Applications of Back-Propagation

1 Classification

The reason we've been looking at neural networks is that they give us yet another way of encoding action functions in reactive agents. In the lecture, we'll see a demo of a wall-following agent whose action function is encoded in a neural net. The weights of the net were learned by training the agent to follow the walls of a training room. *See the demo in the lecture.*

This demo is not as fanciful as it may seem. The ALVINN system was an autonomous car, whose action function was encoded in a neural net. The net had 960 input units, corresponding to the 30×32 pixels obtained from a camera that was mounted on the car. The net had one hidden layer, containing 4 units. And the output layer contained 30 units, each one corresponding to different amount of steering wheel turn (e.g. one for sharp left, one for straight ahead, one for sharp right and 27 for various lesser amounts of left or right turn). ALVINN was trained on 5 minutes of data in which camera images were paired with human steering wheel actions. ALVINN learned to drive (i.e. to steer) successfully at speeds of up to 70 m.p.h. and for distances of 90 miles on two-way public roads amid other traffic.

Learning action functions is, however, not the only use for ANNs within AI. The ANNs that we have been looking at are ideal for *classification tasks* in general. Here, we'll describe what we mean by classification tasks, show how common they are (including their practical applications), and look at one example in detail.

Classification is the task of deciding which class something belongs to. In AI, we define it a little more precisely. We receive some data that describes some object or phenomenon. The data is typically in the form of a set of values for various attributes or features of the object or phenomenon. (The values may be Booleans, integers, reals or more complex structured data types.) On the basis of this data, we then decide which of a *predefined* set of classes this object or phenomenon belongs to. We output the name (or *label*) of the class that we decide upon. (In some variants of the task, we might be allowed to output no label if the object or phenomenon doesn't belong to any of the classes, or we might be allowed to output more than one label if it belongs to more than one class.)

Our definition of classification requires that the classes already be known to us. They're predefined. They have already been enumerated. (It's also typically the case, and some people include this as part of their definition, that there be only a 'small' set of classes.) Sometimes there are only two classes. This is the case where you want to decide whether your object or phenomenon belongs to some class or not. The output labels can then simply be 0 or 1 (**true** or **false**).

Simple forms of medical diagnosis can be construed as classification tasks. The data would be symptoms of the patient (e.g. a Boolean-valued attribute that says whether the patient has spots or not) or measurements about the patient (e.g. a real-valued attribute that records the patient's temperature). The classes would be diseases or other medical conditions (e.g. various kinds of glaucoma). Optical character recognition (and similar pattern recognition tasks) are also classification tasks. The attributes might describe geometrical properties of a character that has been scanned into the computer (e.g. there is or isn't a long vertical line), and the classes are letters of the alphabet. In credit-worthiness decisions, attributes describe the financial status and history of the applicant; and there might be only two classes: credit-worthy or not credit-worthy. Finally, the task performed by the action function of a reactive agent can also be construed as classification: the agent selects one of a small predefined set of actions on the basis of its sensory inputs.

Classification is important in itself. But it gains added importance because it often crops up as a subtask within other problem-solving tasks. For example, a robot might need to select a particular action if it encounters a door; to know whether it is looking at a door requires classification of the inputs received from its vision system. Or, an agent that is supposed to help humans navigate around a web site might need to decide, on the basis of the pages that the human visits, what the human's goal is. Here, users are classified by goal based on sequences of browser events.

2 Preparing yourself

If you want to build a classification system for some problem domain, you have to make some decisions. First, you need to decide what attributes or features you are going to use when describing the objects or phenomena that you want your system to classify. This might be done manually by a knowledge engineer (an AI guy) in collaboration with a domain expert (someone who is knowledgeable about the problem domain). But it can also be done automatically: if you have a database of data about your objects or phenomena, there are AI systems that can determine which values in this data are most predictive of the classification.

Second, you need to decide on your set of classes. Here again, you might get a knowledge engineer and a domain expert to determine them, or, if you have some suitable data, you might use an AI program to determine them.

Third, you must decide how your classification system is going to work. We'll be looking at ANNs but bear in mind that there are, in fact, many many other technologies.

3 Methodology

Let's apply our ANN implementation to a classification task.

There's no point using something as simple as XOR. This would not be a typical example. There are only four possible inputs, so we can train using all four. Normally, the space of possible inputs is much larger — too large for exhaustive training. For example, consider the number of different visual stimuli involved in face or handwriting recognition.

Instead, we train using a small proportion of the input space. But then, of course, we hope that the learned net will subsequently correctly classify inputs that were not in the training data, *unseen examples*. A net is said to *generalise* if it can correctly classify unseen examples. And we judge this by measuring the *accuracy* of its classifications.

For our example, we can get some data from a web database much used by machine learning researchers and hosted by the University of California, Irvine. This database is known as the UCI Machine Learning Repository. Each data set in the database is different. E.g. in some, each object has a value for each attribute; in others, there are objects which have unknown values for certain attributes (something that is very common in real data); in some, the target outputs are guaranteed to be accurate; in others, the data is *noisy* (some of the data may be wrong — in other words, we would be training our network using data that includes incorrectly described or misclassified objects; again this is common in real data).

I chose to use the Iris Database. This contains data about 150 plants. All of them are irises of some kind. There are 4 attributes (all numeric-valued): sepal length in cm; sepal width in cm; petal length in cm; and petal width in cm. (There is no missing data.) There are three classes: a plant can be of class Iris Setosa (class label $\langle 0, 0\rangle$), Iris Versicolour (class label $\langle 0, 1\rangle$) or Iris Virginica (class label $\langle 1, 0\rangle$). In the data, there are 50 examples of each of the three classes. The data is not (to the best of my knowledge) noisy. (Of course, if we want to experiment with noisy data, we simply randomly change some of the data.)

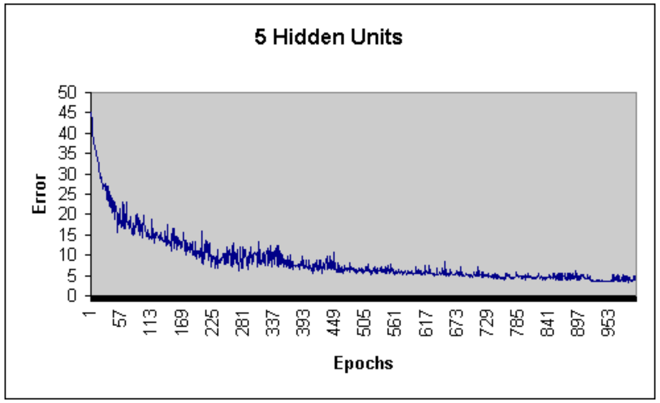
I had to decide on a network topology. Obviously, I need 4 input units (excluding the 'extra' threshold unit) and two output units. My Java implementation of back-prop only allows one hidden layer, so, with my software I've no real choice about how many hidden layers to use! But I still need to decide how many hidden units there will be in that layer. I decided to try out a number of different cases: I tried out 5, 10 and 15 (excluding the 'extra' one).

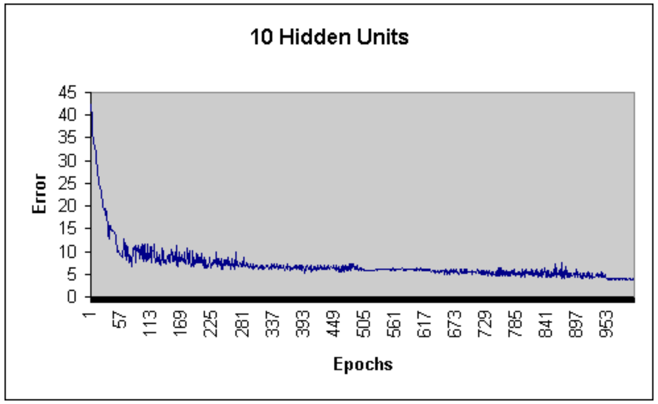
I split the data into two. I used 66% of the data as the *training set* within the back-prop algorithm. The remaining 34% I used as a *test set* (or *validation set*). In other words, after running the back-prop algorithm using the training set, I then used the learned ANN on each plant in the test set, and I was able to see whether the ANN made a correct prediction on this unseen data (data it wasn't trained on). It's from this we can judge whether there has been any generalisation.

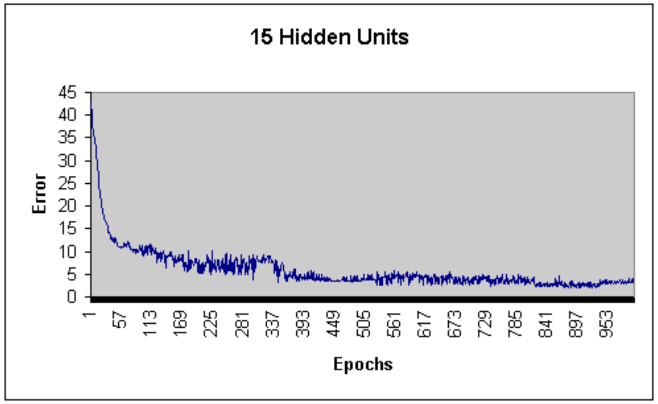
Of course, we have to be careful about how we interpret the results we get. After all, the random split of the data might, by accident, give us a really good training set or a really poor one, or a really easy test set or a really hard one.

To lessen such problems, it is common to measure accuracy using a method called *cross validation*. The idea is to report the *average* accuracy obtained from *different splits* of the data into training set and test set. In other words, you split the data into training set and test set, train the net on the training set, test it on the test set and note the error. Then, you do this several more times but each time you split the data differently. You then take the average of the errors.

Here are graphs of average error (using 5 cross validation runs) for different numbers of epochs and different numbers of hidden units for the iris data.







We'll discuss these graphs in the lectures.

4 An Evaluation of Back-Prop

4.1 Efficiency

ANNs typically have long training times. But, once the weights have been learned, use of the network for, e.g., classification is usually very fast.

The efficiency of the learning algorithm can be very variable in practice and, in the worst case, is not good at all. If there are n_e examples and n_w weights, each epoch takes $O(n_e n_w)$ time, and, in the worst case, the number of epochs can be exponential in n, the number of inputs.

We saw that a large number of epochs were needed to learn exclusive-or. This was a small net, and we were presenting it with all possible examples!

There are various techniques to try to speed up learning in practice. One is to use a learning rate, α, that is closer to 1. Another is to introduce another term into the update rules called a *momentum factor*. We won't go into details, but in the early epochs, this factor is close to zero, and then, in the remaining epochs, it is set to, e.g., 0.9. This gives the algorithm time to find a good general direction (making small changes) in the early epochs, and then increases the learning speed once that direction has been found.

Of course, another idea is to abandon sequential software simulations and use parallel hardware.

4.2 Convergence

Will the network learn the function that we are training it to learn? Quite apart from how long the process will take, there is the question of whether we will ever reach a situation of zero error (or near zero error).

As previously discussed for TLU learning, if the value chosen for the learning rate is too large, there is a danger of 'overshooting', and even oscillating around, the optimal weights. But there is another danger, which applies to nets but not to single TLUs. The error surface may have local minima. No small change in the weights makes the error any smaller, but the net has not yet reached the global minimum error.

Opinion differs on the significance of this problem; people disagree about whether it is common enough in real domains to be a worry. In domains where you do think that it is a problem, there are techniques for overcoming it. One approach is to allow the learning algorithm to make some weight changes that actually increase the error (see *simulated annealing*).

4.3 Generalisation

As noted earlier, we want generalisation: accurate classifications on unseen data. Will we get it?

The network architecture plays a major role here. The wrong choices can lead to poor generalisation performance. In ANNs, and (I suppose) in learning in general, being short of space (but not too short) forces generalisation. You have to find patterns and suppress distinctions.

If the network is too small, it will not be able to represent the function that it needs to represent: it won't be able to draw the distinctions it needs for making good predictions even on the seen data.

But, if the network is too big, it will have room to simply memorise the examples and act like a large look-up table. In this case, i.e. if there is little or no generalisation, we say that there has been *overfitting* of the data.

How might we come up with a good network architecture? Some people claim to be good at deciding architectures on the basis of the input and output spaces. But even then, they follow their 'informed' guesswork by a period of experimentation with different architectures. The alternative is to seek to do this automatically. One approach is random generation of architectures — but there are too many possibilities to make this a good approach. Another approach is to try to evolve the architecture using a GA. This sounds promising but it is so computationally expensive that it is rarely feasible. More common is to use a *hill-climbing* approach. Again we won't cover the details but basically you start with a single random architecture and make it successively smaller (or bigger) using information-theoretic measures to decide which weights (and hence hidden units) to retain.

4.4 Transparency

Neural networks are 'black-boxes'.

The first problem with this is that they cannot (currently) explain their reasoning. Even if you are satisfied with the prediction accuracy of your network, users may be dissatisfied because the network cannot say *why* it makes a particular prediction. In safety-critical domains (especially where life is endangered) or in domains where litigation risks and costs are high, this is not acceptable. (By contrast, if classification is done using rules to encode necessary and/or sufficient conditions, rather than using nets, paraphrasing the rules that were used to make a prediction gives a rudimentary form of explanation of classification decisions.)

A second, related problem is that neural nets do not give any indication of the certainty of their predictions. (For example, they do not quantify their certainty using probabilities.) Such an indication would often be valuable, especially given that nets are often trained and used on noisy data.

A final, related problem is their inability to encode prior knowledge. By prior knowledge, we simply mean knowledge that your domain expert already possesses about this application domain. (E.g. it might take the form of rules-of-thumb that relate certain inputs to certain outputs.) It's common for us to have some prior knowledge, and if we could incorporate it into the network from the outset, the learning algorithm might give us a better final network. But, there's no easy way to encode this knowledge into a network: the only thing we have control over is the topology.