MLP & Backpropagation Issues

Slides thanks to J. Bullinaria

Considerations

- Network architecture
  - Typically feedforward, however you may also use local receptive fields for hidden nodes; and recursive architectures for sequence learning
- Number of input, hidden, output nodes
  - Number of hidden nodes is quite important, others determined by problem setup
- Activation functions
  - Careful: regression requires linear activation on the output
  - For others, sigmoid or hyperbolic tangent is a good choice
- Learning rate

Performance Surface

To provide motivation for some of the practical issues

Local Minima of the Performance Criteria

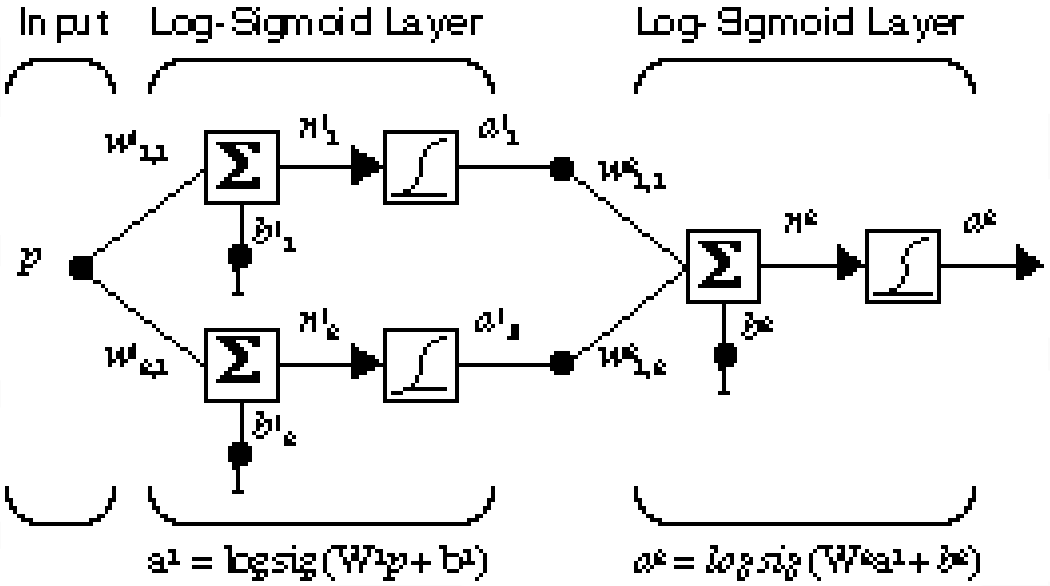
- The performance surface is a very high-dimensional (W) space full of local minima.

- Your best bet using gradient descent is to locate one of the local minima.
  - Start the training from different random locations (we will later see how we can make use of several thus trained networks)
  - You may also use simulated annealing or genetic algorithms to improve the search in the weight space.

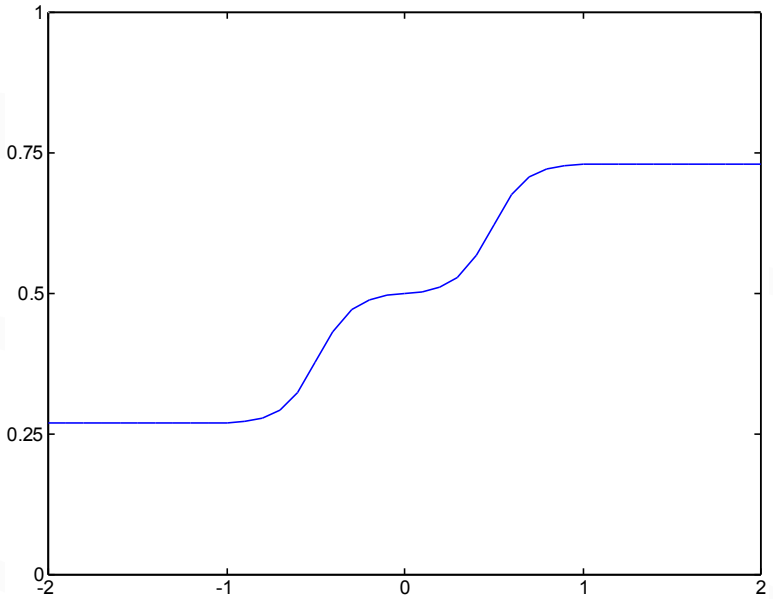
- See how complex the performance surface look like even with a few weights in the next few slides.space

Performance Surface Example

Network Architecture



Nominal Function



Parameter Values

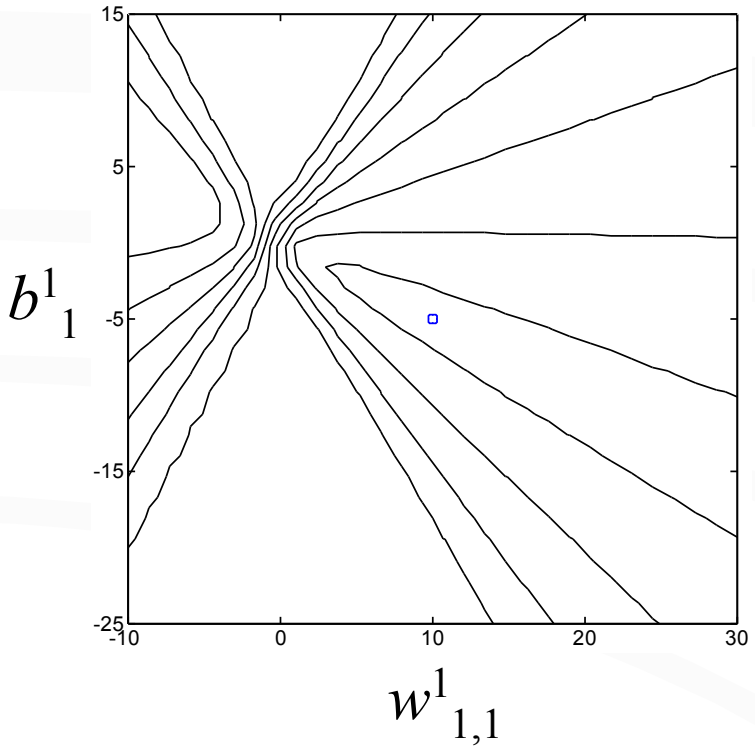
$$w^1_{1,1} = 10 \qquad b^1_1 = -5$$

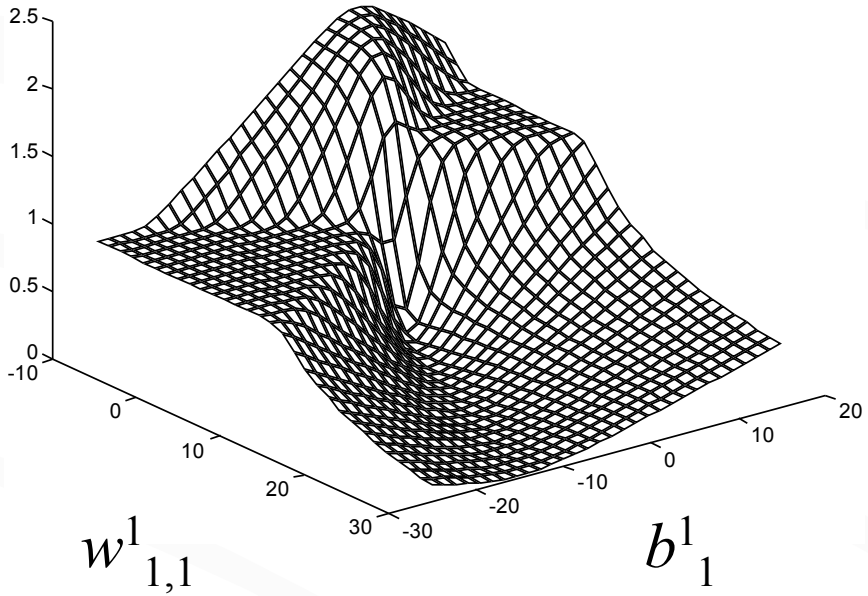
$$w^1_{2,1} = 10 \qquad b^1_2 = 5$$

$$w^2_{1,1} = 1 \qquad w^2_{1,2} = 1 \qquad b^2 = -1$$

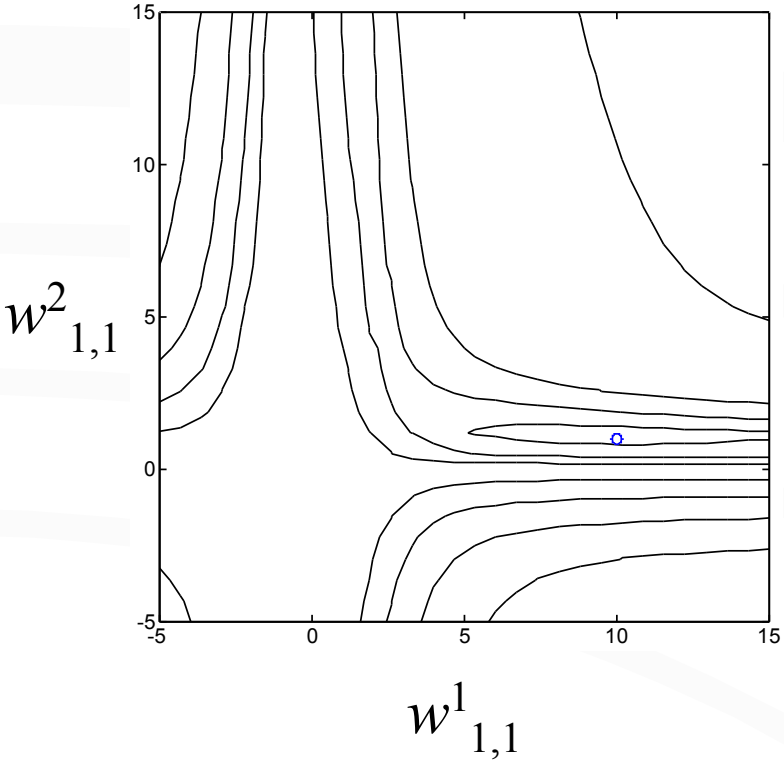
Layer numbers are shown as superscripts

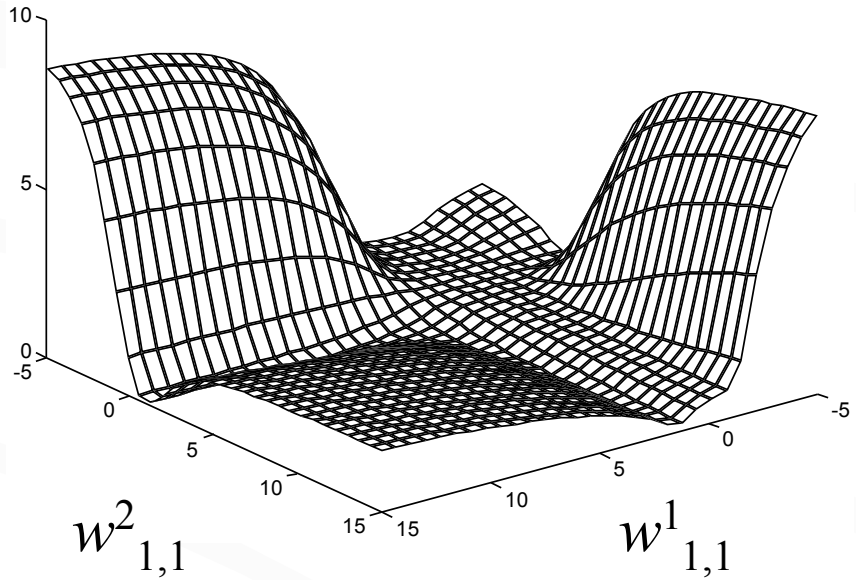
Squared Error vs. $w^1_{1,1}$ and b^1_1



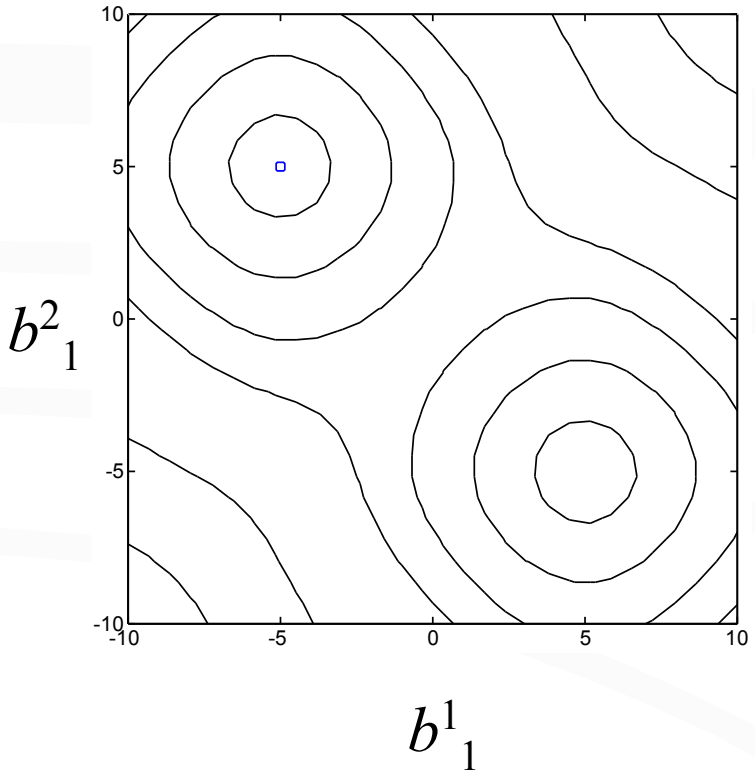


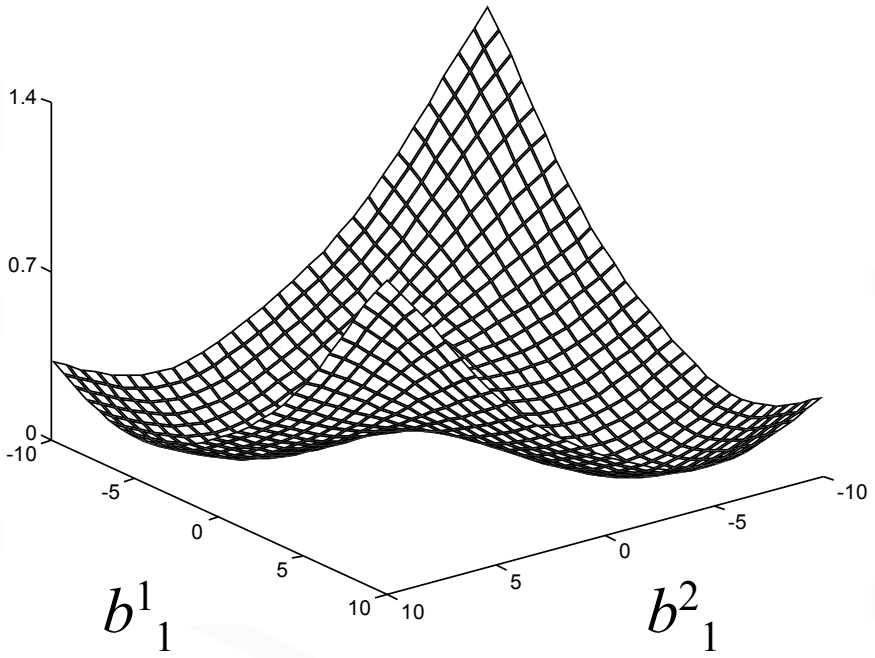
Squared Error vs. $w^1_{1,1}$ and $w^2_{1,1}$





Squared Error vs. b^1_1 and b^1_2





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 (trained) one.
3. Partial derivatives of the error function $\partial E(w_{ij})/\partial w_{ij}$ tell us the direction we need to move in weight space to minimise the error.
4. The learning rate η determines 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. The derivatives for sigmoidal activation functions take on particularly simple forms that make the weight update computations very efficient.

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

Training a Two-Layer Feed-forward Network

This is practically identical to training a single layer 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 *nhidden* hidde units via connections with weights $w_{ij}^{(1)}$, which in turn are fully connected t *noutputs* output units via connections with weights $w_{jk}^{(2)}$.

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

4. Select an appropriate error function $E(w_{jk}^{(n)})$ and learning rate η.

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

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

The extension to networks with more hidden layers should be obvious.

Practical Considerations for Gradient Descent Learning

From the above it is clear that there remain a number of issues about training single layer neural networks that need to be resolved:

1. Do we need to pre-process the training data? How?
2. How do we choose the initial weights from which we start the training?
3. How do we choose the learning rate?
4. Do we change the weights after each training pattern, or after the whole set?
5. Are some activation functions better than others?
6. How do we avoid flat spots in the error function?
7. How do we avoid local minima in the error function?
8. When do we stop training?

We shall now consider each of these issues in turn.

Practical Considerations for Back-Propagation Learning

Most of the practical considerations necessary for general Back-Propagation learning were already been covered when we talked about training single layer Perceptrons:

1. Do we need to pre-process the training data?
2. How do we choose the initial weights from which we start the training?
3. How do we choose the learning rate?
4. Do we change the weights after each training pattern or after the whole set?
5. Are some activation functions better than others?
6. How do we avoid flat spots in the error function?
7. How do we avoid local minima in the error function?
8. When do we stop training?

However, there are two important issues that were not covered before:

1. How may hidden units do we need?
2. Should we have different learning rates for the different layers?

Pre-processing the Training Data

In principle, we can just use any raw 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.

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.

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.

Batch Training vs. On-line Training

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

$$\Delta w_{kl} = \eta \sum_{p} (targ_l - out_l).f'(\sum_{i} prev_i w_{il}).prev_k$$

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***.

An alternative is to update all the weights 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.

Choosing the Initial Weights

The gradient descent learning algorithm treats all the weights the same, and 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*.

Since there is no real significance to the order in which we label the weights in each hidden layer of the network, we can expect very different final sets of weights to emerge from the learning process depending on the precise choice of random initial weights.

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 initial weight sets.

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.

Choosing the Learning Rate

Choosing a good value for the learning rate η is constrained by two opposing facts:

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

There is no necessity to keep the learning rate fixed throughout the learning process. Common variable learning rates are:

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

Similar reductions in learning rate with age are found in human children.

Different Learning Rates for Different Layers?

A network as a whole will usually learn most efficiently if all its neurons are learning at roughly the same speed. So maybe different parts of the network should have different learning rates η. There are a number of factors that may affect the choices:

1. The later network layers (nearer the outputs) will tend to have larger local gradients (deltas) than the earlier layers (nearer the inputs).
2. The activations of units with many connections feeding into or out of them tend to change faster than units with fewer connections,
3. Activations required for linear units will be different for sigmoidal units
4. There is also empirical evidence that it helps to have different learning rates η for the thresholds/biases compared with the real connection weights.

In practice, it is often quicker to just use the same rates η for all the weights and thresholds rather than spend time trying to work out appropriate differences. A very powerful approach is to use evolutionary strategies to determine good learning rates.

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 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, having a sigmoidal transfer functions no longer makes sense. In these cases a simple linear transfer function $f(x) = x$ is appropriate.

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:

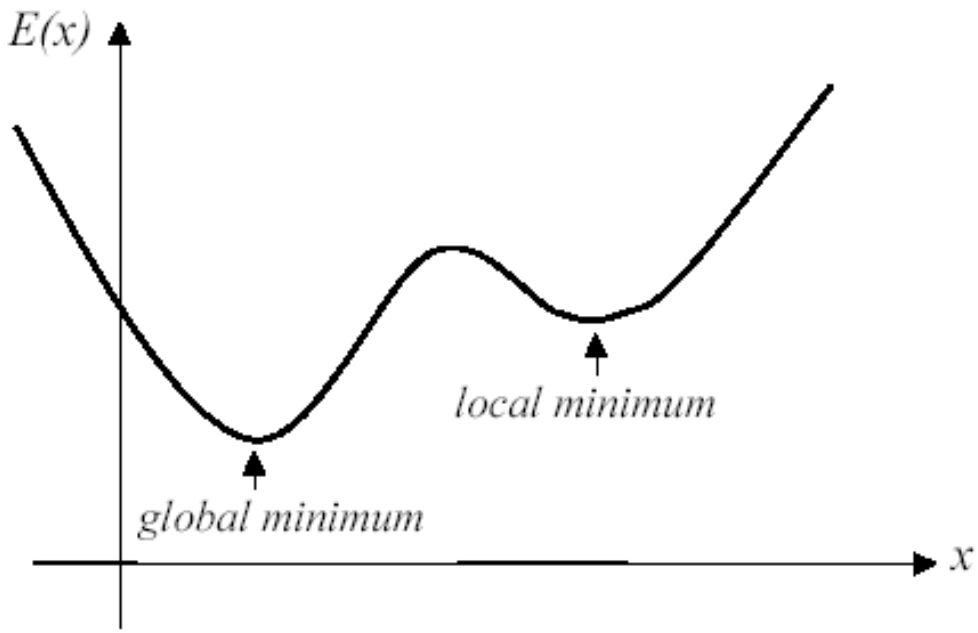
Target Off-sets Use targets of 0.1 and 0.9 (say) instead of 0 and 1. The sigmoids will no longer saturate and the learning will no longer get stuck.

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.

Stopping 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. 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 generalisation error are related, and that appropriate stopping criteria will emerge in order to improve the network's generalisation.

How Many Hidden Units?

The best number of hidden units depends in a complex way on many factors, including:

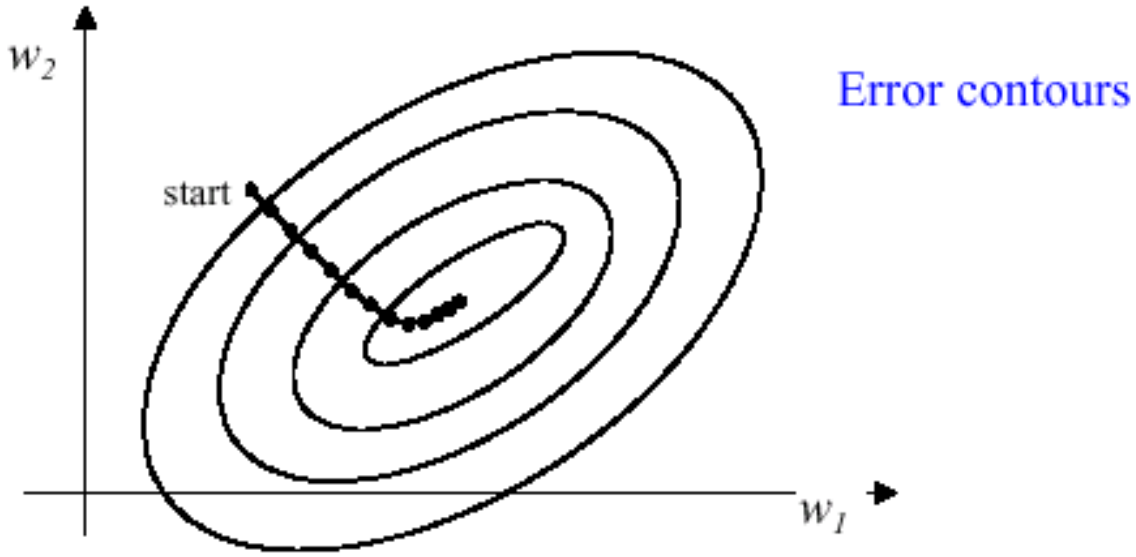
1. The number of training patterns
2. The numbers of input and output units
3. The amount of noise in the training data
4. The complexity of the function or classification to be learned
5. The type of hidden unit activation function
6. The training algorithm

Too few hidden units will generally leave high training and generalisation errors due to under-fitting. Too many hidden units will result in low training errors, but will make the training unnecessarily slow, and will result in poor generalisation unless some other technique (such as *regularisation*) is used to prevent over-fitting.

Virtually all "rules of thumb" you will hear about are actually nonsense. A sensible strategy is to try a range of numbers of hidden units and see which works best.

Visualising Learning

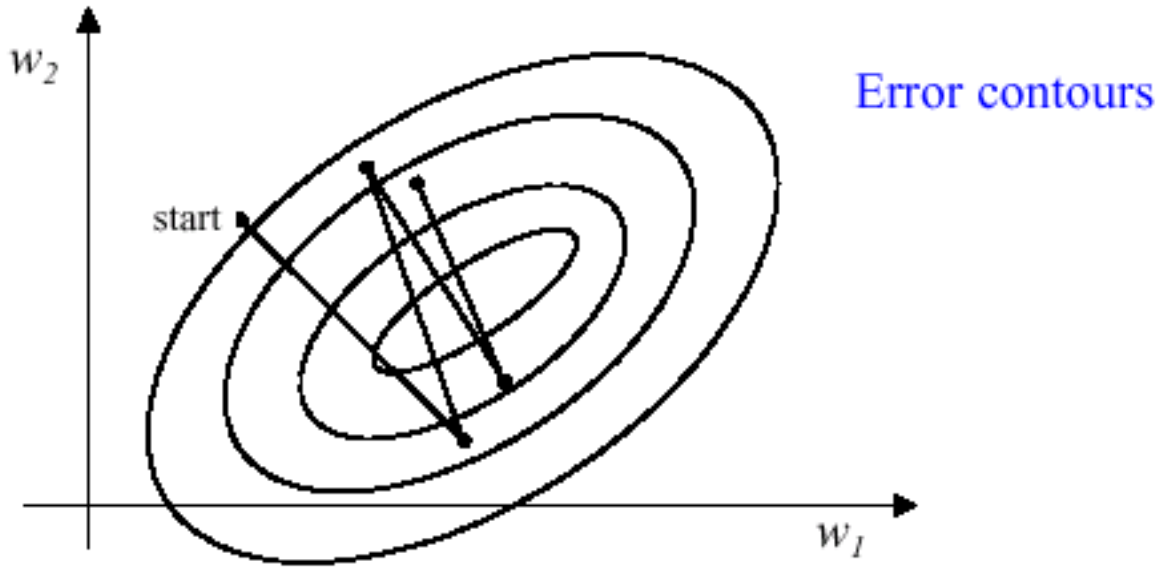
Visualising neural network learning is difficult because there are so many weights being updated at once. However, we can plot error function contours for pairs of weights to get some idea of what is happening. The weight update equations will produce a series of steps in weight space from the starting position to the error minimum:



True gradient descent produces a smooth curve perpendicular to the contours. Weight updates with a small step size η will result in the good approximation to it shown.

Step Size Too Large

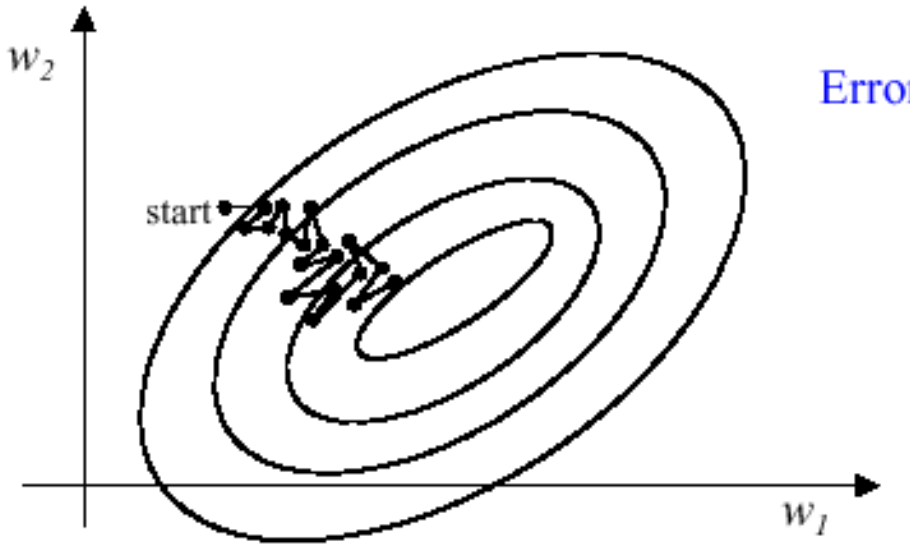
If the our learning rate step size is set too large, the approximation to true gradient descent will be poor, and we will end up with overshoots, or even divergence:



In practice we don't need to plot error contours to see if this is happening. Certainly if the error function is fluctuating, we should reduce the step size. It is also worth checking individual weights and output activations as well. On the other hand, if everything is smooth, it is worth trying to increase η and seeing if it stays smooth.

On-line Learning

If we update the weights after each training pattern, rather than adding up the weight changes for all the patterns before applying them, the learning algorithm is no longer true gradient descent, and the weight changes will not be perpendicular to the contours:



If we keep the step sizes small enough, the erratic behaviour of the weight updates will not be too much of a problem, and the increased number of weight changes will still get us to the minimum quicker than true gradient descent (i.e. batch learning).

Alternatives to Gradient Descent

SUMMARY

There are alternatives to standard backpropagation, intended to deal with speeding up its convergence.

These either choose a different search direction (p) or a different step size (α).

In this course, we will cover updates to standard backpropagation as an overview, namely momentum and variable rate learning, skipping the other alternatives (those that do not follow steepest descent, such as conjugate gradient method).

- Remember that you are never responsible of the HİDDEN slides (that do not show in show mode but are visible when you step through the slides!)

- Variations of Backpropagation
  - Momentum: Adds a momentum term to effectively increase the step size when successive updates are in the same direction.
  - Adaptive Learning Rate: Tries to increase the step size and if the effect is bad (causes oscillations as evidenced by a decrease in performance)
- Conjugate Gradient
- Levenberg-Marquardt:
  - These two methods choses the next search direction so as to speed up convergence. **You should use them instead of the basic backpropagation.**
- Line search: ...

Motivation for momentum (Bishop 7.5)

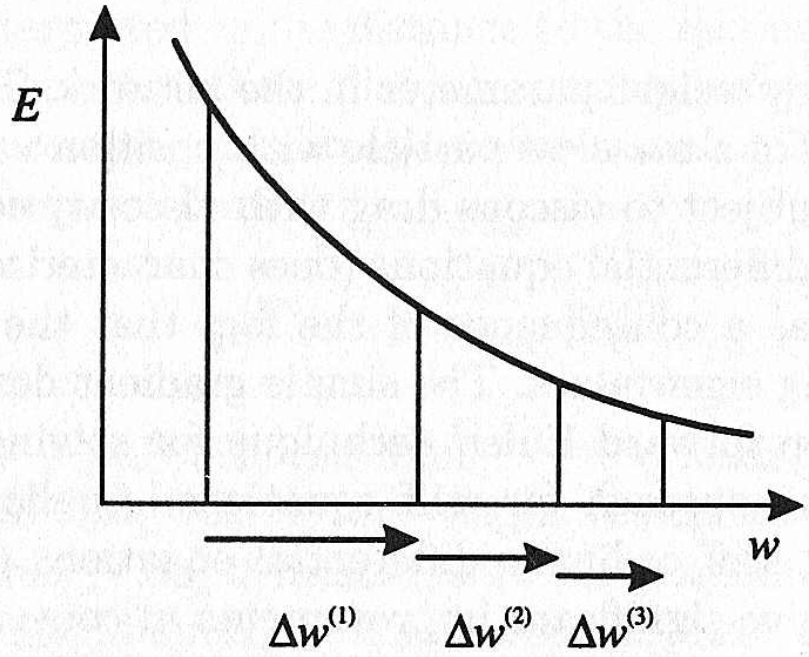


Figure 7.5. With a fixed learning rate parameter, gradient descent down a surface with low curvature leads to successively smaller steps (linear convergence). In such a situation, the effect of a momentum term is similar to an increase in the effective learning rate parameter.

Effect of momentum

$$\Delta w_{ij}(n) = \eta \nabla E/dw_{ij}(n) + \alpha \Delta w_{ij}(n-1)$$

If same sign in consecutive iterations => magnitude grows
If opposite sign in consecutive iterations => magnitude shrinks

$$\Delta w_{ij}(n) = \eta \sum_{t=0}^{n} \alpha^{n-t} \nabla E/dw_{ij}(t)$$

- For $\Delta w_{ij}(n)$ not to diverge, α must be < 1.
- Effectively adds inertia to the motion through the weight space and smoothes out the oscillations
- The smaller the η, the smoother the trajectory

Effect of momentum

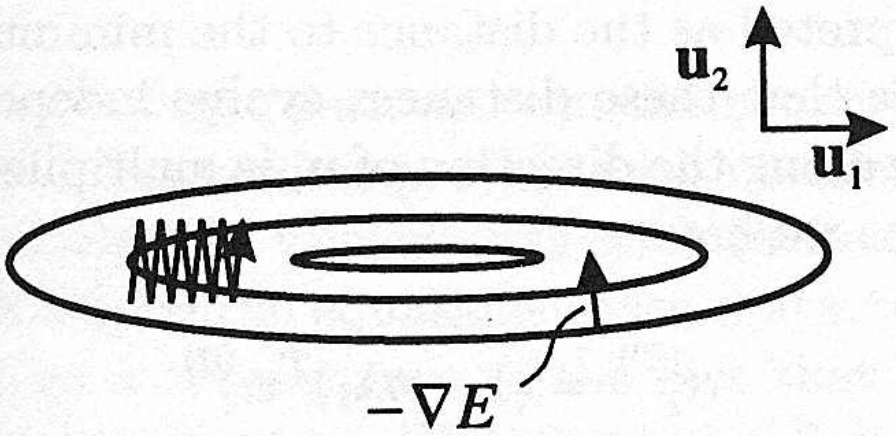
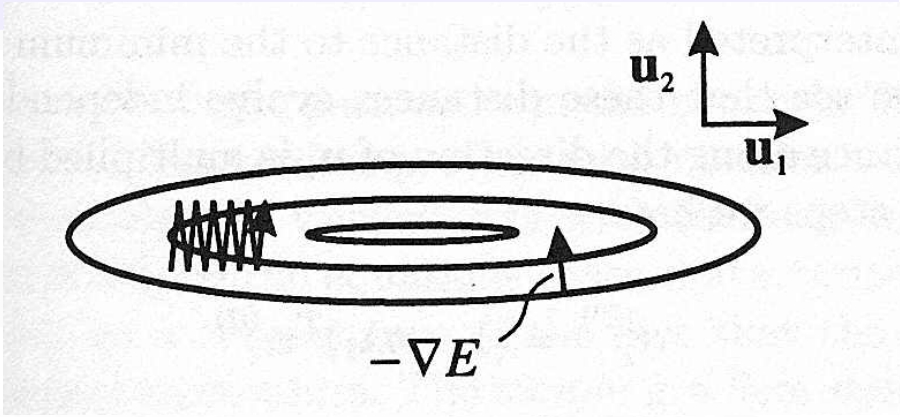


Figure 7.4. Schematic illustration of fixed-step gradient descent for an error function which has substantially different curvatures along different directions. Ellipses depict contours of constant E, so that the error surface has the form of a long valley. The vectors $\mathbf{u}_1$ and $\mathbf{u}_2$ represent the eigenvectors of the Hessian matrix. Note that, for most points in weight space, the local negative gradient vector $-\nabla E$ does not point towards the minimum of the error function. Successive steps of gradient descent can oscillate across the valley, with very slow progress along the valley towards the minimum.

Effect of momentum (Bishop 7.7)



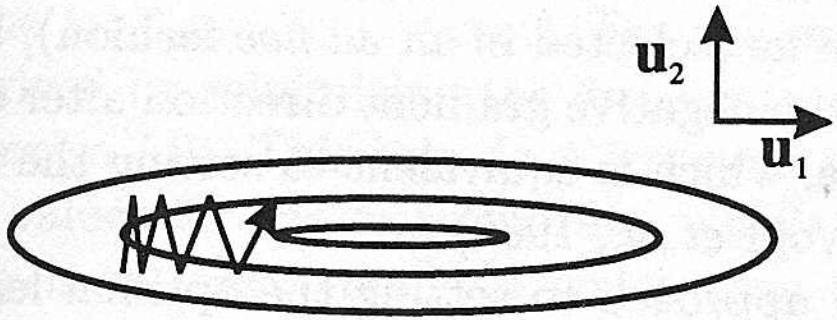
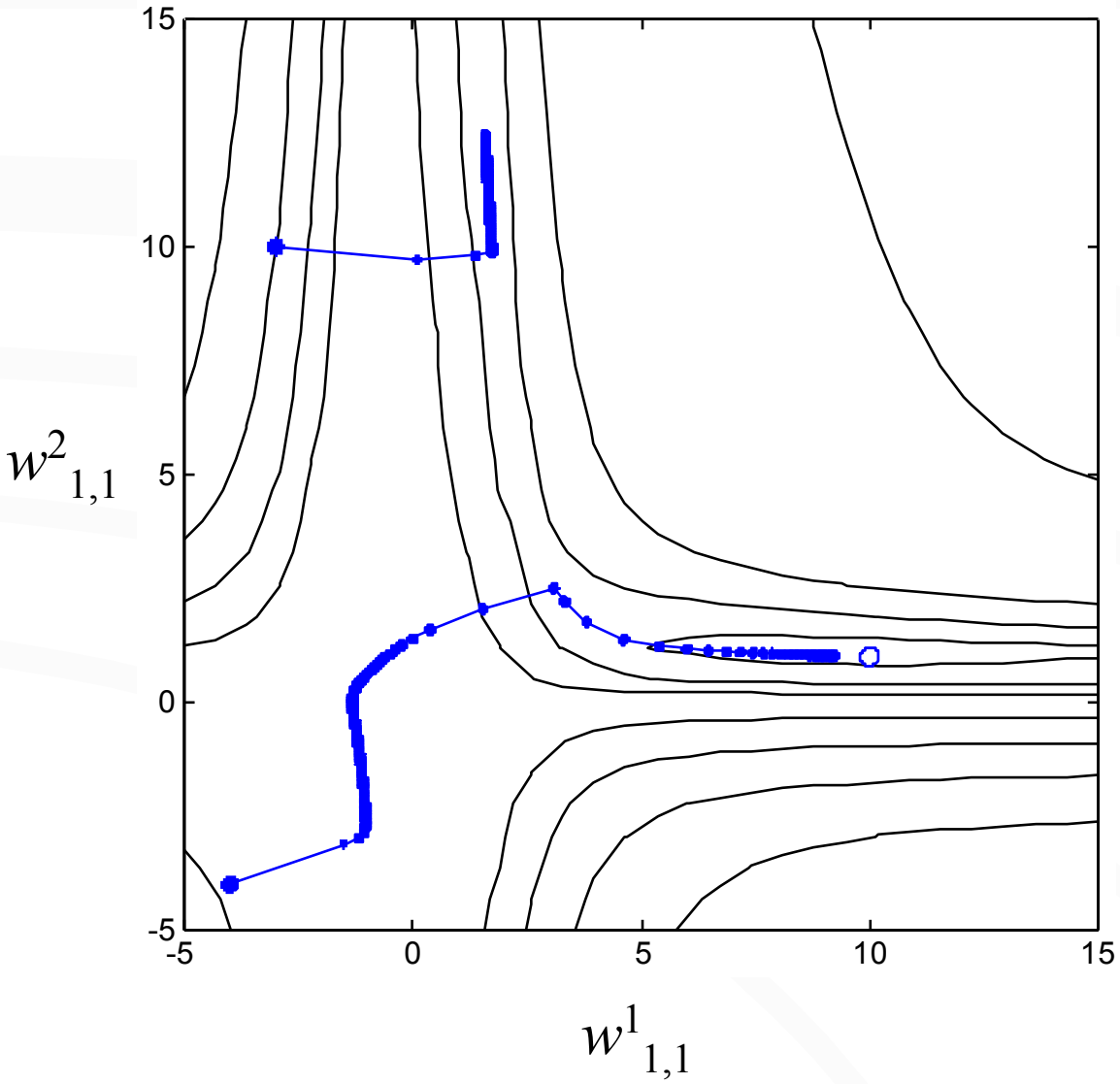
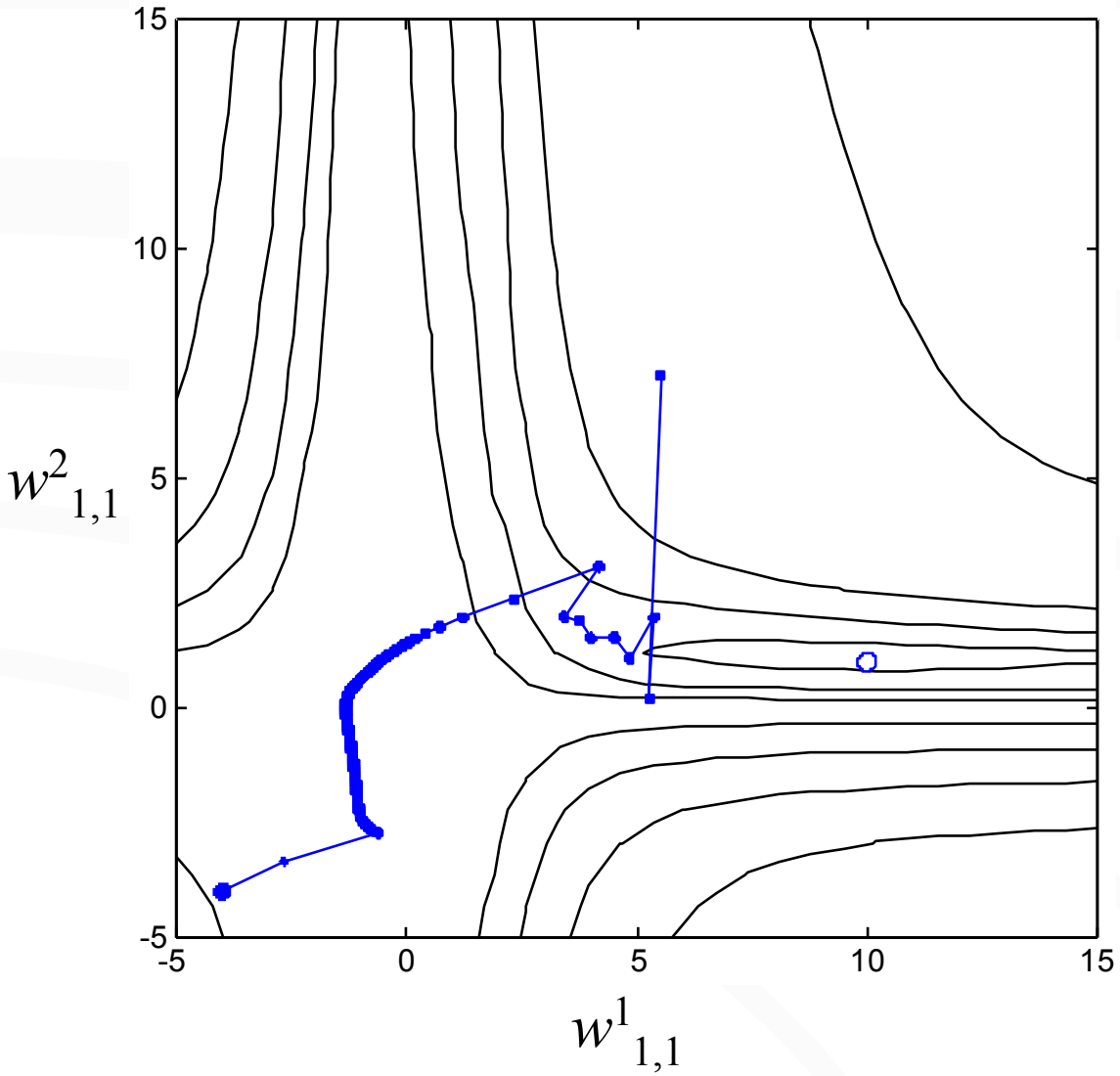


Figure 7.7. Illustration of the effect of adding a momentum term to the gradient descent algorithm, showing the more rapid progress along the valley of the error function, compared with the unmodified gradient descent shown in Figure 7.4.

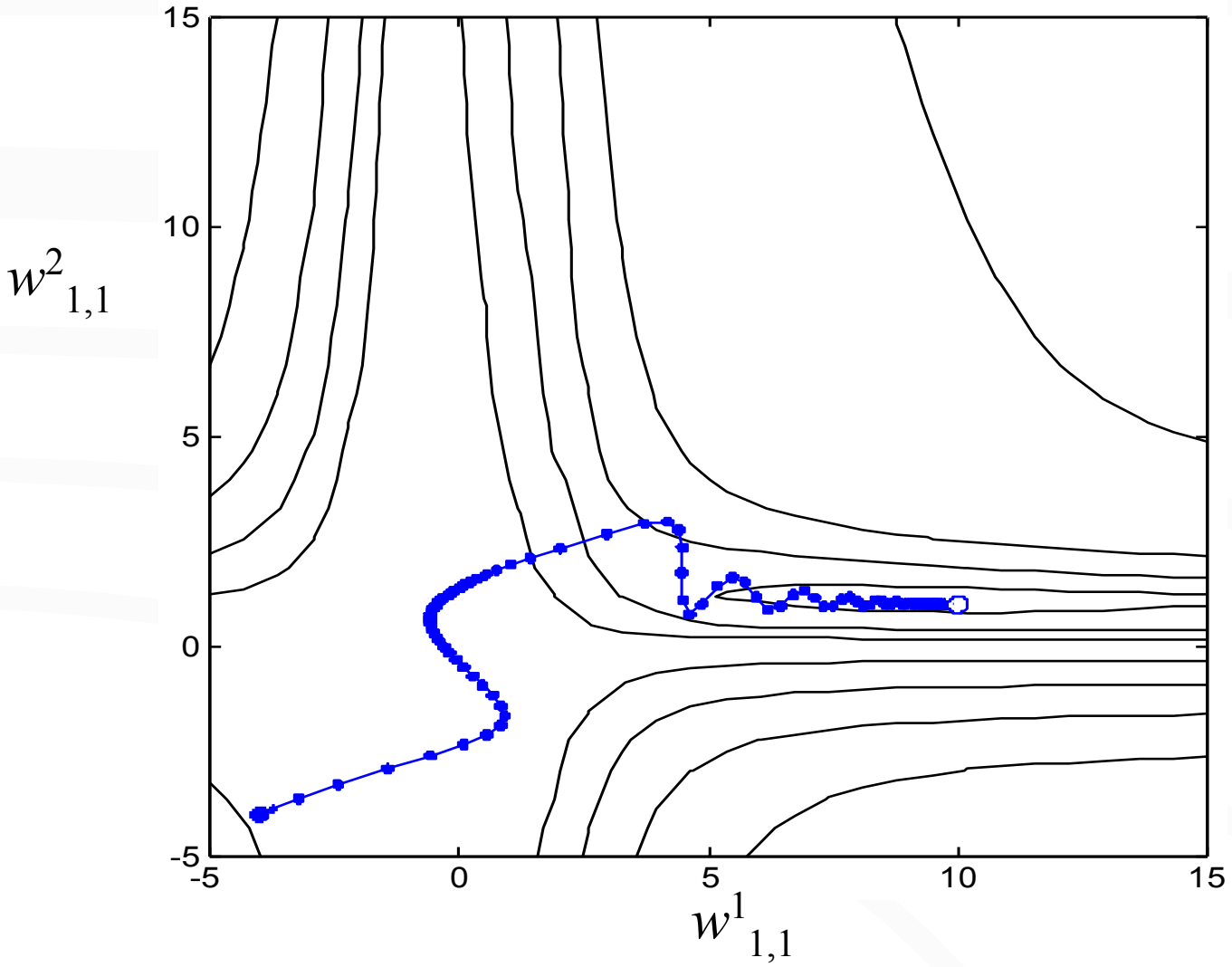
Convergence Example of Backpropagation



Learning Rate Too Large



Momentum Backpropagation

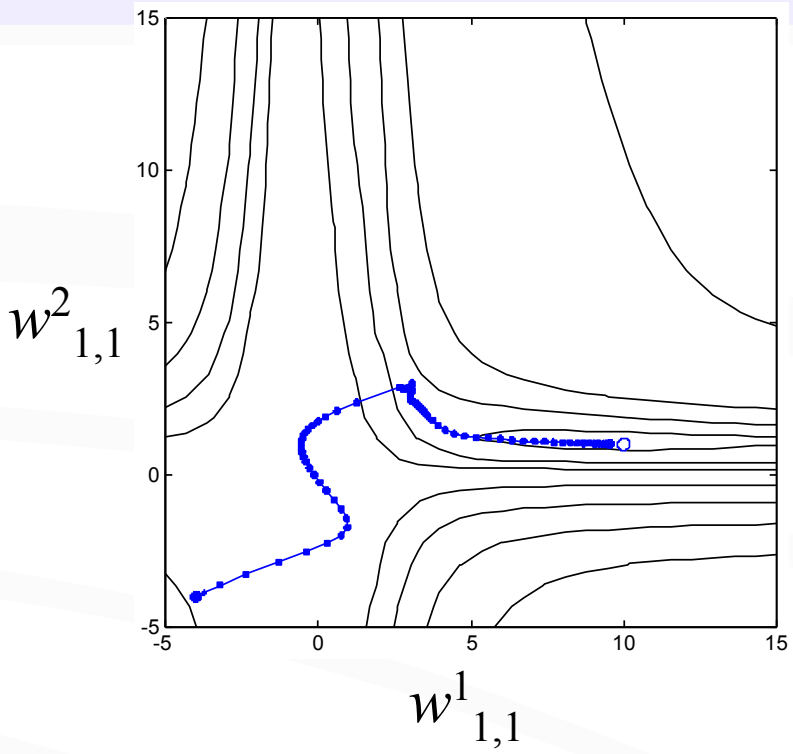


$\gamma = 0.8$

Variable Learning Rate

- If the squared error decreases after a weight update
  - •the weight update is accepted
  - •the learning rate is multiplied by some factor $\eta>1$.
  - •If the momentum coefficient γ has been previously set to zero, it is reset to its original value.
- If the squared error (over the entire training set) increases by more than some set percentage ζ after a weight update
  - •weight update is discarded
  - •the learning rate is multiplied by some factor $(1>\rho>0)$
  - •the momentum coefficient γ is set to zero.
- If the squared error increases by less than ζ, then the weight update is accepted, but the learning rate and the momentum coefficient are unchanged.

Example



$\eta = 1.05$

$\rho = 0.7$

$\zeta = 4\%$