0.0.1 Classification

Classification is applying the result of clustering to individual samples. For example, an online book retailer may have two clusters of customers: those who like science fiction and everyone else. Presented with lots of individual *labeled* samples, in this case, customer information (purchase histories) with science fiction preference (the label), we need to come up with a rule (returning *yes* or *no*) to apply to any future unlabeled customer. The clusters and labels are already there (that's the job of clustering)—we are just generalizing them into a rule.

Such a 'rule' often takes the form of a centroid—the center point in the cluster. For example, consider the clusters in Figure **??**. Given a sample to classify, we may compare it to the *center* of each cluster—assuming that the center is representative of the cluster in some way. The cluster with the closest center wins.

Another classification scheme involves defining a rule—usually a hyperplane (as in Figure 1) or a curve through the sample space. Classification is then accomplished by plugging in the sample into the plane equation, and seeing on which side of the plane the point ends up. Of course our choice of a line is an assumption, and it may not be accurate—we may need to use a curve, or several layered classifiers to achieve an acceptable 'rule'.

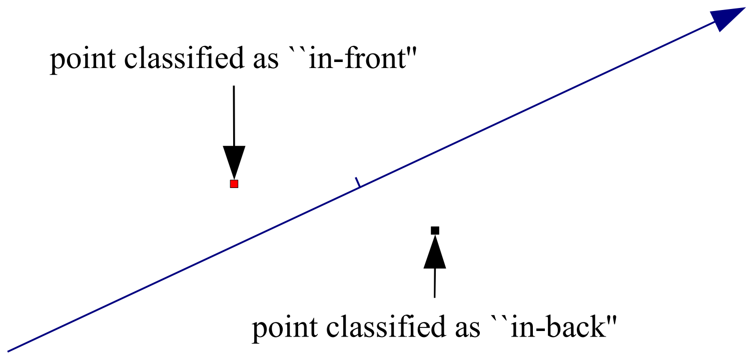
point classified as ``in-front"
point classified as ``in-back"

Figure 1: Classification. The line is our rule. Point is either 'in-front' or 'in-back'.

There are many ways and reasons to do classification. Email spam filters use labeled emails to build a Bayesian network to classify previously unseen emails as either spam or not. A genetic algorithm, after running through the fitness routine, classifies each point as surviving or not for the next generation. We can even view arithmetic as classification—we may classify $2 + 2$ as being in the class of 4.

0.1 Least Squares Discriminator

While the 'least squares' method is used primarily for interpolation and extrapolation, a similar technique can be used for classification [LV00]. Given a training set:

$$S = \{(\boldsymbol{x}_1, y_1), \ldots, (\boldsymbol{x}_L, y_L)\}$$

where $y_i \in \{-1, +1\}$ indicates the class. Our model is a hyperplane, with weights $\boldsymbol{w}$, and distance D, such that:

$$w_1 x_1 + \cdots + w_N x_N = D$$

With such a hyperplane, we get a notion of things being in 'front' of the plane and in the 'back' of the plane. If we plug $\boldsymbol{x}$ into the plane equation (represented by $\boldsymbol{w}$ and D), and get a positive value, then $\boldsymbol{x}$ is in front of the plane, etc.

To turn this problem into the 'least squares' problem described above consider the dual solution. We only used the inner products to find the interpolating line. Now we need to incorporate the y_i values into that Gram matrix. What we end up with is known as a Hessian matrix:

$$H_{ij} = y_i y_j \boldsymbol{x}_i \boldsymbol{x}_j$$

We can then obtain a KKT[1] system:

$$\left[\begin{array}{c|c} 0 & \boldsymbol{y}^T \\ \hline \boldsymbol{y} & H \end{array}\right] \left[\begin{array}{c} -D \\ \hline \boldsymbol{\alpha} \end{array}\right] = \left[\begin{array}{c} 0 \\ \hline \mathbf{1} \end{array}\right]$$

where $\boldsymbol{y} = (y_1, \ldots, y_L)$, $\mathbf{1} = (1_1, \ldots, 1_L)$, H is the Hessian matrix, and $\boldsymbol{\alpha} = (\alpha_1, \ldots, \alpha_L)$ are Lagrange multipliers. We can then solve for $\boldsymbol{w}$ via:

$$\boldsymbol{w} = \boldsymbol{X}^T [\boldsymbol{\alpha} \times \boldsymbol{y}]$$

where $\boldsymbol{\alpha} \times \boldsymbol{y}$ is an element-wise multiplication.

0.2 Φ-embedding

Least squares is a great linear method (we fit things to lines, hyperplanes, etc.), but what if our data isn't linear? Since learning non-linear models is extraordinarily difficult (there doesn't seem to be any general method of doing it), a practical approach is to make non-linear data linear. We can do this by defining a non-linear function Φ, and transforming our data using that function. In other words, instead of working with x, we would work with $\Phi(x)$.

The function Φ can be anything at all. It can reduce or increase the dimensionality of the sample point x. For example, a 2-dimensional x may be turned into a 3-dimensional $\Phi(x)$. i.e.: (x_1, x_2) mapped to $(x_1, x_2, \sqrt{2}x_1x_2)$. This has the capacity of turning our 'learning lines' method into a 'learning curves' method.

[1] Karush-Kuhn-Tucker

To see why this works, consider fitting points to $y = Be^{Ax}$. We can take log of both sides to get $\ln(y) = Ax + \ln(B)$, which is linear. The Φ embedding would apply ln function to Y, and upon output, apply exponential to get B. This technique only works for trivial nonlinear cases.

0.3 Kernels

The dual solution to least squares method requires the creation of the Gram matrix, $\boldsymbol{G} = \boldsymbol{X}\boldsymbol{X}^T$, where

$$\boldsymbol{G}_{ij} = \boldsymbol{x}_i \cdot \boldsymbol{x}_j$$

The dot product operation is often used to find similarity between vectors [Kor97]. We can define a *kernel function* [STC04] that will return the same notion of similarity, except it would not be bound to being linear, might simplify ϕ-embedding, and allow us to handle input data of unlimited dimensions.

For an example, consider our Φ embedding: (x_1, x_2) to $(x_1, x_2, x_1x_2\sqrt{2})$. Every entry in the Gram matrix will be:

$$(x_1, x_2, x_1x_2\sqrt{2}) \cdot (z_1, z_2, z_1z_2\sqrt{2})$$

this can be simplified to:

$$(x_1z_1 + x_2z_2)^2$$

Notice that this is just a dot product of the original two-dimensional inputs squared.

0.4 Support Vector Machines

One of the major problems with the least squares method is that it becomes impractical with a relatively low number of samples. For N input samples, we would need an N by N Gram matrix, with most matrix multiplications taking $O(N^3)$ operations—consider a modest problem with 10000 samples to get an idea of how quickly this becomes impractical.

This is where *Support Vector Machines* [CV95] come in. SVMs are binary classifiers, identical to least squares discriminator in every way, except they don't use all the input samples for training. The important points, as far as classification is concerned, are the ones on the boundaries. If we use just the boundary points, the classifier will be just as good as if we used all the points. The big question now is how to find the boundary points.

Most algorithms have a notion of 'working set'; where in every iteration the algorithm picks a 'working set' of input points to use for training. The working set is generally relatively small. A technique described in [Joa98] picks those points for the working set which have the maximum influence on the resulting classifier (essentially picking inputs with the corresponding largest Lagrange multipliers).

Specialized Boosting-like methods can be used for linear SVMs (no kernels) that run in linear time [Joa06]. SVMs can also be extended to online learning scenarios [SLS99].

1 Connectionism

In the early days of Artificial Intelligence, researchers got the idea that the shortest road to 'intelligence' was to copy something that supposedly already has it. That is, they attempted to mimic the functionality of the brain. By observing brain tissue, they concluded that the brain is a network of cells, called 'neurons'. Each neuron is supposedly computing a simple function of its inputs, and 'fires' (outputs a burst of electrochemical energy) as an output.

Thus, a simplistic (and biologically quite incorrect) computational model emerged.

1.1 Perceptrons

There is a lot of terminology used to describe connectionist systems, mostly due to the way the field developed over time, and slight variations on the same theme. For our purposes, we will refer to perceptrons as neurons and vice versa.

Perceptrons are linear discriminators, meaning that they use a linear function, such as a line, to split the input domain into two classes, the 'in front' class, and 'in back' class.

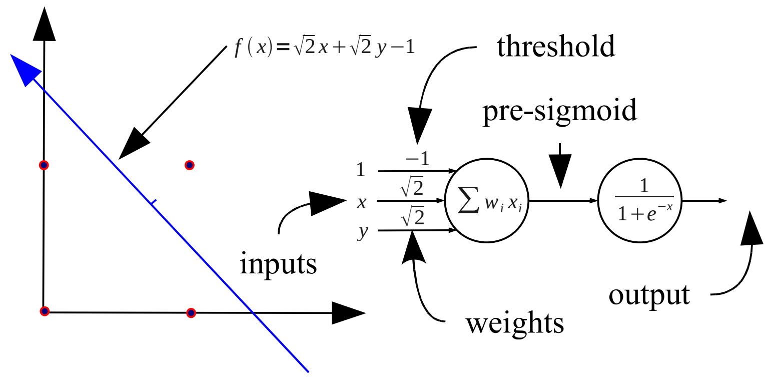

Figure 2: Sample perceptron.

One can easily visualize this by drawing a line on a piece of paper, and noting that the line splits the paper in two halves; Figure 2. Inputs on the same side of the splitting line have the same class. When we train a perceptron using labeled data, we are in effect learning where this splitting line should be.

Perceptrons generally have two parts, the summing part, and the threshold part. The summing part simply performs the inner product of inputs and outputs, while the threshold part applies either the *step*, or more commonly, the *sigmoid* function to the result of the inner product.

$$step(x) = \begin{cases} 0 & \text{if } x \leq 0 \\ 1 & \text{otherwise} \end{cases} \qquad sigmoid(x) = \frac{1}{1+e^{-x}}$$

What function is the perceptron in Figure 2 computing? If we test out every point, such as $(0, 0)$, $(0, 1)$, $(1, 0)$ and $(1, 1)$, we'll see that the function approximates the logical operator *AND*.

While pictures are great for visualizing perceprons, it is much easier to express and work with them using vector and matrix notation. For example, the above perceptron can be seen as a column vector of weights, such as:

$$\begin{bmatrix} \sqrt{2} \\ \sqrt{2} \\ -1 \end{bmatrix}$$

The operation then becomes

$$\begin{bmatrix} x & y & 1 \end{bmatrix} \times \begin{bmatrix} \sqrt{2} \\ \sqrt{2} \\ -1 \end{bmatrix} \qquad \text{leading to} \qquad \mathit{sigmoid}(\sqrt{2}x + \sqrt{2}y - 1)$$

1.2 Limitations

There are certain problems perceptrons are incapable of dealing with, such as approximating the *XOR* function, see Figure 3. There is no line to separate the sample inputs—imagine inputs with labels: $((0, 0), 0)$, $((0, 1), 1)$, $((1, 0), 1)$, and $((1, 1), 0)$, and try to separate them using a single straight line. Such problems are not linearly separable—and generally require a different approach to solve (such as several layers of perceptrons).

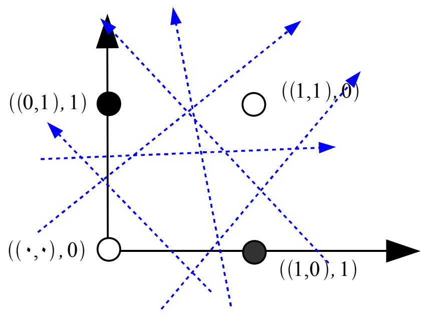

Figure 3: XOR function, and some lines that fail to classify the points into two correct classes.

1.3 Perceptron Learning

Learning in this context refers to finding appropriate weight values, given a set of examples of the form $(\boldsymbol{x}, t)$, where $\boldsymbol{x}$ is the input, and t is the desired label. We start with small $(-0.05$

to 0.05) random weights, and while iterating through samples, apply the learning rule:

$$w_i = w_i + \eta(t - o)x_i$$

Where t is the target label, x is the input, and η is the learning rate, normally set to something small, like 0.05.

The meaning of o depends on the learning method. In *perceptron learning*, o is the final output of the perceptron (after applying the thresholding function, such as $step(x)$). The values of o are generally either -1 or 1.

In *gradient descent*, o is the pre-threshold value, or the inner product of input and weights, and its value has a very wide range.

Notice that with perceptron learning, the learner only knows how many errors it makes for the training data, while in gradient descent, the learner has a good idea of how well the line fits the training data (not just the number of errors). Gradient descent can use this extra information to improve the fit between the model and the data—in fact, while perceptron learning algorithm will not converge on non-linearly separable training data, gradient descent will still manage to find a line that fits (with the least number of errors).

1.4 Neural Networks

The next logical step is to use multiple perceptrons and to layer them, as in Figure 4. These are called Neural Networks, and unlike the perceptron, can, with enough layers, approximate any function (not just linearly separable ones).

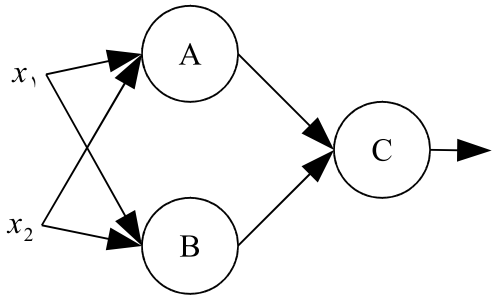

Figure 4: Simple Neural Network

Conceptually, these can be viewed as applying multiple perceptrons, where each only cares about classifying the input on some particular feature—and then doing some aggregate of those results.

For 'XOR' function, for example, the first layer could have two perceptrons, with weights $(-1/\sqrt{2}, 1/\sqrt{2}, -\sqrt{2}/4)$ and $(1/\sqrt{2}, -1/\sqrt{2}, -\sqrt{2}/4)$. Each perceptron would return either a 0 or 1 value, depending on whether their inputs are positive or negative. The final perceptron (in second layer) could take those two outputs (from first layer) and apply weights $(1, 1, -0.5)$

to them. The resulting final output (after the step function) is precisely the 'XOR' function.

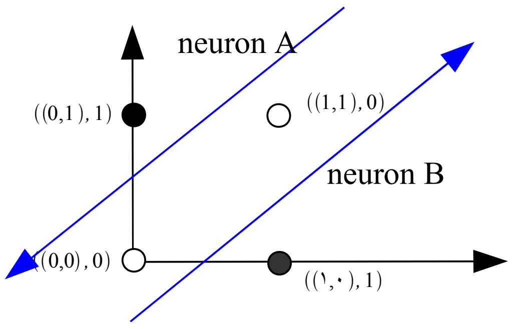

Figure 5: Neural Network separates the samples.

To get an intuition of how this works, notice that the first layer has two neurons, which means it makes two 'cuts' through parameter space, Figure 5. The parameter space is: $(0,0),(0,1),(1,0),(1,1)$. The first node of first layer cuts off $(0,1)$. The second node of the first layer, cuts off $(1,0)$. The third layer just aggregates those activations into a single 0 or 1 output.

1.5 Backpropagation

Now that we know neural networks can compute "XOR" (and NAND, if needed), we can use them to compute any conceivable function[2]. Unfortunately, there's this complicated issue of coming up with the weights for these neurons. For the overused XOR example, we just came up with the values manually (by drawing lines). For more complicated functions, some automated learning method is needed.

Training a neural network may be accomplished via backpropagation [Mit97, Roj96]. The idea is that we have a labeled training set; samples from parameter space, with correct outputs, for example, for XOR function, we might have $(0,0,0),(0,1,1),(1,0,1),(1,1,0)$, where the first two values are x, y, and the third value is the label.

Backpropagation is essentially gradient descent of all the weights of the network. For gradient descent, we need to know the *error* or direction of where to move—once we know that, we can advance in that direction using the learning rate. Finding the direction of the move for a single perceptron was simple; it is just a difference between the expected (target) value and the actual output of the neuron. For a layered network, it gets a bit trickier. Essentially, we need to compute the derivative of the whole neural network—which is difficult. Backpropagation is a clever method to compute the network derivative a small piece at a time, using only the local information available at each neuron.

[2]Since computers are made up of NAND gates, neural networks, with at least 2 layers, are capable of doing everything that a general purpose computer can do.

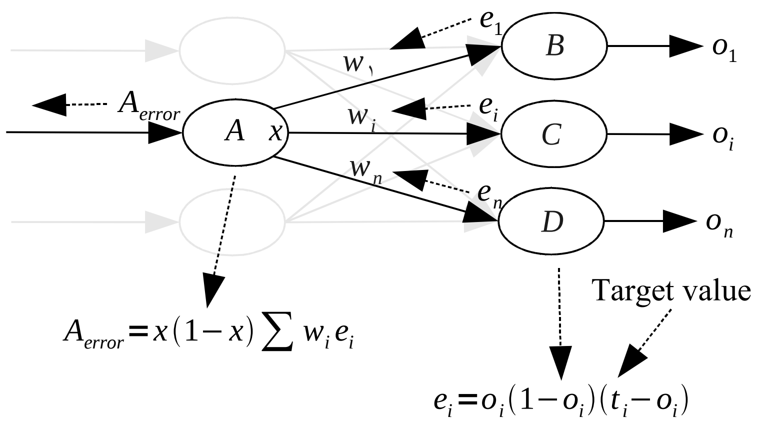

Figure 6: Backpropagation for a single link.

Backpropagation is a two step process. We first feed forward the input through the network, keeping track of all output values at each layer. We then calculate the error of the final layer as we did in the single perceptron case (just take difference between expected and actual outputs, multiplied by the derivative of the *sigmoid* function[3]). We then propagate that error down the layers, adjusted (weighted average) by the appropriate weight (if the weight is high, then that link contributed a lot to that error, etc.). Refer to Figure 6: we directly calculate e_i (i.e.: error), and to *backpropagate* that error to a previous layer, we calculate the A_{error} by taking a weighted (the w_is) average of e_is, and multiplying that by derivative of *sigmoid* for that neuron. This can be repeated for any number of layers. Once we have an error for each neuron, we can use the perceptron learning algorithm on it.

Neural network training would normally iterate through the training data multiple times, applying backpropagation every time, until some condition is met. Either we iterate a fixed number of times, or stop when sum squared error stops improving or reaches some threshold.

In general, backpropagation tends to do very badly on non-trivial problems (such as networks with more than 2 layers), and requires quite a bit of tweaking. Somewhat paradoxically, it also tends to do badly on many trivial problems—simple functions with few dimensions. The reason is that with low dimensional inputs, there is a high chance of quickly getting stuck in a bad local minima (see Section **??** on gradient descent), while with high dimensional inputs, there is a high chance of getting out of local minima via some downward leading dimension.

Some of the improvements involve adding momentum and/or adding regularization. Momentum [Mit97] considers previous weight update as part of current weight update—allowing gradient descent to roll over small bumps (local minima).

Regularization [Mac02] addresses a problem often associated with learning weights: the

[3]The derivative of *sigmoid* is just $sigmoid(x)(1 - sigmoid(x))$

perceptron learning rule encourages weights to get outrageously huge; over fitting the training set. This can often be avoided by starting with very small weights, and quitting before over fitting occurs. The regularization technique essentially adds a weight decay value, so with every iteration, weights tend to get smaller—even while the learning rule is trying to make them bigger. One of the problems with regularization is that the perceptron weights become ‘stuck’ near the origin.

1.6 Recurrent Networks

Recurrent networks are networks that are not strictly layered; more of a general graph structure. These can have outputs going back as inputs, etc. Such networks are considered to be more representative of how the brain might work [GK02], as they can have memory, and do timing tasks [Str03].

Backpropagation can be applied in recurrent networks, except instead of layers, we might imagine a predecessor concept—where we backpropagate errors to the predecessor neurons, etc. Obviously these things can fall into infinite loops—so normally one would backpropagate only N “hops” back, etc.

1.7 Hopfield Networks

A Hopfield network is a form of recurrent artificial neural network that can function as content addressable associative memory [Hop82, Mac02].

Essentially it’s a network where correlation in activity corresponds with weight between different neurons. Imagine stimulus m causing neuron m to be activated, and stimulus n causing neuron n to be activated. If both stimulus m and n appear in the environment together, the network will learn the activity correlation. If at a later time the network is presented stimulus m, it can recall n.

1.8 Autoencoders

The idea behind *autoencoders* is to train the network of the form pictured in Figure 7.

If we feed this network samples such as: 10000000, 01000000, 00100000, 00010000, 00001000, 00000100, 00000010, 00000001, and use input itself as the target, something remarkable happens: the network learns to equate the input with a binary encoding of the input, such as 001, 010, 011, 100, 101, 110, and 111. In other words, it finds a more compact representation of the data [Mit97].

Now consider a more extreme example in Figure 8. The input and output is a 256×256 image, and the goal is to learn the hidden layer. Once trained, the hidden layer will have a compact representation of an image—it will have the most important features of the input image. Trained on a collection of images, it will pick out the most important feature out of all of them [HS06, SH07].

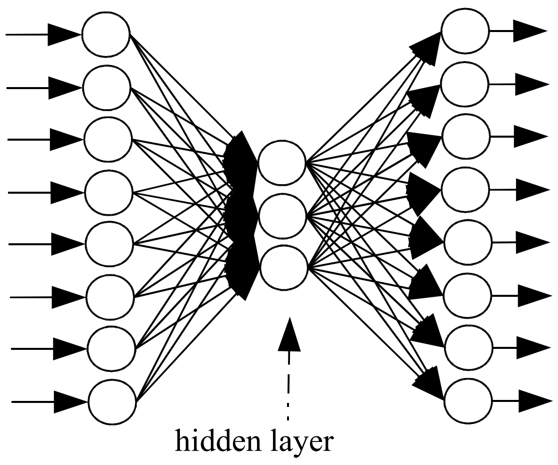

Figure 7: Learning the hidden layer finds a new compact representation of the data.

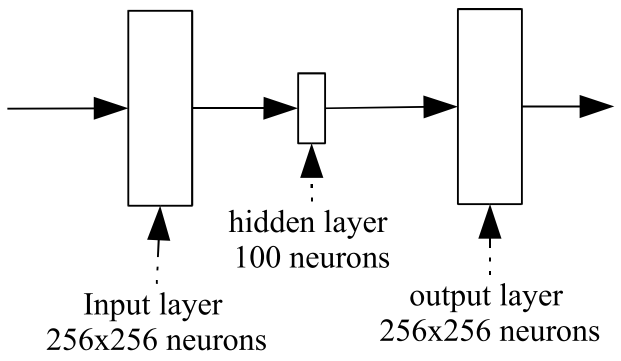

Figure 8: Learning the hidden layer finds the most important features of the input image.

We can layer such architectures like onions, and train them in steps, as in Figure 9. First train the outer layer, then unroll, and train the inner layer using the inputs/outputs of the first layer. Then unroll again, etc. Such an approach allows us to build *deep neural networks*.

Applications for autoencoders (or deep belief nets, as they're sometimes referred to) include image processing and character recognition [OH08, HOT06, HO06]. Essentially a deep network is trained to pull out the most relevant features from many sample patches of writing. In computer vision, autoencoders are used for object recognition [LHB04, LCH$^+$06].

1.9 Evolving Neural Network Weights

Backpropagation operates by piece-wise computing the derivative of the neural network function, and then adjusting the weights in a gradient descent manner. Evolving neural

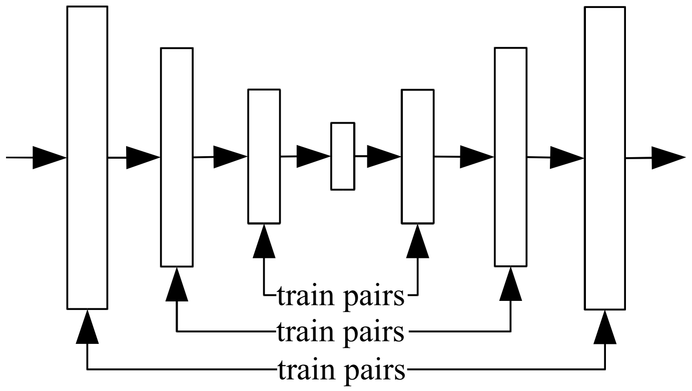

Figure 9: Autoencoders are trained in layers.

network weights essentially relies on sampling the weight space—and keeping only good values.

Evolving a neural network is accomplished by assuming that all weights (let's say there are N of them) collectively make up a 'point' in N-dimensional space. We can make a whole population of these points, let's say 1000.

We then calculate how each of these performs on the sample set, and eliminate the weaker half (or some percentage). We reproduce, using some replication method, a few hundred nodes, so that we again have 1000. We repeat this procedure over and over again—which will hopefully lead us to having weight points that do relatively well on the problem domain (only the best survive each cycle—we are left with best of the best of the best—though these may themselves be pretty bad).

Iterative methods rely on the assumption that the function you're optimizing is convex (that you'll eventually get to the minimum by applying small steps). If the function is not convex, then any iterative method will potentially get stuck in a local minimum.

References

[CV95] Corinna Cortes and Vladimir Vapnik. Support Vector Networks. In *Machine Learning*, volume 20, pages 273–297, 1995.

[GK02] Wulfram Gerstner and Werner M. Kistler. *Spiking Neuron Models.* Cambridge University Press, August 2002.

[HO06] Geoffrey E. Hinton and Simon Osindero. A fast learning algorithm for deep belief nets. *Neural Computation*, 18, 2006.

[Hop82] J.J. Hopfield. Neural networks and physical systems with emergent collective computational abilities. In *Proc. Natl. Acad Sci*, volume 79, pages 2554–2558, 1982.

[HOT06] Geoffrey E. Hinton, Simon Osindero, and Yee-Whye Teh. A fast learning algorithm for deep belief nets. *Neural Comp.*, 18(7):1527–1554, July 2006.

[HS06] G. E. Hinton and R. R Salakhutdinov. Reducing the dimensionality of data with neural networks. *Science*, 313(5786):504–507, 2006.

[Joa98] T. Joachims. Making large-scale support vector machine learning practical. In A. Smola B. Schölkopf, C. Burges, editor, *Advances in Kernel Methods: Support Vector Machines*. MIT Press, Cambridge, MA, 1998.

[Joa06] Thorsten Joachims. Training linear SVMs in linear time. In *KDD '06: Proceedings of the 12th ACM SIGKDD international conference on Knowledge discovery and data mining*, pages 217–226, New York, NY, USA, 2006. ACM.

[Kor97] Robert R. Korfhage. *Information Storage and Retrieval*. John Wiley & Sons, Inc., New York, NY, USA, 1997.

[LCH$^+$06] Yann Lecun, S. Chopra, R. Hadsell, F. J. Huang, and M. A. Ranzato. *A Tutorial on Energy-Based Learning*. MIT Press, 2006.

[LHB04] Y. Lecun, F. J. Huang, and L. Bottou. Learning methods for generic object recognition with invariance to pose and lighting. volume 2, 2004.

[LV00] S. Lukas and L. Vandewalle. Sparse least squares support vector machine classiers. *European Symposium on Articial Neural Networks*, pages 37–42, 2000.

[Mac02] David J. C. Mackay. *Information Theory, Inference & Learning Algorithms*. Cambridge University Press, June 2002.

[Mit97] Tom M. Mitchell. *Machine Learning*. McGraw-Hill, New York, 1997.

[OH08] Simon Osindero and Geoffrey Hinton. Modeling image patches with a directed hierarchy of Markov random fields, 2008.

[Roj96] Raul Rojas. *Neural Networks: A Systematic Introduction*. Springer, 1st edition, 1996.

[SH07] Ruslan Salakhutdinov and Geoffrey Hinton. Semantic hashing. In *Proceedings of the SIGIR Workshop on Information Retrieval and Applications of Graphical Models*, Amsterdam, 2007.

[SLS99] N. Syed, H. Liu, and K. Sung. Incremental learning with support vector machines. *International Joint Conference on Artificial Intelligence*, 1999.

[STC04] John Shawe-Taylor and Nello Cristianini. *Kernel Methods for Pattern Analysis.* Cambridge University Press, June 2004.

[Str03] Steven Strogatz. *Sync: The Emerging Science of Spontaneous Order.* Theia, March 2003.