The Generalized Delta Rule and Practical Considerations

Neural Computation : Lecture 6

© John A. Bullinaria, 2015

1. Gradient Descent and the Generalized Delta Rule
2. Different Error Functions for Classification and Regression
3. Practical Considerations for Gradient Descent Learning
   (*1*) Pre-processing the Training Data
   (*2*) Choosing the Initial Weights
   (*3*) Choosing the Learning Rate
   (*4*) On-line Training *vs*. Batch Training
   (5) Avoiding Local Minima
   (6) Avoiding Flat Spots in the Error Function
   (*7*) Deciding When to Stop the Training

Gradient Descent Learning

It is worth summarising the key 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 start by defining an appropriate Error or Cost Function $E(w_{ij})$ that "measures" how far the current network is from the desired (correctly trained) one.
3. Gradients given by partial derivatives of the error function $\partial E(w_{ij})/\partial w_{ij}$ then tell us which direction we need to move in weight space to reduce the error.
4. The gradients are multiplied by a learning rate η that 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 the neuron activation functions have derivatives that take on particularly simple forms, that can make the weight update computations very efficient.

These factors lead to powerful learning algorithms for training neural networks.

The Generalised Delta Rule

We have derived the basic gradient descent learning algorithm for single layer networks with Sum Squared Error (SSE) cost function and general activation function $f(x)$:

$$\Delta w_{kl} = \eta \sum_{p} \left(targ_l - out_l \right) . f'\left(\sum_{n} in_n w_{nl} \right) . in_k$$

For a simple linear activation function $f(x) = x$ this gives the standard delta rule:

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

We now move on to consider generalized versions of this that correspond to other output activation functions and other cost functions.

Clearly, having differentiable error functions and activation functions is crucial for the gradient descent algorithm to work. We shall now see how different types of problem (e.g., classification versus regression) naturally have different choices for these functions, that could potentially result in different weight update rules.

Error Function for Classification Problems

The most appropriate cost or error function $E(w_{ij})$ will depend on the type of problem being studied. The general idea is to find the neural network that maximizes the likelihood $\mathcal{L}$ of observing the training data. That is equivalent to minimizing the negative log of the likelihood, i.e. the error/cost function $E = -\ln \mathcal{L}$.

Classification problems involve determining the most likely class that an input pattern belongs to. Or, more formally, modelling the posterior probabilities of class membership conditioned on the input variables.

The obvious way to do this is to have one output unit for each class, and for each input pattern have a target output of 1 for the output unit corresponding to that class, and 0 for all the other output units. If there are only two classes, one output unit is sufficient, with a target of 1 for one class and 0 for the other.

But what should one use for the activation and cost functions?

Classification Outputs as Probabilities

A powerful feature of using neural networks for classification problems is that it is possible for non-binary outputs to be interpreted as 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 one to do this is the ***Sigmoid*** (a.k.a. logistic function) activation function we have seen before.

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

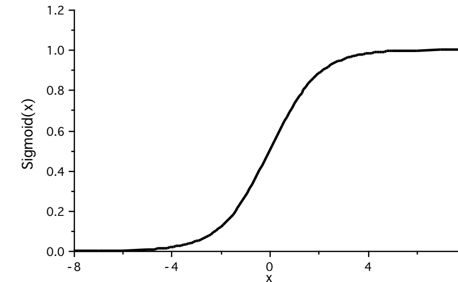
$$out_j = e^{\sum_i in_i w_{ij}} \Big/ \sum_k e^{\sum_n in_n w_{nk}}$$

These are analogous to the smooth threshold functions found in real brains.

The Derivative of a Sigmoid

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

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

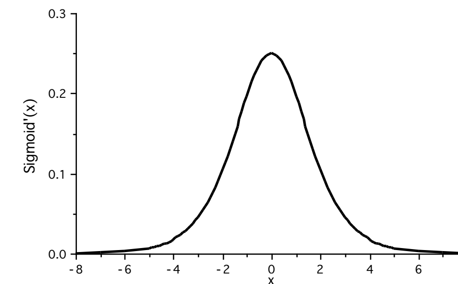


We can use the chain rule by putting $f(x) = g(h(x))$ with $g(h) = h^{-1}$ and $h(x) = 1 + e^{-x}$ so

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

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

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



This relation makes the update equations much simpler and saves a lot of computing time!

Cross Entropy Cost Function for Two Classes

If the output *out* of a network represents the probability of a particular class, and *targ* is the binary target output, the probability of observing the whole training data set is

$$\prod_p out^{targ} (1 - out)^{1 - targ}$$

The neural network model aims to maximise this likelihood, which is equivalent of minimizing the negative logarithm of it, so we have the natural cost function

$$E_{ce} = -\sum_p \left[targ.\log(out) + (1 - targ).\log(1 - out) \right]$$

This is called the ***Cross Entropy (CE)*** error function. Computing the gradient descent partial derivatives $\partial E(w_{ij})/\partial w$ here has the sigmoid derivative term cancel out leaving

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

which is actually easier to compute than the *SSE* equivalent.

Cross Entropy Cost Function for Multiple Classes

If we have network output out_j representing the probability of class j, and $targ_j$ is the binary target output, the probability of observing the whole training data set is now

$$\prod_p \prod_j out_j^{\,targ_j}$$

Then minimizing the negative logarithm of this likelihood leads to the cost function

$$E_{ce} = -\sum_p \sum_j targ_j . \log\left(out_j\right)$$

which is the multiple class ***Cross Entropy (CE)*** error function. Computing the gradient descent partial derivatives $\partial E(w_{ij})/\partial w$ here with the softmax activation function again has the output activation function derivative term cancelling out leaving

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

which is exactly the same weight update equation as for the two class case.

SSE Cost Function for Regression Problems

Regression or Function Approximation problems involve approximating an underlying function from a set of noisy data. Or, more formally, modelling the distribution of the output variables conditioned on the input variables.

In this case we will have continuous real valued (i.e. non-binary) target outputs, and having sigmoidal output activation functions no longer makes sense. Sometimes there is a natural problem dependent activation function that is appropriate to use, but generally it is most appropriate to use the simple ***linear activation function*** $f(x) = x$.

The mathematics in this case is complicated, but one can again derive the appropriate cost function from the principle of maximum likelihood (here assuming Gaussian distributed target data) and find it to be the simple ***Sum Squared Error (SSE)*** cost function that we have already discussed. We thus end up with the same weight update equation for regression, two-class classification and multiple-class classification if we use the right (different) cost and activation functions for each case.

Practical Considerations for Gradient Descent Learning

The general idea is straightforward, but 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 the training is started?

3. How do we choose an appropriate learning rate η?

4. Should we change the weights after each training pattern, or after the whole set?

5. How can we avoid local minima in the error function?

6. How can we avoid flat spots in the error function?

7. How do we know when we should stop the training?

We shall now consider each of these practical issues in turn.

Pre-processing the Training Data

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

First, 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 particularly 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. Shifting the zero of the scale so that the minimum or mean value of each input is near zero, and normalising so that the range or 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 on-line training is being used rather than batch training, it is sensible to make sure that the order of the training data is shuffled each epoch.

Choosing the Initial Weights

The gradient descent learning algorithm treats all the weights in the same way, so if they are all started off with the same value, all the hidden units will end up doing the same thing, and the network will never learn properly.

For that reason, one generally starts off all the weights with small random values. Usually they are taken from a uniform distribution around zero [–*smwt*, +*smwt*], or from a Gaussian distribution around zero with standard deviation *smwt*.

The optimal value of the parameter *smwt* is problem dependent, and needs to be found empirically. It is important that it is not so large that the sigmoids become saturated. For sufficiently small values, the final network performance should not depend much on the precise choice of initial weights, but this is something that needs to be checked by training the network using a number of different random initial weight sets.

In networks with hidden layers, there is no significance to the ordering of the hidden neurons in each layer, so very different final sets of weights can be expected 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 η is constrained by two opposing factors:

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

Unfortunately, the optimal value depends on the problem and network architecture, 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. In fact, epoch dependent learning rates can prove advantageous, for example:

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

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 the training patterns p

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

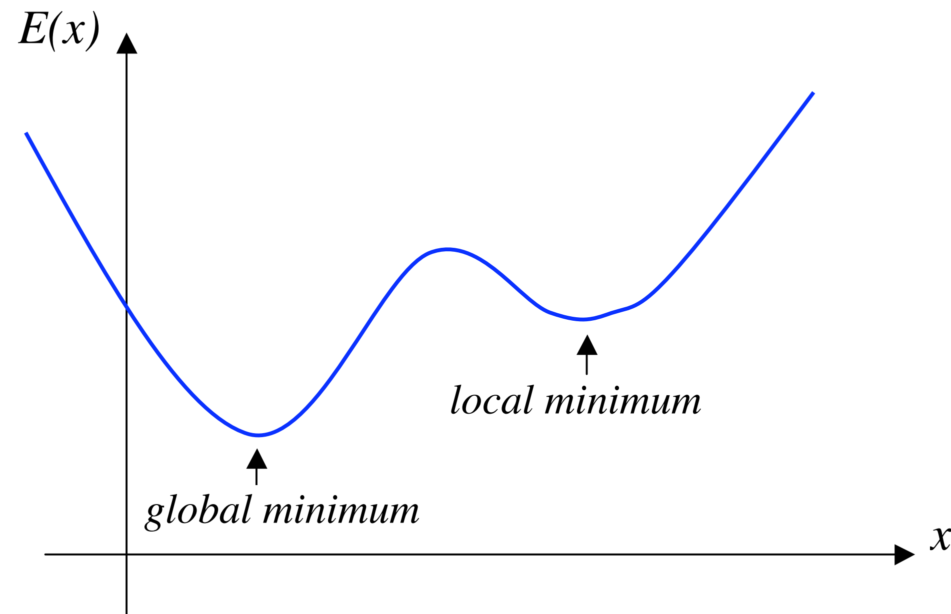
When we add up the weight changes for all the training patterns like this, and apply them in one go, it is called ***Batch Training***.

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 η 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.

Local Minima

Error functions can quite easily have more than one minimum:



If one starts off in the vicinity of the local minimum, learning is likely to end at the local minimum rather than the global minimum. Starting with a range of different initial weight sets increases the chances of finding the global minimum. Variations from true gradient descent will also increase the chances of stepping into the deeper valley.

Flat Spots in the Error Function

The gradient descent weight changes are proportional to 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.

Probably the most well-known problem of this type arises when the sigmoid activation function is used with the SSE cost function (which is not usually sensible anyway). In this case, the sigmoid derivative tends to zero as it saturates (i.e. gets near 0 or 1) in

$$\Delta w_{kl} = \eta \sum_{p} \left(targ_l - out_l \right).out_l.(1 - out_l).in_k$$

This means that if the outputs somehow become close to being 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 the error.

This is the reason why one should keep the initial network weights small enough that the sigmoids are not saturated before training begins.

Avoiding Sigmoid Flat Spots

There are three common solutions that are used to avoid sigmoid-based flat spots:

Target Off-sets Use targets of 0.1 and 0.9 (say) instead of 0 and 1. Learning will no longer saturate the sigmoids, and so it will not become stuck due to that. Off-setting the targets also has the effect of stopping the network weights growing too large.

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.

Cross Entropy Cost Function If the SSE cost function is replaced by the CE cost function, the sigmoid derivative term cancels out of the weight update equation, so the fact that it goes towards zero doesn't matter.

In fact, the Sigmoid Prime Offset approach is equivalent to adding a small amount of Cross-Entropy cost function to the SSE cost function.

Using Evolutionary Computation as an optimization process has demonstrated that the full Cross Entropy approach works best.

When Does the Learning Algorithm Finish?

With step function outputs and binary targets, the network learning (e.g., using the Perceptron Learning Rule) stops when the target outputs are reached.

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

Even if the binary targets are off-set (e.g., to 0.1 and 0.9) an infinite number of increasingly small gradient descent steps will be required to achieve those targets.

If we have a regression problem with real valued targets and linear output units, this will again generally require an infinite number of increasingly small gradient descent steps to achieve those targets.

Clearly, if the training algorithms can never actually reach the error minimum, we have to stop the training process when it is “near enough”.

Deciding When to Stop the Training

What constitutes “near enough” depends on the problem. If the targets are binary, it might be enough that all outputs are within a “tolerance” (0.1 say) of their targets.

Or, it might be easier to stop the training when the total SSE or CE cost function over all training patterns becomes less than a particular small value (0.2 say). Clearly, the appropriate small value will depend on the number of training patterns, how noisy they are, and (for regression problems) the magnitudes of the target outputs.

We shall see later that a network’s generalization error will often depend on when the training is stopped. In particular, when the training data is noisy, training the network for too long can make the generalization performance worse. What needs to be done in that case is to estimate how well the network is generalizing at each stage of training, and stop the training when the estimated error reaches its minimum.

This idea of “stopping early” to improve generalization is an important topic for artificial neural network learning, and we shall return to look at it in more detail later.

Overview and Reading

1. We started by considering why sigmoid/softmax output activations with a Cross Entropy cost function was appropriate for classification problems.
2. Then we considered why linear output activations with a Sum Squared Error cost function was appropriate for regression problems.
3. Finally, we systematically considered the main practical issues that are generally involved in successfully training single layer feed-forward networks using gradient descent learning algorithms.

Reading

1. Bishop: Sections 3.1, 6.1, 6.7, 6.8, 6.9
2. Haykin-1999: Sections 3.5, 3.7, 4.4, 4.6
3. Gurney: Sections 5.2, 5.3, 5.4, 5.5
4. Callan: Sections 2.4, 6.4
5. Beale & Jackson: Sections 4.4, 4.7