# The Generalized Delta Rule and Practical Considerations

Introduction to Neural Networks : Lecture 6



1. Training a Single Layer Feed-forward Network
2. Deriving the Generalized Delta Rule
3. Practical Considerations for Gradient Descent Learning
   - (*1*) Pre-processing of the Training Data
   - (*2*) Choosing the Initial Weights
   - (*3*) Choosing the Learning Rate
   - (*4*) On-line *vs.* Batch Training
   - (*5*) Choosing the Transfer Function
   - (*6*) Avoiding Flat Spots in the Error Function
   - (*7*) Avoiding Local Minima
   - (*8*) Knowing When to Stop the Training

# Gradient Descent Learning

It is worth summarising all the factors involved in Gradient Descent Learning:

1. The purpose of neural network learning or training is to minimise the output errors on a particular set of training data by adjusting the network weights $w_{ij}$.
2. We define an Error Function $E(w_{ij})$ that "measures" how far the current network is from the desired (correctly trained) one.
3. Partial derivatives of the error function $\partial E(w_{ij})/\partial w_{ij}$ tell us which direction we need to move in weight space to reduce the error.
4. The learning rate $\eta$ specifies the step sizes we take in weight space for each iteration of the weight update equation.
5. We keep stepping through weight space until the errors are 'small enough'.
6. If we choose neuron activation functions with derivatives that take on particularly simple forms, we can make the weight update computations very efficient.

These factors lead to powerful learning algorithms for training our neural networks:

# Training a Single Layer Feed-forward Network

Now we understand how gradient descent weight update rules can lead to minimisation of a neural network's output errors, it is straightforward to train any network:

1. Take the set of training patterns you wish the network to learn

   $\{in_i^p, out_j^p : i = 1 \ldots ninputs, j = 1 \ldots noutputs, p = 1 \ldots npatterns\}$

2. Set up your network with *ninputs* input units fully connected to *noutputs* output units via connections with weights $w_{ij}$

3. Generate random initial weights, e.g. from the range [–*smwt*, +*smwt*]

4. Select an appropriate error function $E(w_{ij})$ and learning rate $\eta$

5. Apply the weight update $\Delta w_{ij} = -\eta \, \partial E(w_{ij})/\partial w_{ij}$ to each weight $w_{ij}$ for each training pattern *p*. One set of updates of all the weights for all the training patterns is called one ***epoch*** of training.

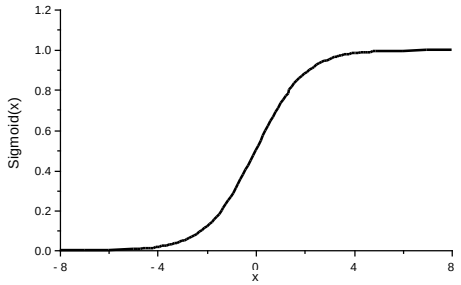
6. Repeat step 5 until the network error function is 'small enough'.

You thus end up with a trained neural network. But step 5 can still be difficult…

# The Derivative of a Sigmoid

We noted earlier that the Sigmoid is a smooth (i.e. differentiable) threshold function:

$$f(x) = \text{Sigmoid}(x) = \frac{1}{1+e^{-x}}$$

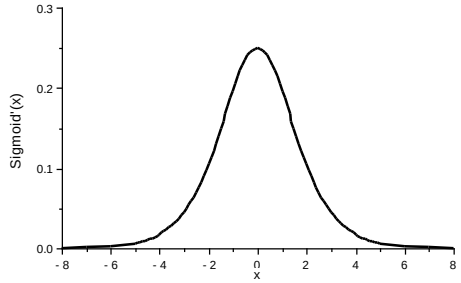



We can use the chain rule by putting $f(x) = g(h(x))$ with $g(h) = h^{-1}$ and $h(x) = 1 + e^{-x}$ so

$$\frac{\partial g(h)}{\partial h} = -\frac{1}{h^2} \quad \text{and} \quad \frac{\partial h(x)}{\partial x} = -e^{-x}$$

$$\frac{\partial f(x)}{\partial x} = -\frac{1}{(1+e^{-x})^2} \cdot (-e^{-x}) = \left(\frac{1}{1+e^{-x}}\right)\cdot\left(\frac{1+e^{-x}-1}{1+e^{-x}}\right)$$

$$f'(x) = \frac{\partial f(x)}{\partial x} = f(x).(1-f(x))$$




This simple relation will make our equations much easier and save a lot of computing time!

# The Generalised Delta Rule

We can avoid using tricks for deriving gradient descent learning rules, by making sure we use a differentiable activation function such as the Sigmoid. This is also more like the threshold function used in real brains, and has several other nice mathematical properties.

If we use the Sigmoid activation function, a single layer network has outputs given by

$$out_l = \text{Sigmoid}(\sum_i in_i w_{il})$$

and, due to the properties of the Sigmoid derivative, the general weight update equation

$$\Delta w_{kl} = \eta \sum_p (targ_l - out_l).f'(\sum_i in_i w_{il}).in_k$$

simplifies so that it only contains neuron activations and no derivatives:

$$\Delta w_{kl} = \eta \sum_p (targ_l - out_l).out_l.(1 - out_l).in_k$$

This is known as the ***Generalized Delta Rule*** for training sigmoidal networks.

# Practical Considerations for Gradient Descent Learning

From the above discussion, it is clear that there remain a number of important questions about training single layer neural networks that still need to be resolved:

1. Do we need to pre-process the training data? If so, how?
2. How do we choose the initial weights from which we start the training?
3. How do we choose an appropriate learning rate $\eta$?
4. Should we change the weights after each training pattern, or after the whole set?
5. Are some activation/transfer functions better than others?
6. How can we avoid flat spots in the error function?
7. How can we avoid local minima in the error function?
8. How do we know when we should stop the training?

We shall now consider each of these issues in turn.

# Pre-processing the Training Data

In principle, we can just use any raw input-output data to train our networks. However, in practice, it often helps the network to learn appropriately if we carry out some pre-processing of the training data before feeding it to the network.

We should make sure that the training data is representative – it should not contain too many examples of one type at the expense of another. On the other hand, if one class of pattern is easy to learn, having large numbers of patterns from that class in the training set will only slow down the over-all learning process.

If the training data is continuous, rather than binary, it is generally a good idea to re-scale the input values. Simply shifting the zero of the scale so that the mean value of each input is near zero, and normalising so that the standard deviation of the values for each input are roughly the same, can make a big difference. It will require more work, but de-correlating the inputs before normalising is often also worthwhile.

If we are using on-line training rather than batch training, we should usually make sure we shuffle the order of the training data each epoch.

# Choosing the Initial Weights

The gradient descent learning algorithm treats all the weights in the same way, so if we start them all off with the same values, all the hidden units will end up doing the same thing and the network will never learn properly.

For that reason, we generally start off all the weights with small random values. Usually we take them from a flat distribution around zero [–*smwt*, +*smwt*], or from a Gaussian distribution around zero with standard deviation *smwt*.

Choosing a good value of *smwt* can be difficult. Generally, it is a good idea to make it as large as you can without saturating any of the sigmoids.

We usually hope that the final network performance will be independent of the choice of initial weights, but we need to check this by training the network from a number of different random initial weight sets.

In networks with hidden layers, there is no real significance to the order in which we label the hidden neurons, so we can expect very different final sets of weights to emerge from the learning process for different choices of random initial weights.

# Choosing the Learning Rate

Choosing a good value for the learning rate $\eta$ is constrained by two opposing facts:

1. If $\eta$ is too small, it will take too long to get anywhere near the minimum of the error function.
2. If $\eta$ is too large, the weight updates will over-shoot the error minimum and the weights will oscillate, or even diverge.

Unfortunately, the optimal value is very problem and network dependent, so one cannot formulate reliable general prescriptions. Generally, one should try a range of different values (e.g. $\eta$ = 0.1, 0.01, 1.0, 0.0001) and use the results as a guide.

There is no necessity to keep the learning rate fixed throughout the learning process. Typical variable learning rates that prove advantageous are:

$$\eta(t) = \frac{\eta(1)}{t} \qquad\qquad \eta(t) = \frac{\eta(0)}{1 + t/\tau}$$

Similar age dependent learning rates are found to exist in human children.

# Batch Training *vs.* On-line Training

The gradient descent learning algorithm contains a sum over all training patterns *p*

$$\Delta w_{kl} = \eta \sum_{p} \left( targ_l - out_l \right) . f'\left( \sum_{i} in_i w_{il} \right) . in_k$$

When we add up the weight changes for all the training patterns like this, and apply them in one go, it is called ***Batch Training***.

A natural alternative is to update all the weights immediately after processing each training pattern. This is called ***On-line Training*** (or ***Sequential Training***).

On-line learning does not perform true gradient descent, and the individual weight changes can be rather erratic. Normally a much lower learning rate $\eta$ will be necessary than for batch learning. However, because each weight now has *npatterns* updates per epoch, rather than just one, overall the learning is often much quicker. This is particularly true if there is a lot of redundancy in the training data, i.e. many training patterns containing similar information.

# Choosing the Transfer Function

We have already seen that having a differentiable transfer/activation function is important for the gradient descent algorithm to work. We have also seen that, in terms of computational effeciency, the standard sigmoid (i.e. logistic function) is a particularly convenient replacement for the step function of the Simple Perceptron.

The logistic function ranges from 0 to 1. There is some evidence that an anti-symmetric transfer function, i.e. one that satisfies $f(-x) = -f(x)$, enables the gradient descent algorithm to learn faster. To do this we must use targets of +1 and –1 rather than 0 and 1. A convenient alternative to the logistic function is then the hyperbolic tangent

$$f(x) = \tanh(x) \qquad f(-x) = -f(x) \qquad f'(x) = 1 - f(x)^2$$

which, like the logistic function, has a particularly simple derivative.

When the outputs are required to be non-binary, i.e. continuous real values, having sigmoidal transfer functions no longer makes sense. In these cases, a simple linear transfer function $f(x) = x$ is appropriate.

# Classification Outputs as Probabilities

Another powerful feature of neural network classification systems is that non-binary outputs can be interpreted as the probabilities of the corresponding classifications. For example, an output of 0.9 on a unit corresponding to a particular class would indicate a 90% chance that the input data represents a member of that class.

The mathematics is rather complex, but one can show that for two classes represented as activations of 0 and 1 on a single output unit, the activation function that allows us to do this is none other than our ***Sigmoid*** activation function.

If we have more than two classes, and use one output unit for each class, we should employ a generalization of the Sigmoid known as the ***Softmax*** activation function:

$$out_j = e^{-\sum_i in_i w_{ij}} \Big/ \sum_k e^{-\sum_n in_n w_{nk}}$$

In either case, we use a ***Cross Entropy*** error measure rather than Sum Squared Error.

# Flat Spots in the Error Function

The gradient descent weight changes depend on the gradient of the error function. Consequently, if the error function has flat spots, the learning algorithm can take a long time to pass through them.

A particular problem with the sigmoidal transfer functions is that the derivative tends to zero as it saturates (i.e. gets towards 0 or 1). This means that if the outputs are totally wrong (i.e. 0 instead of 1, or 1 instead of 0), the weight updates are very small and the learning algorithm cannot easily correct itself. There are two simple solutions:

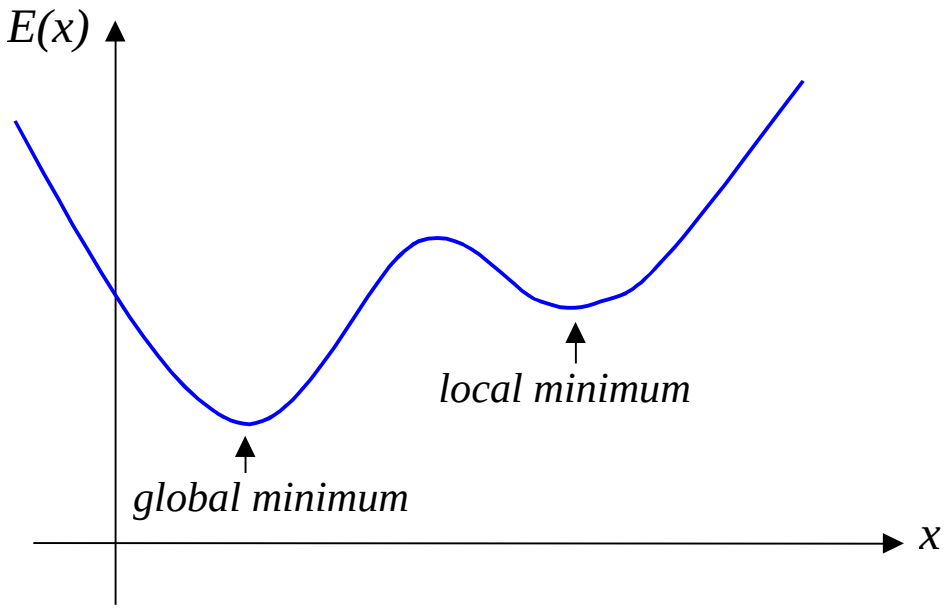
**Target Off-sets** Use targets of 0.1 and 0.9 (say) instead of 0 and 1. The sigmoids will no longer saturate and the learning will no longer get stuck.

**Sigmoid Prime Off-set** Add a small off-set (of 0.1 say) to the sigmoid prime (i.e. the sigmoid derivative) so that it is no longer zero when the sigmoids saturate.

We can now see why we should keep the initial network weights small enough that the sigmoids are not saturated before training. Off-setting the targets also has the effect of stopping the network weights growing too large.

# Local Minima

Error functions can quite easily have more than one minimum:




If we start off in the vicinity of the local minimum, we may end up at the local minimum rather than the global minimum. Starting with a range of different initial weight sets increases our chances of finding the global minimum. Any variation from true gradient descent will also increase our chances of stepping into the deeper valley.

# When to Stop Training

The Sigmoid($x$) function only takes on its extreme values of 0 and 1 at $x = \pm\infty$. In effect, this means that the network can only achieve its binary targets when at least some of its weights reach $\pm\infty$. So, given finite gradient descent step sizes, our networks will never reach their binary targets.

Even if we off-set the targets (to 0.1 and 0.9 say) we will generally require an infinite number of increasingly small gradient descent steps to achieve those targets.

Clearly, if the training algorithm can never actually reach the minimum, we have to stop the training process when it is 'near enough'. What constitutes 'near enough' depends on the problem. If we have binary targets, it might be enough that all outputs are within 0.1 (say) of their targets. Or, it might be easier to stop the training when the sum squared error function becomes less than a particular small value (0.2 say).

We shall see later that, when we have noisy training data, the training set error and the generalization error are related, and an appropriate stopping criteria will emerge in order to optimize the network's generalization ability.

# Overview and Reading

1. We started by formulating the steps involved in training single layer neural networks using gradient descent algorithms.
2. We then saw how using sigmoidal activation functions can lead to the efficient Generalized Delta Rule for neural network training.
3. Finally, we systematically considered the main practical issues that are involved in successfully training general single layer feed-forward networks using gradient descent algorithms.

## Reading

1. Gurney: Sections 5.2, 5.3, 5.4, 5.5
2. Haykin: Sections 3.5, 3.7, 4.6
3. Callan: Sections 2.4, 6.4
4. Beale & Jackson: Sections 4.4, 4.7
5. Bishop: Sections 3.1, 6.9