# Contents

- The cross-entropy cost function
- Regularization methods
  - L1 and L2
  - Dropout
  - Artificial expansion of the training data
- Initialization strategies
- Other heuristics

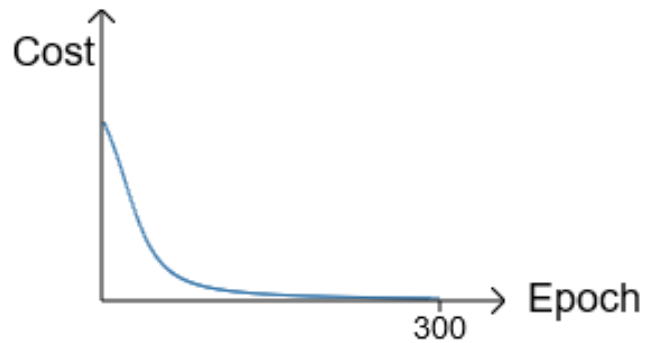
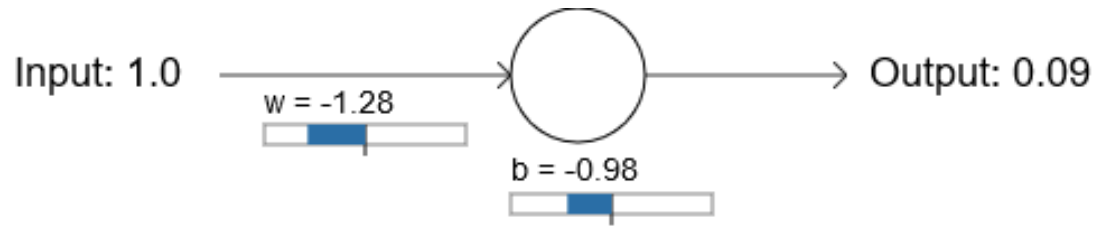
# The cross-entropy cost function

- Simple scenario with a neuron: take an input 1 and return 0.
- Goal: Learn the weight and the bias term:



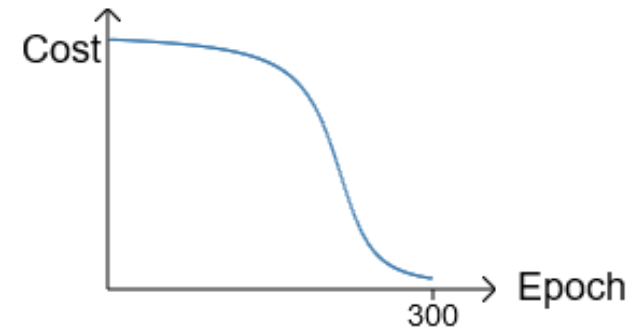
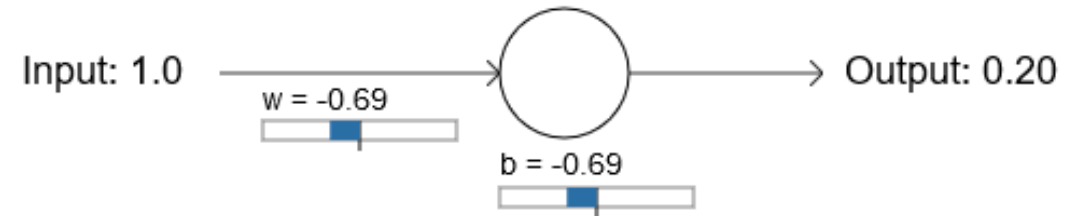
# Two different initialization settings

Initial settings:  $w = 0.6, b = 0.9, \eta = 0.15$ ,  
Takes 300 epochs to converge, final output is still 0.09 and not 0.



Run

Initial settings:  $w = 1, b = 1, \eta = 0.15$ ,  
In the first 150 epochs, the weights don't change much. After 300 epochs, final output is still 0.20 and not 0.



Run

# Learning behavior

- Humans often learn fastest when we're badly wrong about something.
- The artificial neuron has a lot of difficulty learning when it's badly wrong.

# Why is learning so slow? And can we find a way of avoiding this slowdown?

- Learning is slow is same as saying that the “partial derivatives” are small.
- Consider the cost function and partial derivatives:

$$C = \frac{(y-a)^2}{2}, \quad a = \sigma(z), \quad z = wx + b$$

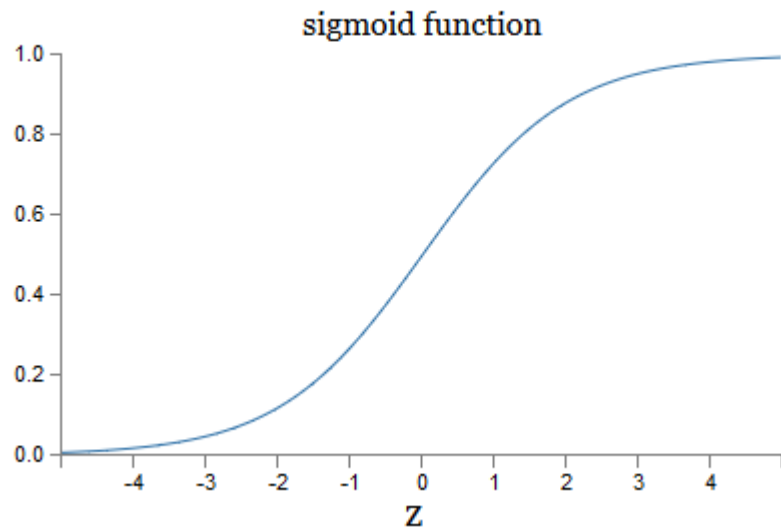
$$\frac{\partial C}{\partial w} = (a - y)\sigma'(z)x, \quad \frac{\partial C}{\partial b} = (a - y)\sigma'(z)$$

Substituting  $x = 1$  and  $y = 0$ , we have:

$$\frac{\partial C}{\partial w} = a\sigma'(z), \quad \frac{\partial C}{\partial b} = a\sigma'(z)$$

# Why is learning so slow? And can we find a way of avoiding this slowdown?

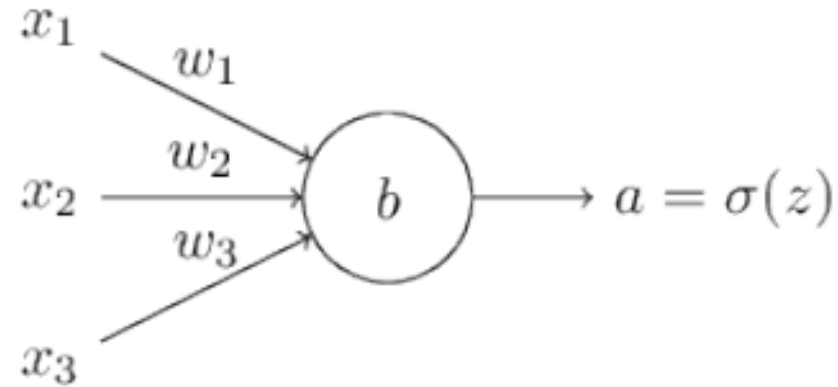
- Learning is slow is same as saying that the “partial derivatives” are small:  $\frac{\partial C}{\partial w} = a\sigma'(z)$ ,  $\frac{\partial C}{\partial b} = a\sigma'(z)$



- When the neuron’s output is close to 1, then the learning becomes very slow.
- For the same reason, learning slowdown also occurs in larger networks and not just the toy scenario.

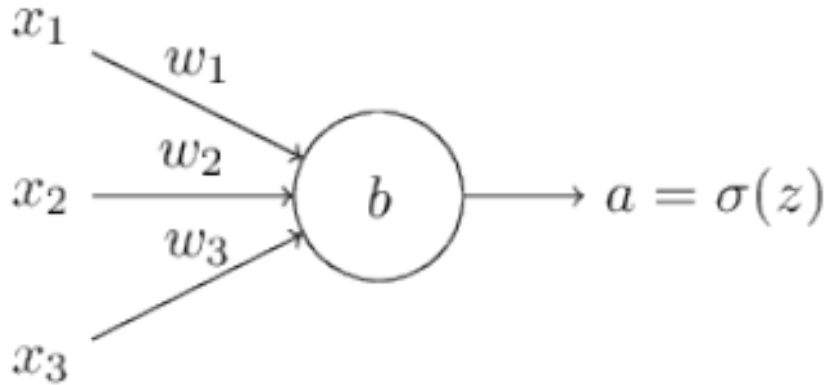


# Cross-entropy loss function



- $C = -\frac{1}{n} \sum_x [y \ln a + (1 - y) \ln(1 - a)]$
- $n$  is the total number of items of training data
- $x$  is the input
- $y$  is the required output and  $a$  is the output from the neuron

# Cross-entropy loss function

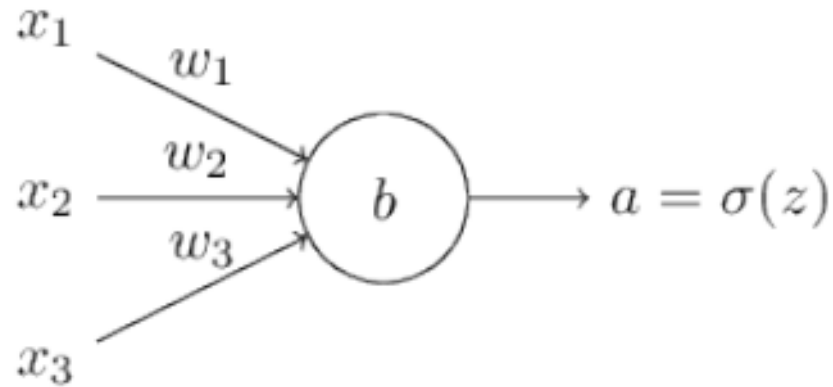


$$C = -\frac{1}{n} \sum_x [y \ln a + (1 - y) \ln(1 - a)]$$

- $n$  is the total number of items of training data
- $x$  is the input
- $y$  is the required output and  $a$  is the output from the neuron

- The cost function is non-negative, i.e.,  $\ln a$  is negative whenever  $0 \leq a \leq 1$ .
- If the neuron's actual output is close to the desired output, then the cost function is close to 0.

# Does cross-entropy avoid learning slowdown?



$$C = -\frac{1}{n} \sum_x [y \ln a + (1 - y) \ln(1 - a)]$$

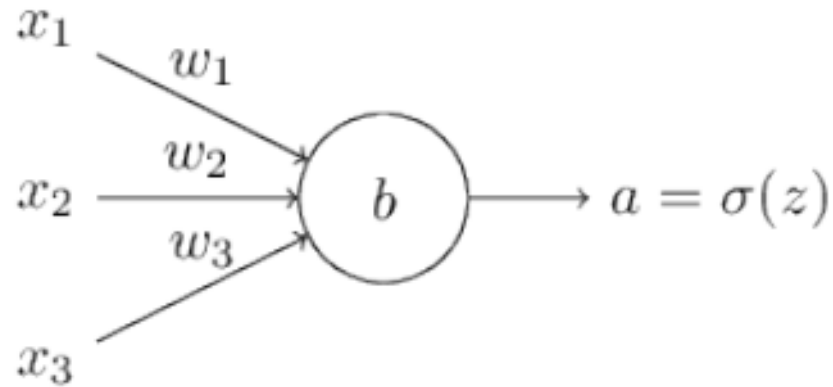
- $n$  is the total number of items of training data
- $x$  is the input
- $y$  is the required output and  $a$  is the output from the neuron

$$\frac{\partial C}{\partial w_j} = -\frac{1}{n} \sum_x \left( \frac{y}{\sigma(z)} - \frac{1 - y}{1 - \sigma(z)} \right) \frac{\partial \sigma(z)}{\partial w_j} = -\frac{1}{n} \sum_x \left( \frac{y}{\sigma(z)} - \frac{1 - y}{1 - \sigma(z)} \right) \sigma'(z) x_j$$

$$\frac{\partial C}{\partial w_j} = \frac{1}{n} \sum_x \frac{\sigma'(z) x_j}{\sigma(z)(1 - \sigma(z))} (\sigma(z) - y) = \frac{1}{n} \sum_x (\sigma(z) - y) x_j$$

Thus we observe that the rate of learning is dependent on  $(\sigma(z) - y)$ , i.e., the error of the output. It avoids the learning slowdown caused by  $\sigma'(z)$  in the quadratic cost function.

# Does cross-entropy avoid learning slowdown?



$$C = -\frac{1}{n} \sum_x [y \ln a + (1 - y) \ln(1 - a)]$$

- $n$  is the total number of items of training data
- $x$  is the input
- $y$  is the required output and  $a$  is the output from the neuron

$$\frac{\partial C}{\partial b} = -\frac{1}{n} \sum_x \left( \frac{y}{\sigma(z)} - \frac{1-y}{1-\sigma(z)} \right) \frac{\partial \sigma(z)}{\partial b} = -\frac{1}{n} \sum_x \left( \frac{y}{\sigma(z)} - \frac{1-y}{1-\sigma(z)} \right) \sigma'(z)$$

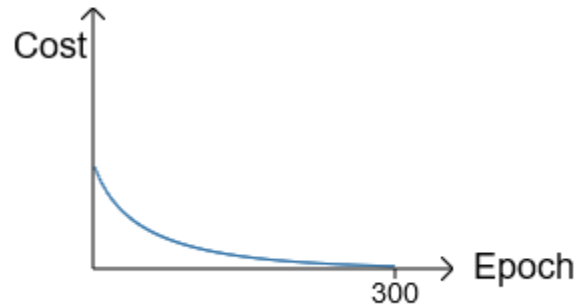
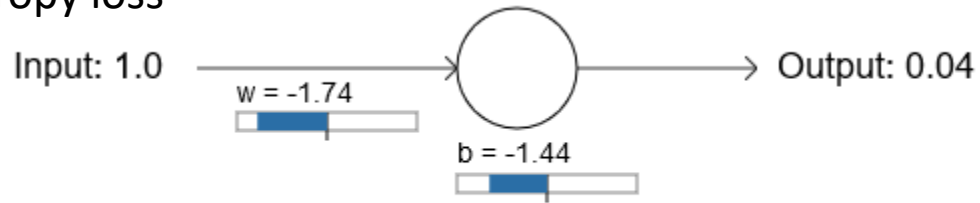
$$\frac{\partial C}{\partial b} = \frac{1}{n} \sum_x \frac{\sigma'(z)}{\sigma(z)(1-\sigma(z))} (\sigma(z) - y) = \frac{1}{n} \sum_x (\sigma(z) - y)$$

Thus we observe that the rate of learning is dependent on  $(\sigma(z) - y)$ , i.e., the error of the output. It avoids the learning slowdown caused by  $\sigma'(z)$  in the quadratic cost function.

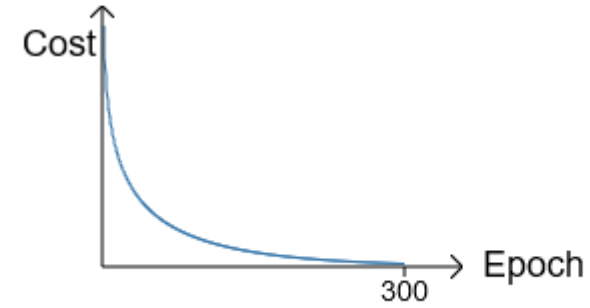
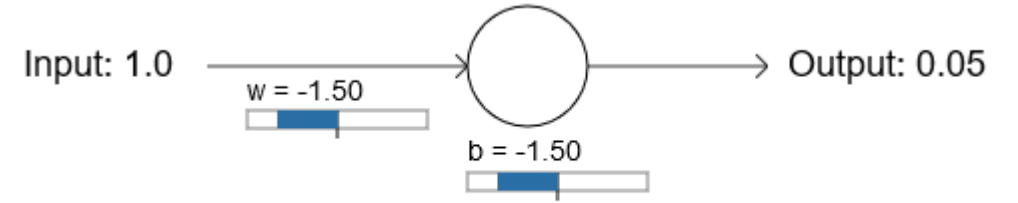
Initial settings:  $w = 0.6, b = 0.9, \eta = 0.005$ ,  
Takes 300 epochs to converge, final output is still 0.04.

Initial settings:  $w = 1, b = 1, \eta = 0.005$ ,  
After 300 epochs, final output is 0.05.

Cross-entropy loss



Run



Run

# Cross-entropy loss for multiple neurons

- $C = -\frac{1}{n} \sum_x \sum_j [y_j \ln a_j^L + (1 - y_j) \ln(1 - a_j^L)]$
- The desired values of the output neurons are given by  $y_1, y_2, \dots$
- The actual output values are given by  $a_1^L, a_2^L, \dots$

# SoftMax layer

- New kind of output layer

$$a_j^L = \frac{e^{z_j^L}}{\sum_k e^{z_k^L}}$$

- The output activations are guaranteed to sum to 1. Could be interpreted as probabilities.
- To prove: Show that a sigmoid output layers don't always sum to 1.

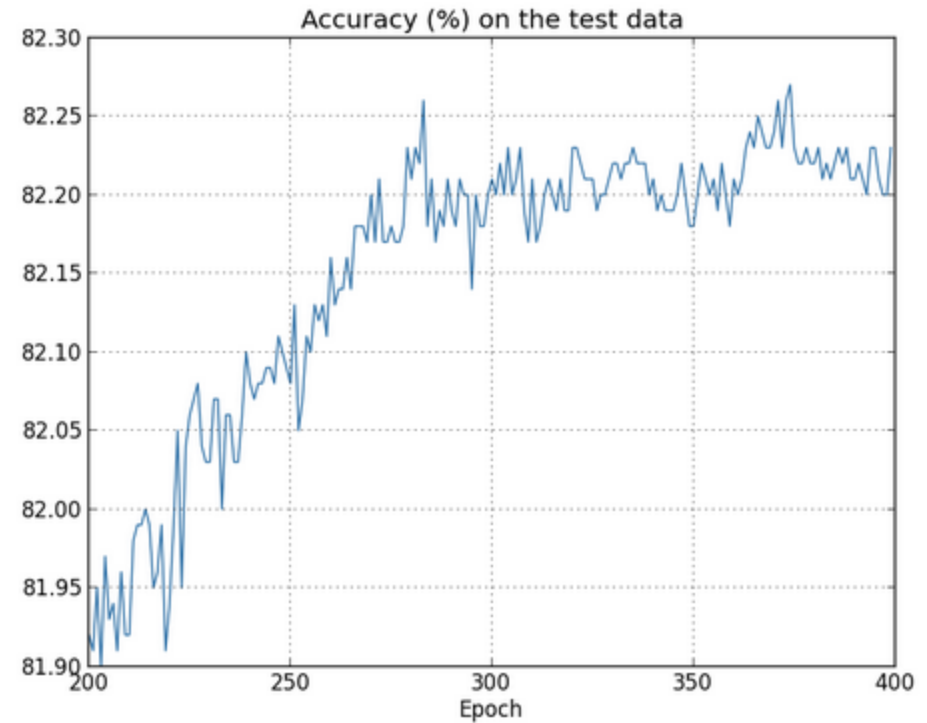
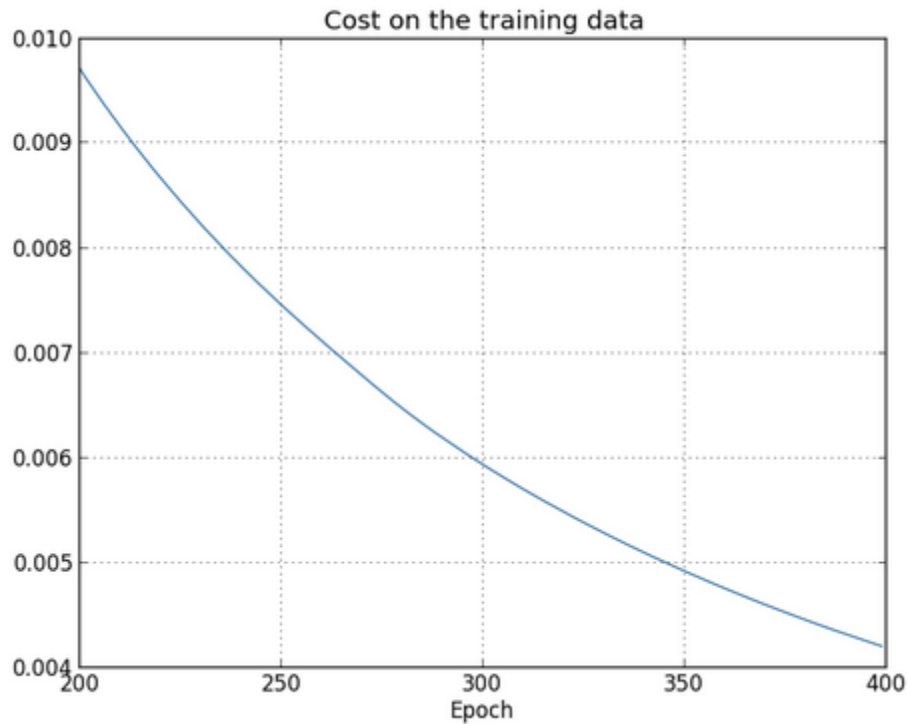
# Monotonicity of SoftMax layer

- Show that  $\frac{\partial a_j^L}{\partial z_k^L}$  is positive if  $j = k$ , and negative otherwise.



# Overfitting and regularization

- Instead of 60000 training images, we use only 1000 training images and check the performance on the test data.



# Use of more data avoids overfitting – not always feasible

On using the full training data, we have the following:



# L2 regularization or weight decay

$$C = -\frac{1}{n} \sum_{x_j} [y_j \ln a_j^L + (1 - y_j) \ln (1 - a_j^L)] + \frac{\lambda}{2n} \sum_w w^2$$

- The first term is just the usual expression for cross-entropy.
- Here  $\lambda$  is the regularization parameter and  $n$  is the size of our training set.
- Note that the regularization does not include the bias terms.
- It is also possible to regularize other cost functions such as the quadratic one:

$$C = \frac{1}{2n} \sum_x |y - a^L|^2 + \frac{\lambda}{2n} \sum_w w^2$$

# L2 regularization or weight decay

$$C = C_0 + \frac{\lambda}{2n} \sum_w w^2$$

- The first term is the original cost function (cross-entropy, quadratic, etc.)

- The partial derivatives are given by:

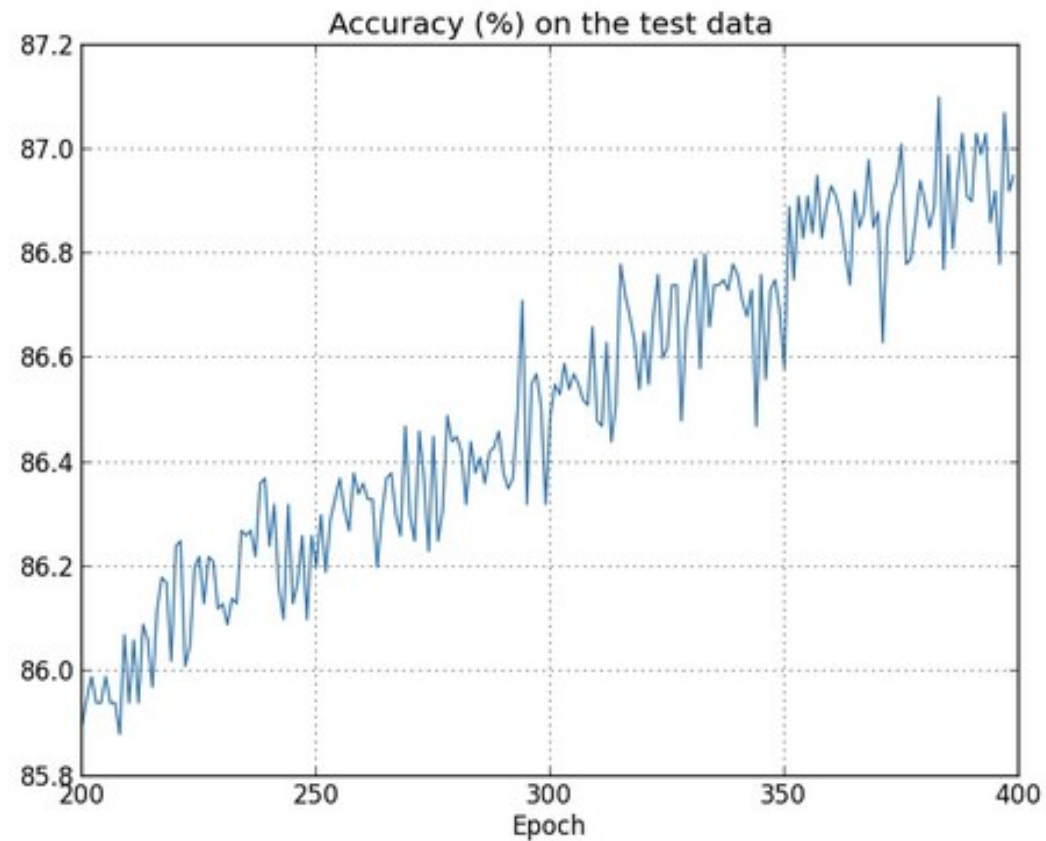
$$\frac{\partial C}{\partial w} = \frac{\partial C_0}{\partial w} + \frac{\lambda}{n} w, \quad \frac{\partial C}{\partial b} = \frac{\partial C_0}{\partial b}$$

- The learning rule for weights and bias terms:

$$b \rightarrow b - \eta \frac{\partial C_0}{\partial b}, \quad w \rightarrow w - \frac{\eta \partial C_0}{\partial w} - \frac{\eta \lambda}{n} w$$

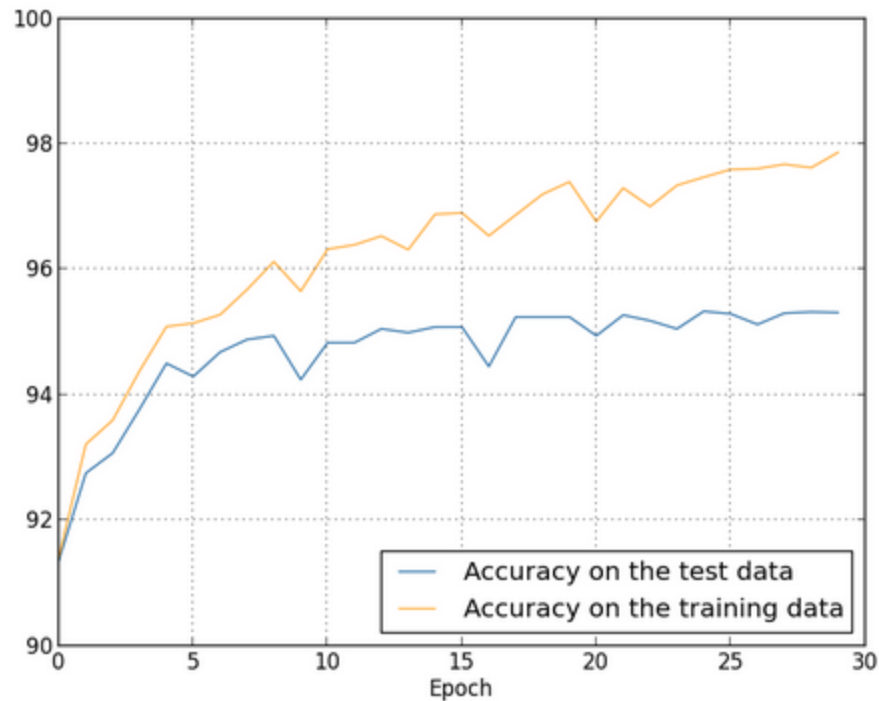
# Benefits of regularization

- Using 1000 training samples, we see some improvement with the regularization:

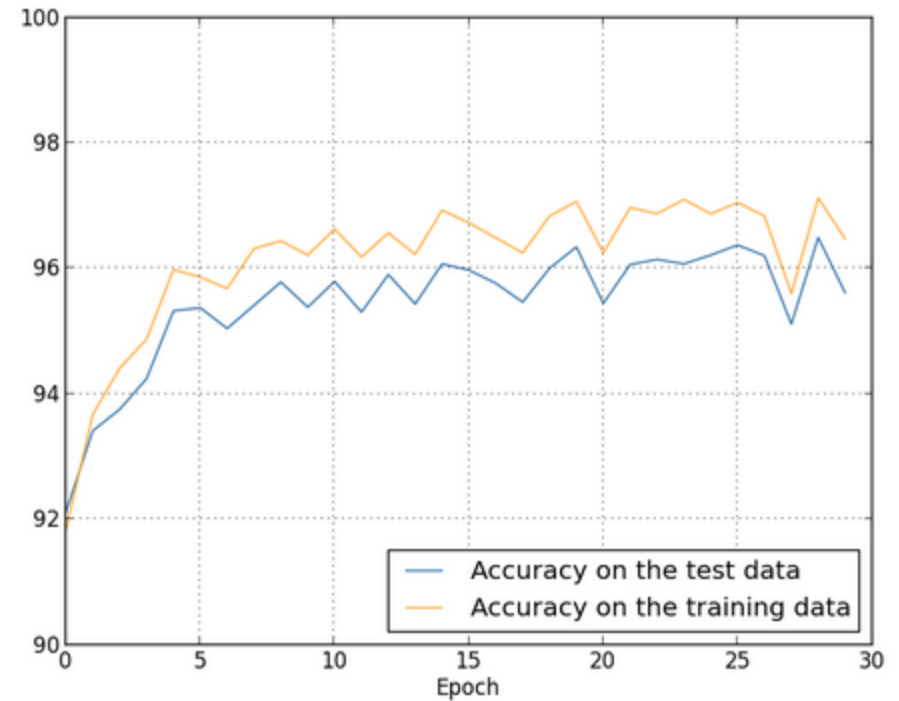


# Benefits of regularization

- With all the training samples:

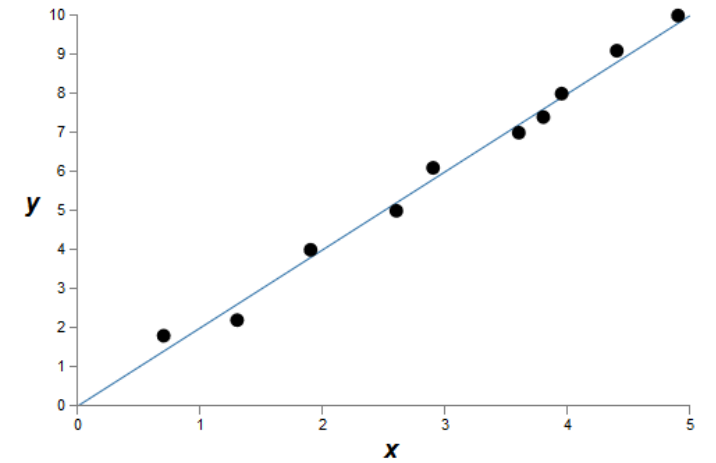
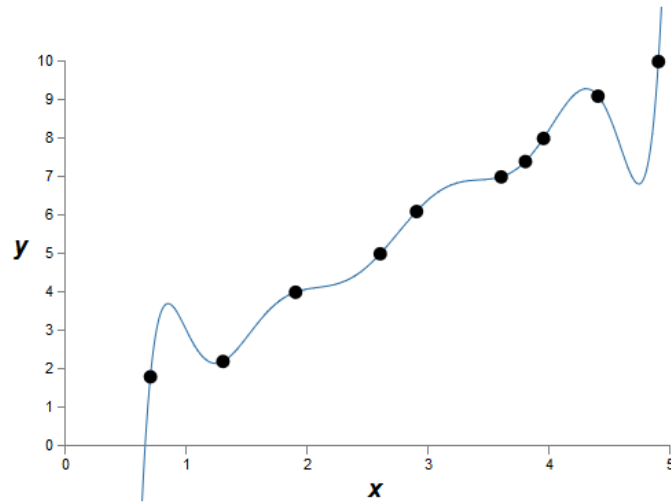
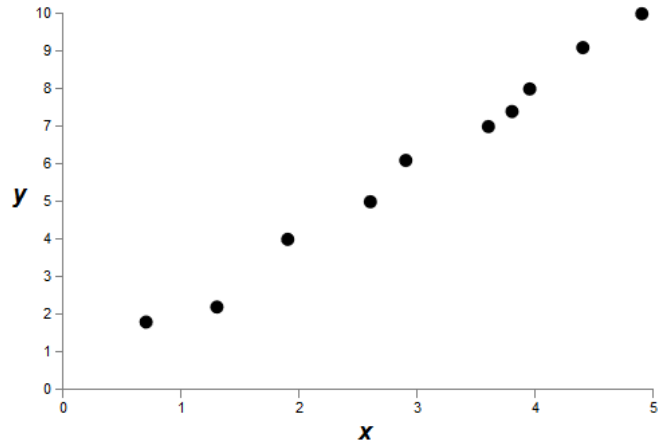


Before



After

# Why regularization reduces overfitting?



# L1 regularization

$$C = C_0 + \frac{\lambda}{n} \sum_w |w|$$

Intuitively it is same as L2 and tries to penalize large weights.

$$\frac{\partial C}{\partial w} = \frac{\partial C_0}{\partial w} + \frac{\lambda}{n} \text{sgn}(w)$$

Update rule:

$$w \rightarrow w - \frac{\eta \partial C_0}{\partial w} - \frac{\eta \lambda}{n} \text{sgn}(w)$$



# Difference between L1 and L2 regularization

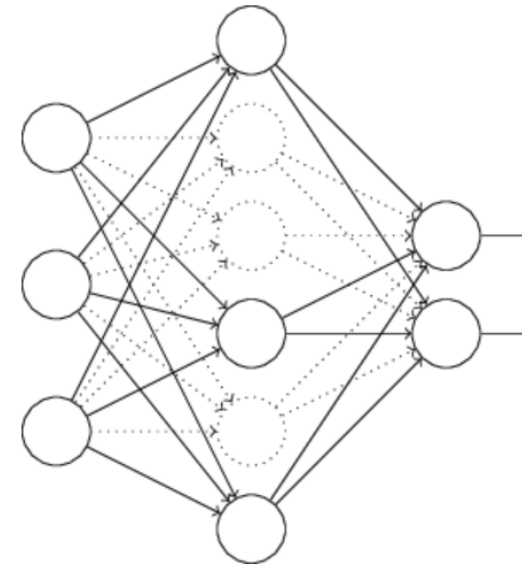
- In L1 case, the weights shrink by a constant amount towards 0.
- In L2 case, the weights shrink by an amount that is proportional to  $w$ .
- When the weight has a large magnitude  $|w|$ , then the L1 regularization shrinks less than the L2.
- When the weight has a small magnitude  $|w|$ , then the L1 regularization shrinks more than the L2.
- The net result is that the L1 regularization focuses on the weights of a few important connections and the rest are driven to zero.

# Corner case in computation of partial derivatives

- $\frac{\partial C}{\partial w}$  is not defined when  $w = 0$ . In such scenarios when  $w = 0$ , we use unregularized rule for stochastic gradient descent.

# Dropout

- We modify the network itself in dropout.
- We start by randomly (and temporarily) deleting half the hidden neurons in the network, while leaving the input and output neurons untouched.
- Note that the dropout neurons, i.e., the neurons which have been temporarily deleted, are still ghosted in.
- By repeating this process over and over, our network will learn a set of weights and biases.
- Of course, those weights and biases will have been learnt under conditions in which half the hidden neurons were dropped out.
- When we actually run the full network that means that twice as many hidden neurons will be active. To compensate for that, we halve the weights outgoing from the hidden neurons.



# Dropout

- This dropout procedure may seem strange and *ad hoc*.
- Imagine we train several different neural networks, all using the same training data.
- Of course, the networks may not start out identical, and as a result after training they may sometimes give different results.
- When that happens we could use some kind of averaging or voting scheme to decide which output to accept.
- For instance, if we have trained five networks, and three of them are classifying a digit as a "3", then it probably really is a "3". The other two networks are probably just making a mistake.
- This kind of averaging scheme is often found to be a powerful (though expensive) way of reducing overfitting.
- The reason is that the different networks may overfit in different ways, and averaging may help eliminate that kind of overfitting.

# Dropout

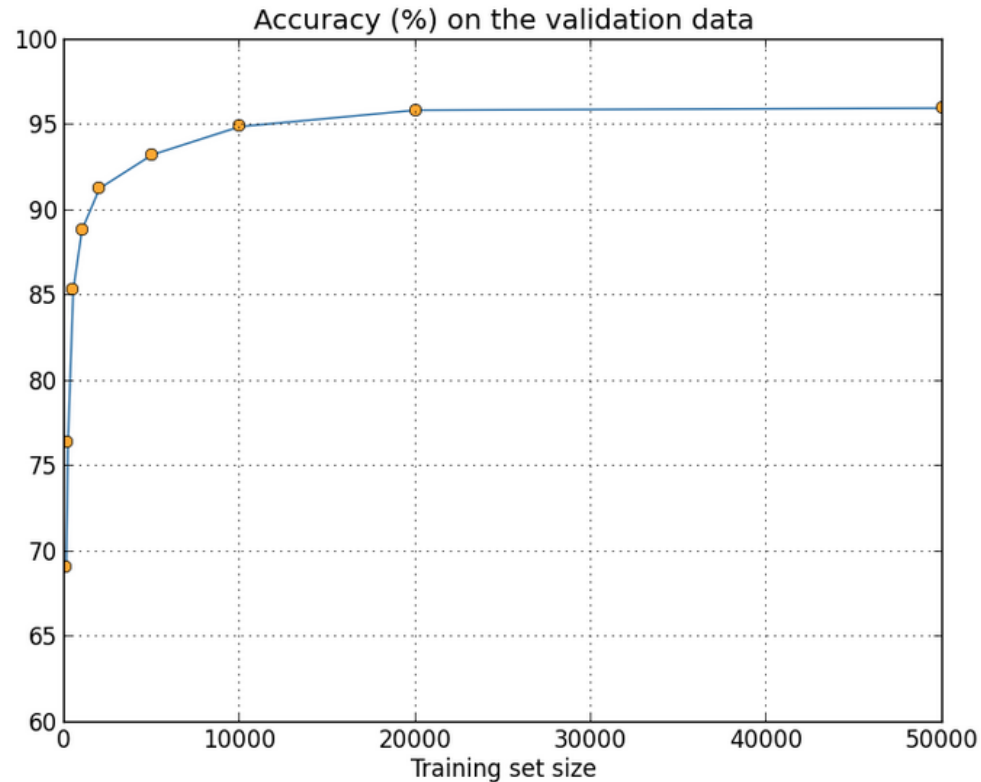
[ImageNet Classification with Deep Convolutional Neural Networks](#), by Alex Krizhevsky, Ilya Sutskever, and Geoffrey Hinton (2012).

- This technique reduces complex co-adaptations of neurons, since a neuron cannot rely on the presence of particular other neurons.
- It is, therefore, forced to learn more robust features that are useful in conjunction with many different random subsets of the other neurons.
- In other words, if we think of our network as a model which is making predictions, then we can think of dropout as a way of making sure that the model is robust to the loss of any individual piece of evidence.
- In this, it's somewhat similar to L1 and L2 regularization, which tend to reduce weights, and thus make the network more robust to losing any individual connection in the network.

# True success of dropout

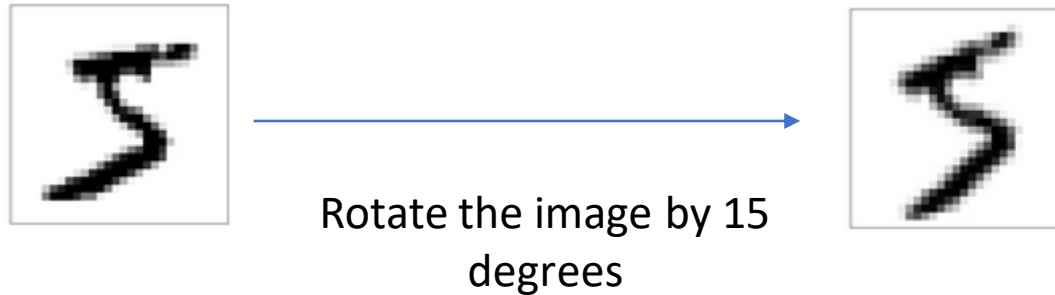
- It has shown improvement in a wide variety of problems.

# Training data size



- More training data helps to improve the performance.
- However, it is not always a feasible solution.

# Artificial training data



- We can expand our training data by making *many* small rotations of *all* the MNIST training images, and then using the expanded training data to improve our network's performance.
- This idea is very powerful and has been widely used.



# Artificial training data

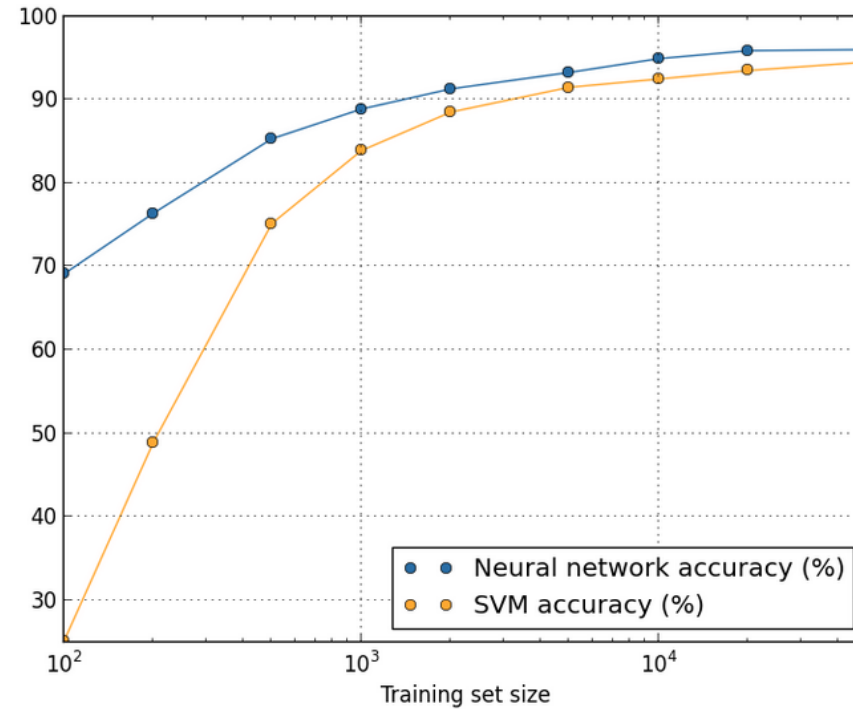
[Best Practices for Convolutional Neural Networks Applied to Visual Document Analysis](#), by Patrice Simard, Dave Steinkraus, and John Platt (2003).

- A feedforward network with 800 hidden neurons and using the cross-entropy cost function.
- Running the network with the standard MNIST training data they achieved a classification accuracy of 98.4 percent on their test set. But then they expanded the training data, using not just rotations, as I described above, but also translating and skewing the images. By training on the expanded data set they increased their network's accuracy to 98.9 percent.
- They also experimented with what they called "elastic distortions", a special type of image distortion intended to emulate the random oscillations found in hand muscles.
- By using the elastic distortions to expand the data they achieved an even higher accuracy, 99.3 percent. Effectively, they were broadening the experience of their network by exposing it to the sort of variations that are found in real handwriting.

# Question

- As discussed above, one way of expanding the MNIST training data is to use small rotations of training images. What's a problem that might occur if we allow arbitrarily large rotations of training images?

# SVM



- Increasing the training data also improves the performance in SVM.

# Dataset

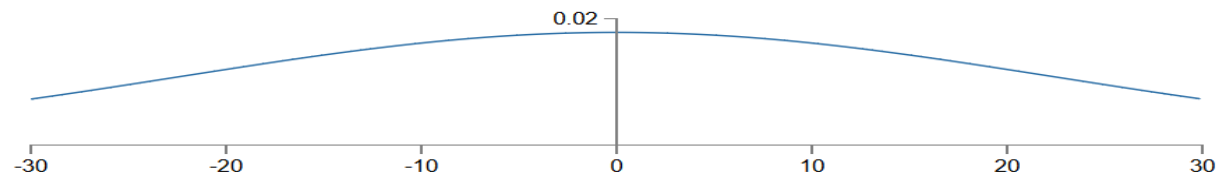
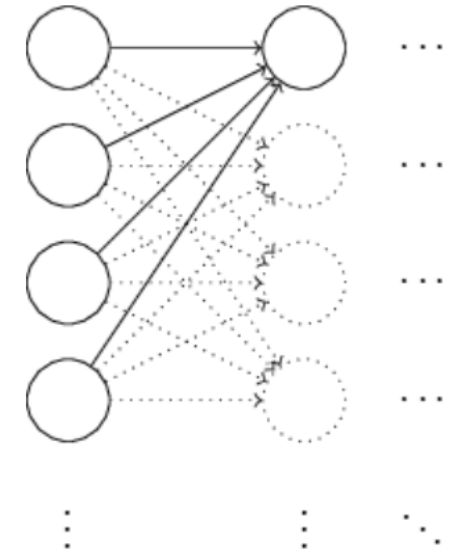
- Suppose we're trying to solve a problem using two machine learning algorithms, algorithm A and algorithm B.
- It sometimes happens that algorithm A will outperform algorithm B with one set of training data, while algorithm B will outperform algorithm A with a different set of training data.
- [Scaling to very very large corpora for natural language disambiguation](#), by Michele Banko and Eric Brill (2001).
- The correct response to the question "Is algorithm A better than algorithm B?" is really: "What training data set are you using?"

# Weight initialization

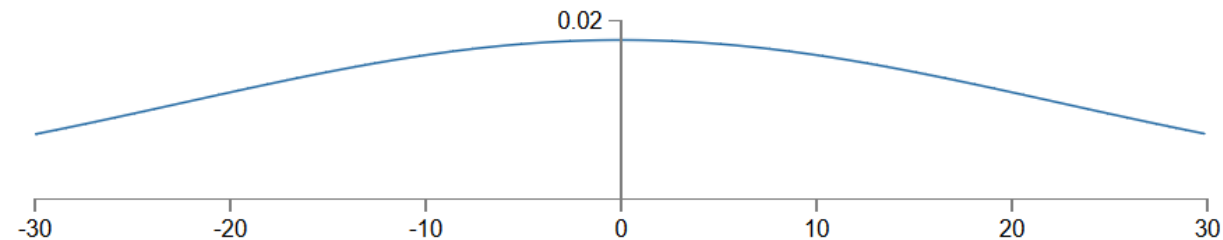
- One way to do it is choose both the weights and biases using independent Gaussian random variables, normalized to have mean 0 and standard deviation 1.
- Is this a good approach?

# Weight initialization

- Let us assume that there are 1000 input neurons.
- Normalized Gaussians to initialize the weights connecting to the first hidden layer.
- For simplicity, assume that half the input neurons are 1s and the remaining 0s.
- Consider the weighted sum of the input neurons  $\sum w_j x_j + b$ . 500 terms will vanish.
- $z$  is the sum over 501 normalized Gaussian variables accounting for 500 weight terms and 1 additional bias.
- $z$  is itself a Gaussian distribution with zero mean and standard deviation  $\sqrt{501} \approx 22$ .



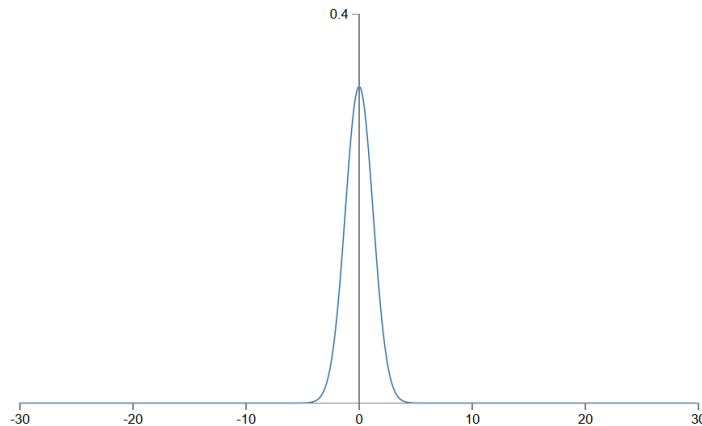
# Weight initialization



- $|z|$  will be large, i.e.,  $z \gg 1$ , or  $z \ll -1$ . Thus  $\sigma(z)$  will be very close to 0 or 1. In other words, the hidden neuron is saturated.
- Making small changes in the weights will make absolutely miniscule changes in the activation of our hidden neuron.
- Output neurons are saturated on the wrong value causes learning to slow down. This can be addressed using cross-entropy loss compared to the quadratic one.
- Although it helped saturated output neurons, it does nothing at all for the problem with saturated hidden neurons.

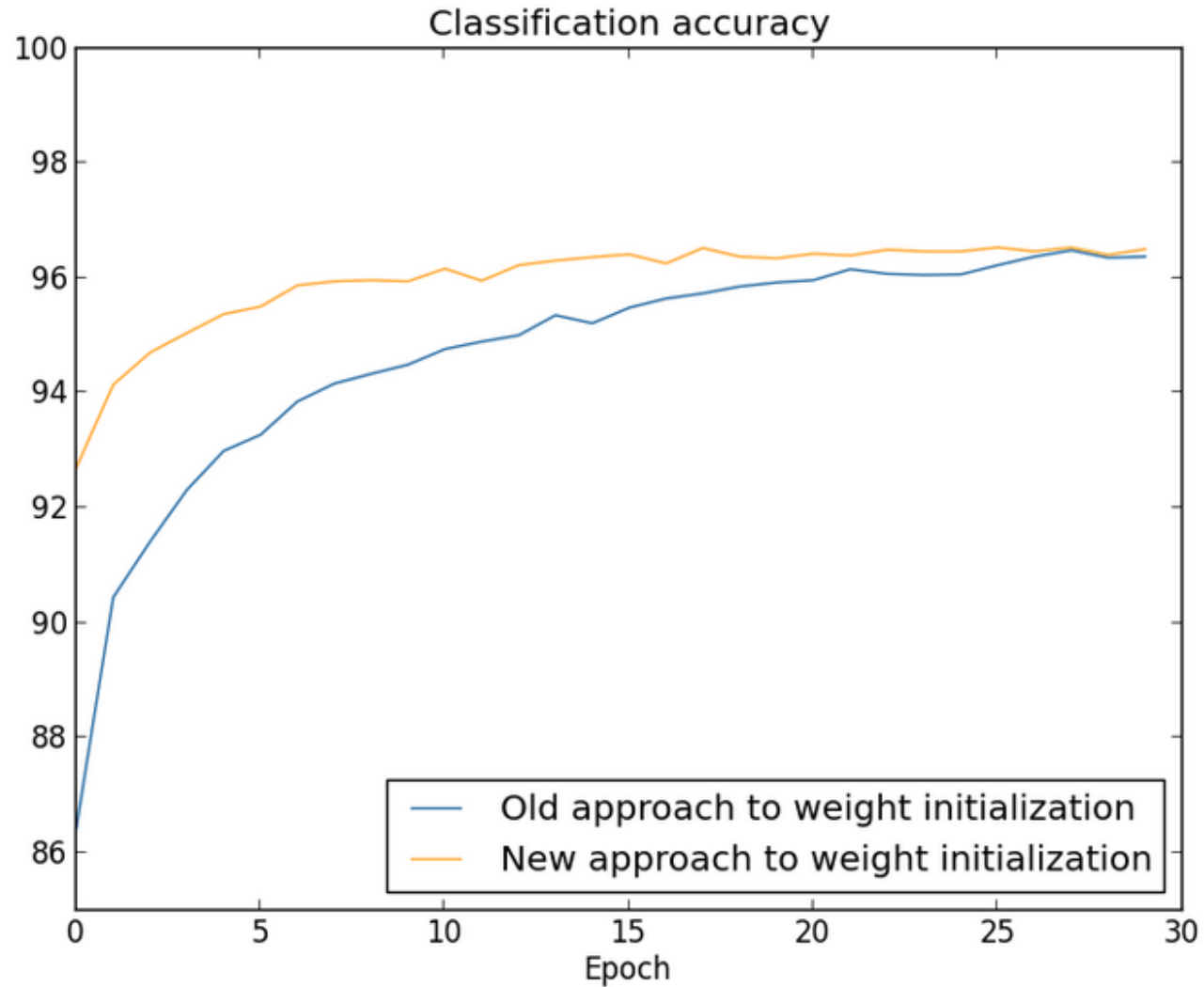
# Alternative initialization

- Suppose we have a neuron with  $n_{in}$  input weights. Let us initialize these weights with Gaussian variables with zero mean and standard deviation  $\frac{1}{\sqrt{n_{in}}}$ .
- Let the bias be a Gaussian with zero mean and standard deviation 1.
- The weighted sum will again be a Gaussian random variable with zero mean and standard deviation  $\sqrt{3/2}$ . The variance of a sum of independent random variables is the sum of the variances of the individual random variables





# Initialization approaches

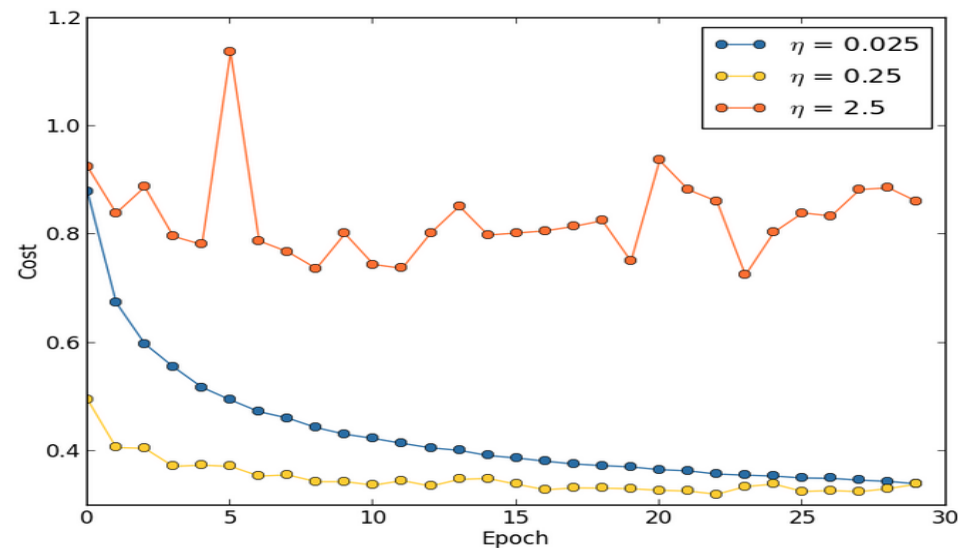


# Initialization approaches

- The final classification accuracy is almost exactly the same in the two cases.
- The new initialization technique brings us there much, much faster. At the end of the first epoch of training the old approach to weight initialization has a classification accuracy under 87 percent, while the new approach is already almost 93 percent.
- What appears to be going on is that our new approach to weight initialization starts us off in a much better regime, which lets us get good results much more quickly.

# How to choose neural network hyperparameters?

- Learning rate  $\eta$
- Procedure: Find the largest learning rate for which the cost decreases during the first few iterations and use this as threshold.
- To remain stable, we use a learning rate that is smaller than the threshold, i.e., a factor of 2 compared to the normal threshold.



# Early Stopping to avoid overfitting

- Early stopping means that at the end of each epoch we should compute the classification accuracy on the validation data. When that stops improving, terminate.
- For robustness, we modify the criteria to do early stopping if the classification accuracy hasn't improved during the last  $n$  epochs.

# Learning rate schedule

- The idea is to hold the learning rate constant until the validation accuracy starts to get worse.
- Then decrease the learning rate by some amount, say a factor of two or ten.
- We repeat this many times, until, say, the learning rate is a factor of 1,024 (or 1,000) times lower than the initial value. Then we terminate.

# Regularization parameter

- Start with no regularization  $\lambda = 0$  and find the learning rate  $\eta$ .
- Using the identified learning rate, initialize  $\lambda = 1$ .
- Increase or decrease the regularization parameter by factors of 10 to see improvement in the validation set.
- Then finetune the regularization parameter.
- Then go back and reoptimize the learning rate again.

# Minibatch size

- Online learning: Use mini-batch of size 1.

$$w \rightarrow w' = w - \eta \nabla C_x$$

- Use mini-batch of size 100.

$$w \rightarrow w' = w - \eta \frac{1}{100} \sum_x \nabla C_x,$$

- There are good matrix libraries for efficient computation. If mini-batch size is too small, you are not utilizing the advantage from matrix libraries. If too large, you are not updating enough.
- Try different mini-batch sizes and choose the one that decreases the cost function efficiently.

# Training or validation set?

- Learning rate
- Regularization parameter
- Early stopping
- Mini-batch size




# Variants of stochastic gradient descent

- Hessian technique

Using Taylor's theorem:

$$C(w + \Delta w) = C(w) + \nabla C \cdot \Delta w + \frac{1}{2} \Delta w^T H \Delta w + \dots,$$

Hessian matrix 

Hessian optimization refers to the process of minimizing the cost function using Hessian and this tends to be more accurate than simple gradient, but very inefficient.

# Momentum-based gradient descent

- The momentum technique modifies gradient descent in two ways that make it more similar to the physical picture of a ball rolling down a terrain.
- First, it introduces a notion of "velocity" for the parameters we're trying to optimize. The gradient acts to change the velocity, not (directly) the "position", in much the same way as physical forces change the velocity, and only indirectly affect position.
- Second, the momentum method introduces a kind of friction term, which tends to gradually reduce the velocity.

# Momentum-based gradient descent

- Let us introduce velocity variables for each of the weight parameter, i.e.  $v = v_1, v_2, \dots$ , for each of the corresponding  $w_j$  variable.

$$\begin{aligned}v &\rightarrow v' = \mu v - \eta \nabla C \\w &\rightarrow w' = w + v'\end{aligned}$$

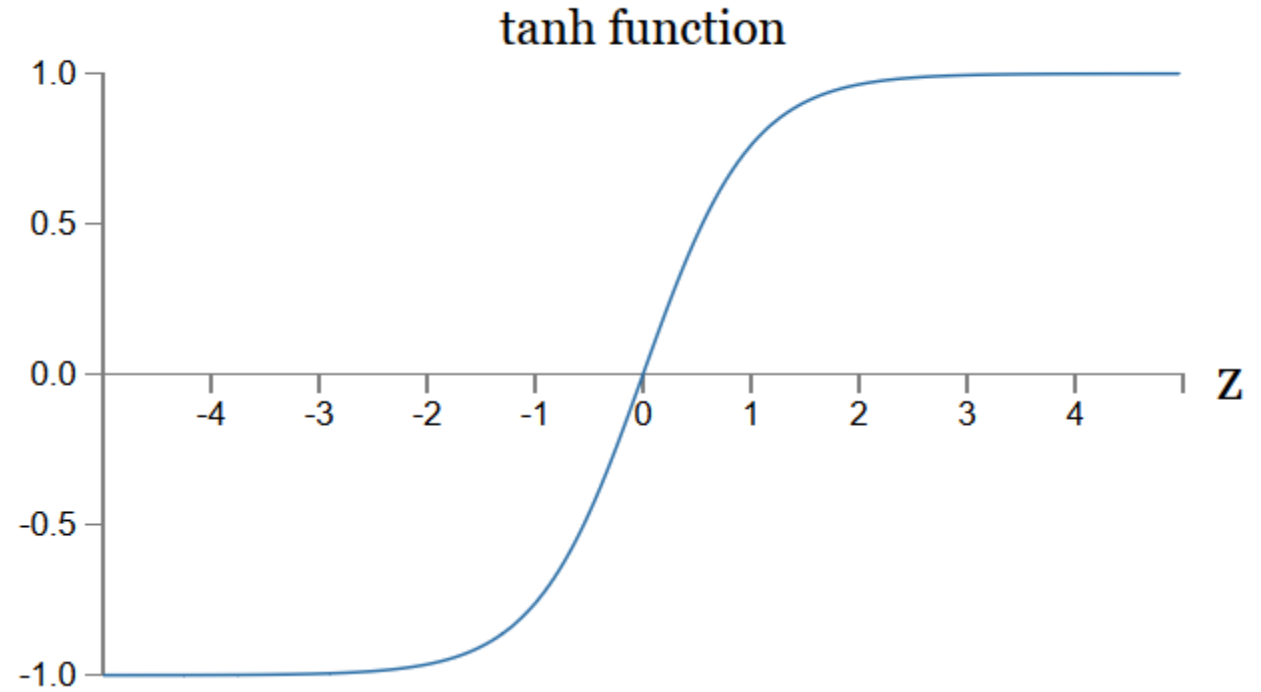
- Here the hyperparameter  $\mu$  controls the amount of damping and friction in the system. It is referred to as the momentum co-efficient and it a bad name since it does not really refer to the actual momentum, but rather refers to the friction.
- What happens when the hyperparameter is greater than 1, or less than 0?

# Hyperbolic tangent

$$\tanh(w \cdot x + b)$$

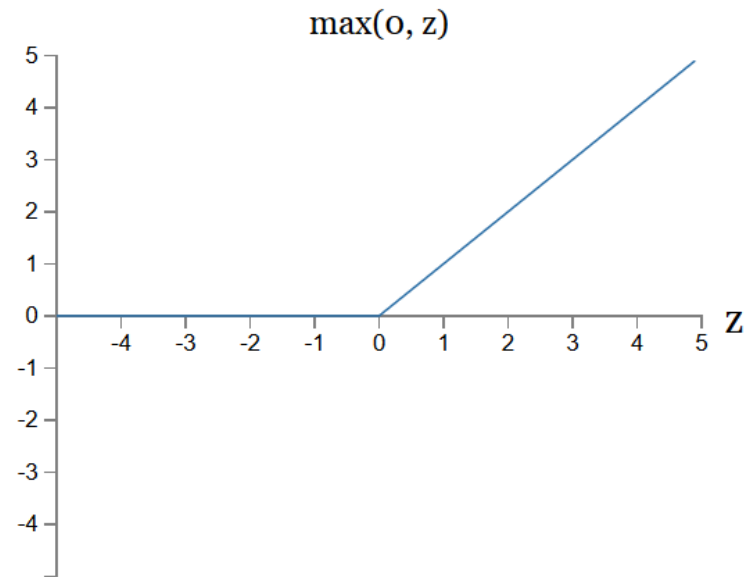
$$\tanh(z) \equiv \frac{e^z - e^{-z}}{e^z + e^{-z}}$$

$$\sigma(z) = \frac{1 + \tanh(z/2)}{2}$$



# Rectified linear neuron

$$\max(0, w \cdot x + b)$$



- By contrast, increasing the weighted input to a rectified linear unit will never cause it to saturate, and so there is no corresponding learning slowdown. On the other hand, when the weighted input to a rectified linear unit is negative, the gradient vanishes, and so the neuron stops learning entirely.
- Preferred choice for many computer vision problems.

Thank You