CS 5600/6600: F20: Intelligent Systems

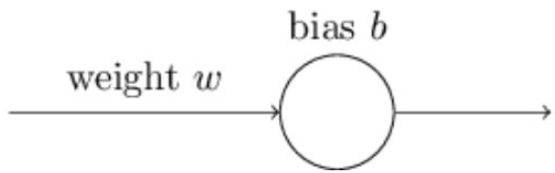
Why NNs are Hard to Train
and
NN Training Tips and Best Practices

Vladimir Kulyukin
Department of Computer Science
Utah State University

Outline

Neuron 1

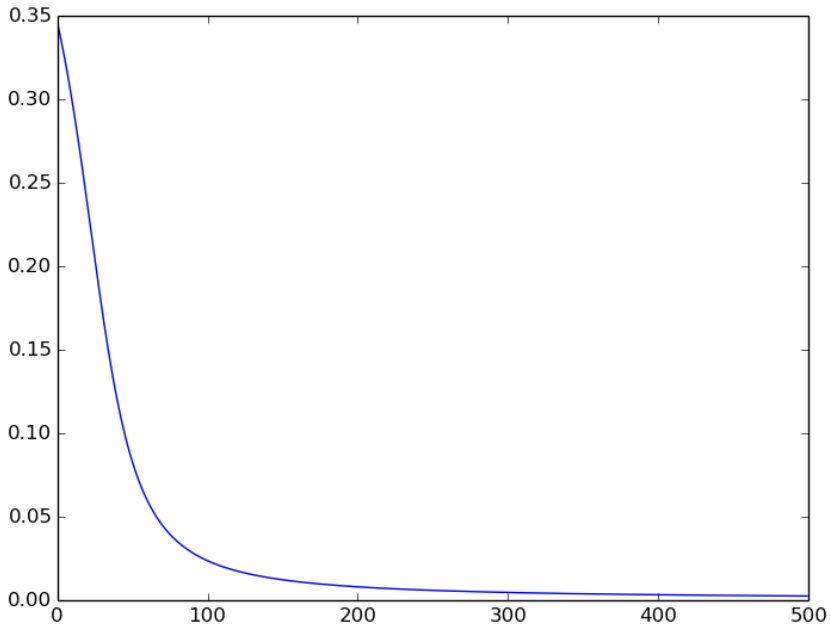
Let's create a simple neuron by setting $w - 1.28$, $b = -0.98$, $\eta = 0.15$. Our job is to train this neuron to take an input of 1 and convert to an input of 0. That simple! Let's use the quadratic cost function (MSE).



$\sigma(w \cdot x + b) = \sigma(-1.28 \cdot 1 - 0.98) \approx 0.09$ so this neuron must do some learning.

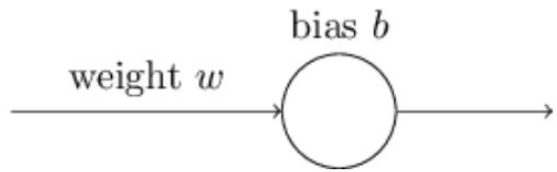
Learning in Neuron 1: Epoch vs. Cost

Let's train the neuron defined on the previous slide for 500 epochs (each epoch consists of just one training input of 1) and graph the costs. The graph shows that the neurons learns rapidly a weight and a bias that drive down the cost to 0.



Neuron 2

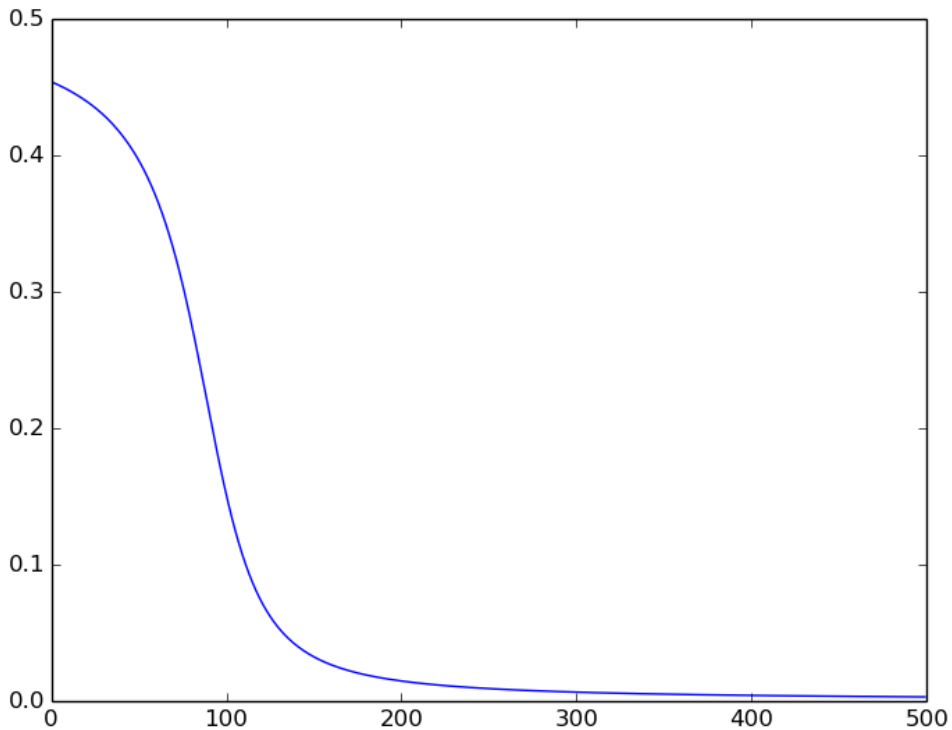
Let's create another version of the simple neuron by setting $w = b = 2.0$, and $\eta = 0.15$. We'll use the quadratic cost function.



$\sigma(w \cdot x + b) = \sigma(2.0 \cdot 1 + 2.0) \approx 0.98$ so this neuron must do even more learning than version 1.

Learning in Neuron 2: Epoch vs. Cost

Let's train the neuron defined on the previous slide for 500 epochs and graph the costs. The graph shows that the neurons learns more slowly than neuron 1.



Why Does Learning Slow Down?

Basic reason: An NN's learning slows down when many of its neurons are off the mark then when they're closer to the mark.

Let's try to understand why. The neuron learns by changing the weight and bias at a rate determined by two partial derivatives of the cost function $\partial C/\partial w$ and $\partial C/\partial b$.

Why Does Learning Slow Down?

Let's compute the partial derivatives. The cost function is

$$C = \frac{(y-a)^2}{2},$$

where a is the neuron's output when the training input $x = 1$ and $y = 0$ is the target output. Then

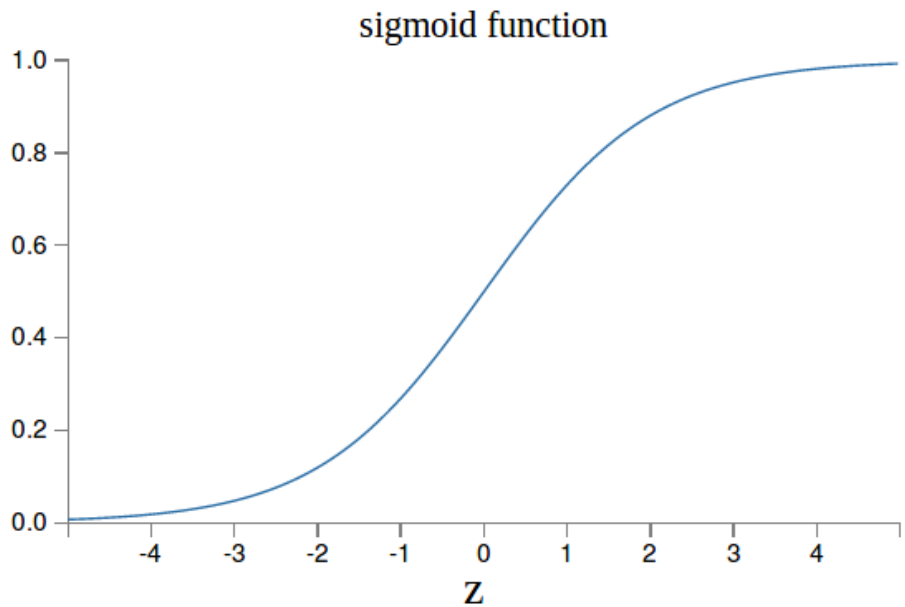
$$\frac{\partial C}{\partial w} = (a-y)\sigma'(z)x = a\sigma'(z)$$

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

because $x = 1$ and $y = 0$.

Why Does Learning Slow Down?

Recall the shape of the sigmoid function. When the neuron's output is close to 1 or 0, the curve gets flat, which makes $\sigma'(z)$ very small. Therefore, $\partial C/\partial w$ and $\partial C/\partial b$ become very small.



Learning Slowdown

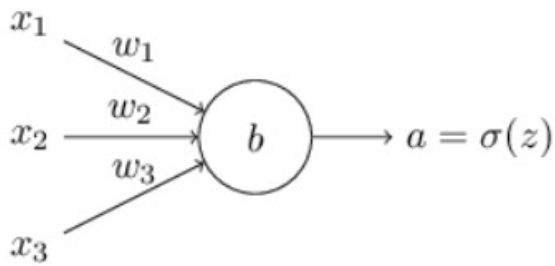
The learning slowdown can be a problem for all NNs.

A best practice to address this problem when it occurs is to replace the quadratic cost with a different cost function such as cross-entropy and softmax.

Outline

Cross-Entropy Formula

Suppose that we have a neuron with several inputs $x_1, x_2, ..., x_n$ and weights $w_1, w_2, ..., w_n$, a bias b, and output $a = \sigma(z)$, where $z = \sum_j w_j x_j + b$.



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

where n is the total number of training data items (i.e., examples).

Cross-Entropy Formula

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

Properties of Cross-Entropy:

1. $C > 0$;
2. $C \approx 0$ when $y = 0$ and $a \approx 0$ or $y = 1$ and $a \approx 1$.

In other words, the cross-entropy is positive and tends to 0 as the neuron gets closer to computing the desired output.

What's the big deal? The quadratic cost function also satisfies these properties. The big deal is that cross-entropy avoids the learning slowdown.

How Cross-Entropy Addresses Learning Slowdown

Recall that $a = \sigma(z)$. Then

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

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

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

$$\frac{1}{n}\sum_x x_j(\sigma(z)-y).$$

How Cross-Entropy Addresses Learning Slowdown

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

This equation shows that the rate at which the weight learns is controlled by $\sigma(z) - y$, i.e., the error in the output. The larger the error, the faster the neuron learns.

A really cool thing is that when we use cross-entropy the term $\sigma'(z)$ cancels out. Thus, we are no longer dependent on it. Recall your experiences with $\sigma'(x)$ in Assignment 2.

How Cross-Entropy Addresses Learning Slowdown

We can similarly compute the derivative of C with respect to b:

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

Again, when we use cross-entropy the term $\sigma'(z)$ cancels out.

Generalizing Cross-Entropy to Multilayer Networks

Let $y = y_1, y_2, \ldots$ be the target output values at the output neurons of a multilayer network. Let $a = a_1^L, a_2^L, \ldots$ be the actual output values of the same multilayer network. Then the cross-entropy C is defined as

$$C = -\frac{1}{n} \sum_x \sum_j [y_j ln(a_j^L) + (1 - y_j) ln(1 - a_j^L)].$$

This is the same as the previous formula except for the second embedded sum that sums over all outputs a_j^L.

Is Cross-Entropy Better than MSE?

Cross-entropy is almost always a better choice provided the output neurons are sigmoid.

Why? Since the initial weights are initialized randomly, it may happen that these initial choices result in the network that computes a really wrong output for some output. If this network uses cross-entropy, it'll likely learn faster than if it uses MSE.

Outline

Another Approach to Learning Slowdown

The idea of softmax is to define a new type of output layer for NNs.

Let the weighted input be defined as $z_j^L = \sum_k w_{jk}^L a_k^{L-1} + b_j^L$.

However, instead of applying the sigmoid function σ to each z_j^L, we'll apply the so-called softmax function to z_j^L, which is defined as

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

where in the denominator we sum over all the output neurons.

Softmax

As z_i^L increases, a_i^L increases and all other output activations decrease. As z_i^L decreases, a_i^L decreases, and all the other output activations increase. In other words,

$$\sum_j a_j^L = \frac{\sum_j e^{z_j^L}}{\sum_k e^{z_k^L}} = 1.$$

One can interpret a_j^L as the probability that the correct classification is j. One can think of softmax as a way of rescaling z^L to form a probability distribution.

Log-Likelyhood Cost Function

Let's define the log-likelihood cost function.

$$C \equiv -ln(a_y^L).$$

Why is this a cost function? When a_y^L is close to 1, then C is small. When a_y^L is smaller, C is larger.

Softmax with Log-Likelihood Cost

In a network with a softmax output and log-likelihood cost, we have

$$\frac{\partial C}{\partial b_j^L} = a_j^L - y_j$$

$$\frac{\partial C}{\partial w_{jk}^L} = a_k^{L-1}(a_j^L - y_j)$$

Note that the derivatives are gone on the right hand sides of both equations.

Practical Insight

Should you use a sigmoid output layer and cross-entropy or a softmax output layer and log-likelihood?

Both approaches work well. Many ANNs use sigmoid output layers with cross-entropy. Many convolutional neural networks (ConvNets) use softmax with log-likelihood.

Additional note: Softmax plus log-likelihood is used when we want to interpret the output activations as probabilities.

Outline

Signs of Overfitting

Sign 1: Cost on the training data decreases while accuracy on the test data slows down and stops growing.

Sign 2: Cost on the training data decreases while cost on the test data increases.

Sign 3: Accuracy on the training data quickly reaches 100% and stays 100% for a considerable number of epochs whereas accuracy on the test data is significanly lower (e.g., 70% or 80%).

Detection of Overfitting

Test Accuracy Stopping: Keep track of accuracy on the test data as your network trains. If the accuracy on the test data is no longer improving, stop training.

Validation Accuracy stopping: Compute the classification accuracy on the validation data at the end of each epoch. Once the classification accuracy on the validation data has saturated, stop training.

Training Data Size: One of the best ways to reduce overfitting is to increase the size of training data. Not always possible, because training data can be expensive to acquire.

Cost Function Regularization

Is there a way to reduce overfitting when we have a fixed network and fixed training data?

A common technique is called *L2 regularization* or weight decay when a a *regularization term* is added to the cost function.

Here is a generic regularized cost function formula:

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

where $\lambda > 0$ is the regularization parameter and n is the size of the training data set. Note that the regularization term added to the cost function doesn't include the biases.

Closer Look at Regularization

Here is a generic regularized cost function formula:

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

Large weights are allowed if they improve the first part of the regularized cost function.

When λ is small, preference is given to minimizing the cost function, when λ is larger, preference is given to finding smaller weights.

Regularized MSE and Cross-Entropy

Regularized MSE:

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

Regularized Cross-Entropy:

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

Derivatives of Regularized Cost Function

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

$$\frac{\partial C}{\partial w} = \frac{\partial C_0}{\partial b}.$$

Regularized Gradient Descent Updates

Weight modification rule:

$$w \to w - \eta \frac{\partial C_0}{\partial w} - \frac{\eta\lambda}{n} w = \left(1 - \frac{\eta\lambda}{n}\right) w - \eta \frac{\partial C_0}{\partial w}.$$

Bias modification rule:

$$b \to b - \eta \frac{\partial C_0}{\partial b}.$$

Pros and Cons of Regularization

Regularization works, heuristically speaking, because it prevents the vectors from becoming too large.

If the cost function is unregularized, the length of the vector can go arbitrarily large.

Large vectors may get stuck pointing in the same direction since changes due to gradient descent tend to make small changes in the direction when the length is long.

Pros and Cons of Regularization

Regularized networks tend to have smaller weights. If there are small weights, the behavior of the network does not change too much if we change a few random weights. This makes it difficult for regularized networks to learn the effects of local noise in the data.

But! Regularlized networks tend to respond to types of evidence seen accross the training set. On the other hand, unregularized networks with larger weights may change its behavior in response to smaller changes in the input due to large weights.

Another Look at Regularization

Regularization gives us some tools to help networks generalize better.

Regularization does not give us any deeper understanding of how generalization works in the human brain.

In everyday life, humans generalize from small sets of examples.

Another Look at Regularization

Hypothesis: The dynamics of gradient descent learning in multilayer nets has a "self-regularization" effect.

Source: Y. LeCun, L. Bottou, Y. Bengio, P. Haffner. "Gradient-Based Learning Applied to Document Recognition."

This is a hypothesis that seems to hold.

L2 Regularization and Biases

Why does L2 regularization not constraint the biases?

We can, but it's not worth it, because a large bias doesn't make a neuron sensitive to its inputs in the same way as having large weights.

Large biases give networks more flexibility by making it easier for neurons to saturate, which may be desirable under some circumstances.

Other Techniques to Deal with Overfitting

Other common techniques to handle overfitting are:

1. Dropout;
2. Increasing training set size;
3. Hyperparameter Adjustment;
4. Other Neuron Models;
5. L1 regularization.

Dropout

Dropout is different from regularization, because it does not modify the cost function, but the network itself.

Dropout Procedure:

1. Randomly select and disable temporarily half the hidden neurons in the network;
2. Leave the input and output neurons intact;
3. Forward and backward propagate the modified network over all examples in a mini-batch;
4. Go to 1 as many times as you want.

Insight 1 on Dropout

When we dropout different sets of neurons, we are training different networks.

Dropout approximates the effect of having many different networks vote on the input.

Since different networks overfit in different ways, the net effect is likely to reduce overfitting.

Insight 2 on Dropout

Dropout reduces complex co-adaptations of neurons, because a neuron cannot rely on the presence of any particular set of neurons.

Each neuron is forced to learn more robust features useful in conjunction with many different random subsets of the other neurons.

Dropout ensures that the network is robust to the loss of any individual piece of evidence.

Artificial Expansion of Training Data

Obtaining more data is great but often impossible.

Artificial expansion of data means adding to data sets items obtained from the existings data items:

1. Image rotation;
2. Image translation;
3. Image skewing;
4. Adding some noise to audio files.

It's OK to look for better algorithms but never look for better algorithms at the exclusion of more and better data.

Outline

Interesting Quote

With four parameters I can fit an elephant, and with five I can make him wiggle his trunk.

Jon von Neumann

Numbers of Parameters and Accuracy

The point of von Neumann's quote is that models with many free parameters can describe wide ranges of phenomena.

A model with many free parameters may work well for a given dataset but may fail to generalize to new situations.

In modern ANNs, we have tens or hundreds of thousands of parameters. How much trust should we put into the accuracies of such "trained" networks?

Which Hyperparameters to Adjust

Typically, there is no a priori knowledge on which hyperparameters you should adjust.

Should you use 30 or 200 hidden neurons? How many layers of hidden neurons? For how many epochs should you train? Are mini-batches too small? Should you use cross-entropy or switch back to MSE? Should you initialize your weights differently? Should you abandon ANNs and try a different method?

Choosing hyperparameters is not a solved problem. But, there are heuristics.

Downsizing and Monitoring

You need to get your ANN to start learning, i.e., it should get better than chance.

The first strategy is to downsize the training data (e.g., take the images of only 0's and 1's and train on those).

Another strategy is to increase the frequency of monitoring. For example, instead of evaluating the performance after an epoch of 30,000 images, evaluate the performance after an epoch of 1,000 images. Instead of using 10,000 images in a validation set, use 500 images.

Now you can experiment with η's and λ's and get faster feedback.

Learning Rate η Threshold

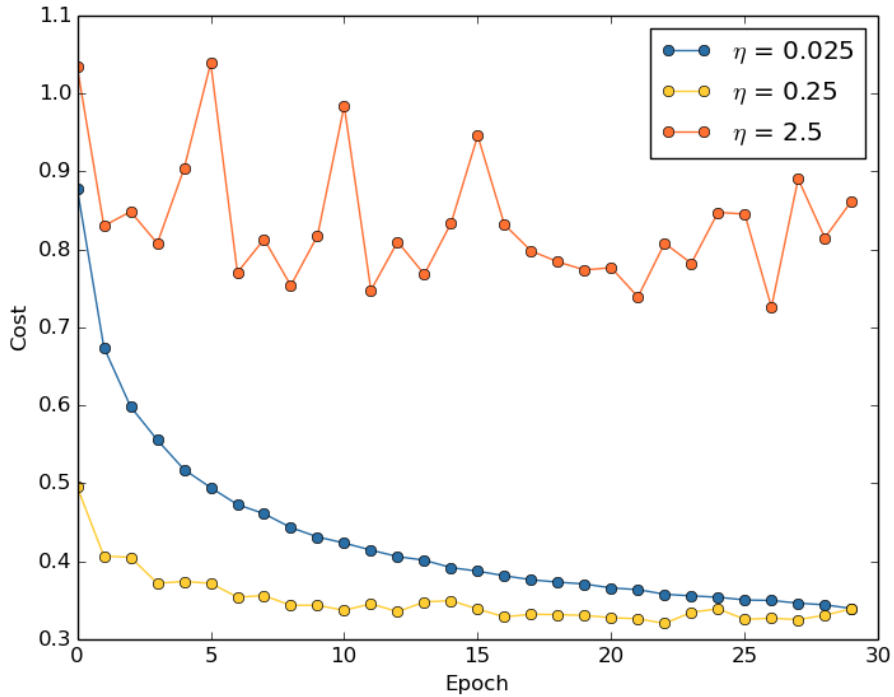
Start with an arbitrary value of η. If the cost oscilates or increases for the first few epochs, decrease η. If the cost decreases for the first few epochs, increase η.

For example, start with $\eta = 0.01$, if the cost decreases, try $\eta = 0.1, 1.0, 2.0, ...$ until you find a value where the cost oscilates or increases during the first few epochs.

If the cost oscilates or decreases for $\eta = 0.01$, try $\eta = 0.001, 0.0001, ...$ until you find a value for η where the cost decreases during the first few epochs.

The actual value for η that you end up using should not be larger that this threshold value you discovered.

Learning Rate η Threshold on MNIST



It looks like we shouldn't go higher than $\eta = 2.5$ on MNIST.

Determining the Number of Training Epochs

Compute the classification accuracy at the end of each epoch. If the accuracy does not improve for a given number of consecutive epochs, terminate.

Many researchers use the no-improvement-in-10 rule: if there is no improvement in 10 consecutive epochs, training stops automatically.

Caveat: your network may plateau for quite a few epochs before begining to train again. You can always use more leanient rules: no-improvement-in-20, no-improvement-in-30, etc.

Learning Rate Variation

We don't have to keep our learning rate constant. Hold it constant until the validation accuracy starts to get worse. Then decrease it by a factor of 2 or 10.

A typical strategy: halve the learning rate each time the validation accuracy satisfies the no-improvement-in-n rule and terminate when the learning rate has dropped to a fraction (e.g., 128) of its original value.

Regularization Parameter λ

Set λ initially to 0 and determine a reasonable value for η first.

Set $\lambda = 1.0$ and then increase by factors of 2 or 10.

Reoptimize η once a good value of λ is discovered.

Mini-batch Size

Is there anything wrong with using a mini-batch of size 1? No, the errors turn out to be insignificant and the ANN will learn just fine.

In practice, it is much faster to use a bunch of examples to compute and upgrade the gradient.

Choosing the best mini-batch size is a compromise. Too small a size doesn't let you take advantage of fast linear algebra packages. Too large a size doesn't let you update the gradient often enough.

You can plot the validation accuracy vs. time (real physical time, not epoch!) and choose whichever mini-size batch gives you the most rapid improvement in performance.

Final Words on Hyperparameters

Many papers and journal articles give contradictory recommendations on how to choose optimal hyperparameters.

A good paper on read is "Practical recommendations for gradient-based training of deep architectures" by Yoshua Bengio. You can also follow the references in this great paper.

Keep in mind that hyperparameter optimization is never completely solved.

Outline

Tanh Neuron

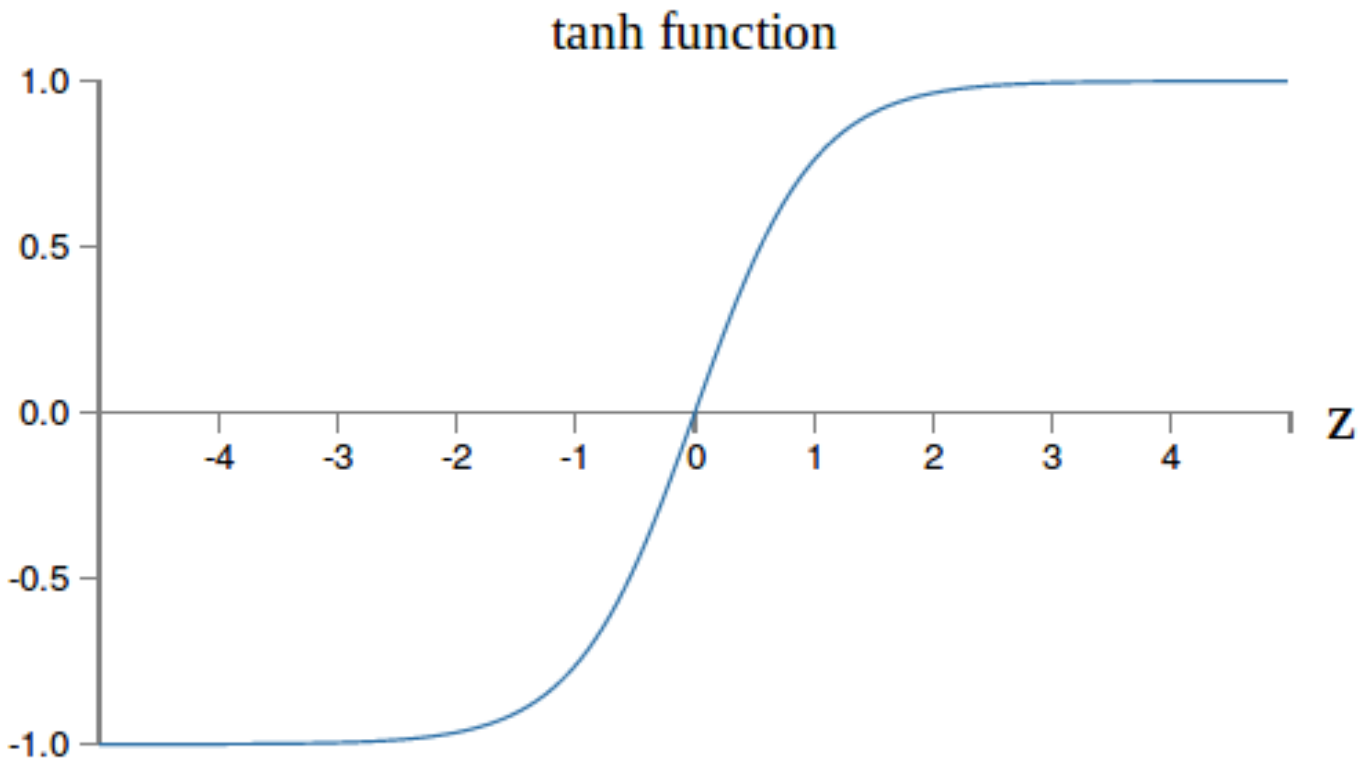
The tanh neuron replaces the sigmoid function by the hyperbolic tangent function.

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

where

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

Tanh Function



Sigmoid vs. Tanh

Why would you want to use tanh neurons? Suppose you use sigmoid neurons and all activations in our network are positive.

Consider w_{jk}^{l+1}, i.e., the weight input to the j-th neuron in layer $l+1$. By backpropagation, the associated gradient is $a_k^l \delta_j^{l+1}$. Since all activations are positive, the sign of this gradient is the same as the sign of δ_j^{l+1}.

If δ_j^{l+1} is positive, then all weights w_{jk}^{l+1} decrease during gradient descent; if δ_j^{l+1} is negative, then all weights w_{jk}^{l+1} increase during gradient descent. In other words, all weights to the same neuron must either increase or decrease.

Since $tanh(-z) = -tanh(z)$, the activations in hidden layers are expected to be equally balanced between positive and negative.

Sigmoid vs. Tanh

The argument in favor of tanh is heuristic.

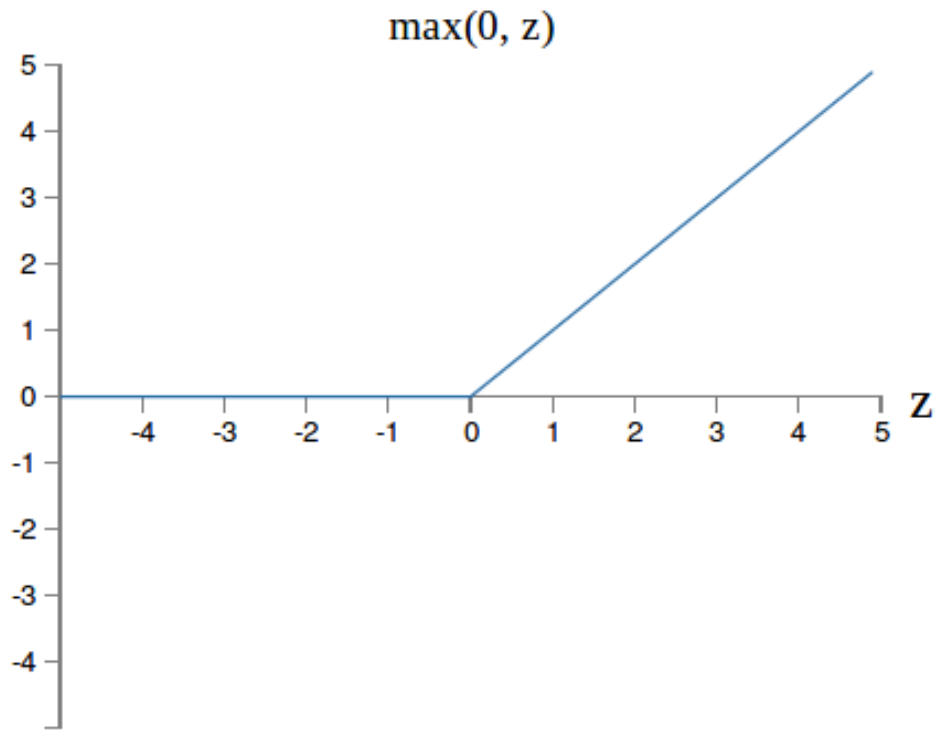
In many cases, tanh provides only a small or no improvement in performance over sigmoid neurons.

There are no hard-and-fast rules to know neuron types will learn fastest or give the best performance.

Rectified Linear Unit (ReLU)

The output of a ReLU with input x, weight vector w, and bias b is given by

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



Why Use ReLUs

Both sigmoid and tanh neurons start learning slowly when they saturate.

Increasing the weighted input to a ReLU never causes it to saturate. Thus, there is no learning slowdown.

But, when the weighted input to a ReLU is negative, the gradient vanishes. Thus, the neuron learns stops learning entirely.

L1 Regularization

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

This is similar to L2 insomuch as it makes networks prefer smaller weights.

Differentiating L1 Regularized Cost Function

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

where

$$sgn(w) = \begin{cases} +1 & \text{if } w > 0 \\ 0 & \text{if } w = 0 \\ -1 & \text{if } w < 0. \end{cases}$$

Gradient Descent Update Rule for L1 Regularized Network

$$w \to w' = w - \frac{\eta\lambda}{n} sgn(w) - \eta \frac{\partial C_0}{\partial w}.$$

Gradient Descent Update Rule for L1 Regularized Network

L1 regularization:

$$w \to w' = w - \frac{\eta\lambda}{n} sgn(w) - \eta\frac{\partial C_0}{\partial w}.$$

L2 regularization:

$$w \to w' = w\left(1 - \frac{\eta\lambda}{n}\right) - \eta\frac{\partial C_0}{\partial w}.$$

Key difference: L1 modifies weights by constant amounts; L2 modifies weights proportionally to w.

Practical insight: L1 tends to concentrate the weight of the network in a small number of connections while the other weights are driven to 0.

Weight Initialization

The weights and biases can be chosen using independent Gaussian random variables normalized to have a mean of 0 and a standard deviation of 1.

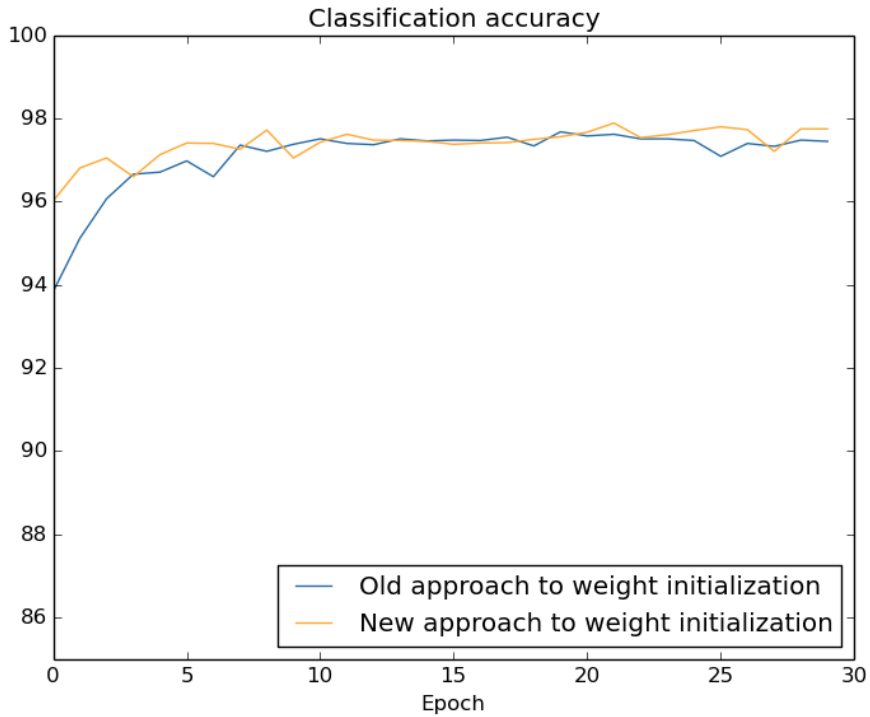
In this case, z tends to have a very broad Gaussian distribution, not sharply peaked, and the output $\sigma(z)$ from a hidden neuron will tend to be either 0 or 1. Consequently, learning will slow down.

Weight Initialization Trick

Suppose there is a neuron with n_{in} input weights. We can initialize the weights as random Gaussians with a mean of 0 and a standard deviation of $1/\sqrt{n_{in}}$.

The Gaussians will be squashed down and it's less likely that the neuron will saturate.

Old vs. New Approach to Weight Initialization on MNIST



In the end both weights get us there, but the new weight initialization gets there a bit faster.

Bias Initialization

It doesnt' matter much how we initialize the biases if the problem of neuron saturation is minimized.

Some researchers initialize all biases to 0 and rely on gradient descent to learn the appropriate biases.

It's quite OK to initialize the biases as random Gaussians with a mean of 0 and a standard deviation of 1.