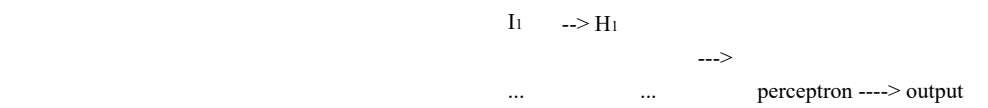
CS181 Lecture 11 — Multi-layer Feed-forward Neural Networks

1 Limitations of Perceptrons

1 As we saw, perceptrons can only represent linearly separable hypotheses. Minsky & Papert's argument against perceptrons was based on this observation, but was a little more subtle than simply saying "Perceptrons can't represent xor, so they can't be a basis for learning and intelligence". There are two counters to that argument. One is that we don't necessarily care if a hypothesis space actually contains the correct hypothesis, only that it can produce a hypothesis that generalizes well to unseen data. In fact, empirical evidence shows that hypothesis spaces of linear separators sometimes perform quite well even when the true hypothesis is not linearly separable. A second counter-argument is that even if a concept is not linearly separable in terms of the raw input data, we can apply some preprocessing to the data to obtain high-level features, and then the hypothesis may be linearly separable in terms of those features. This, in fact, was the scheme of much of the perceptron research in the sixties. A typical system had the following design:



I_{1000} -->H_{100} --->

The input data is contained in attributes I_1 to I_{1000}. These are pre-processed to produce 100 features H_1 to H_{100}, which are then passed to a perceptron. The key point is that the H units are not adaptive — only the weights from the H units to the perceptron units were learned. With this type of scheme, many interesting properties of the data can be represented as linearly separable hypotheses, if the right features are used.

The crux of Minsky & Papert's argument is that using non-adaptive high-level features is not good enough. The number of pre-programmed features that would be needed in order to represent all interesting properties using linearly separable hypotheses is astronomical. For any reasonably sized set of

[1]This section is based on the discussion in Section 3.5.4 of "Neural Networks for Pattern Recognition" by Bishop.

pre-programmed features, there will be some interesting properties that cannot be represented using a linear separator.

An example they give considers a pre-programmed set of features that are restricted to only consider small local regions of the image. Using such features, one cannot tell whether or not an image is connected. For example, consider the following four images:

```
XXXXXXXXX          XXXXXXXXX
X       X                  X
XXXXXXXXX          XXXXXXXXX
                   X
XXXXXXXXX          XXXXXXXXX


XXXXXXXXX          XXXXXXXXX
X
XXXXXXXXX          XXXXXXXXX
        X          X       X
XXXXXXXXX          XXXXXXXXX
```

Now, suppose we only have features that are restricted to looking at individual columns. Whether or not this image is connected depends on the combined configuration of the left and rightmost columns. No local feature can represent the combined configuration of these two columns. In fact, if we consider the two column configurations appearing here, whether or not the image is connected is the xor of the configurations of the left and right columns, which is not linearly separable.

2 Multilayer Feed-Forward Neural Networks

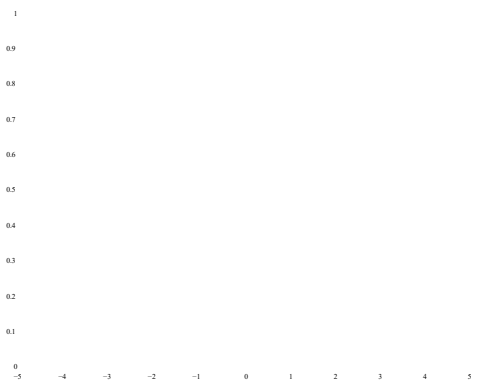
A natural solution to the problem raised by Minsky & Papert is to make the H units adaptive. Then one can hope that the network will be able to learn which high-level features are actually needed for a particular classification problem. Unfortunately, in 1969 people did not know how to learn networks with adaptive hidden features. The answer lies in the back-propagation algorithm, which was developed that same year by Bryson & Ho, but did not really become widely known until the 1980s.

A multilayer feed-forward neural network is a network consisting of multiple layers of units, all of which are adaptive. The network is not allowed to have cycles from later layers back to earlier layers, hence the name "feed-forward". We will start out by considering a network with a single complete hidden layer. I.e., the network consists of some input nodes, some output nodes, and a set of hidden nodes. Every hidden node takes inputs from each of the input nodes, and feeds into each of the output nodes. Following Russell & Norvig's notation,

we denote input units by I_k, hidden units by H_j and output units by O_i. Note that the different units take indices from the same family, so if a network has two inputs, two hidden units and one output, the units are named I_1, I_2, H_3, H_4 and O_5. This means that the subscript of a unit uniquely identifies the unit in a network. We shall often refer to a unit simply as "unit j", meaning the unit with subscript j.

When an example is processed by a multi-layer feed-forward network, each unit has an activation level. We denote the activation level of unit j as a_j. For input units, the activation level is the value of the input in the example. For non-input units, the activation level is computed as a function of the activation levels of the units in the previous layer, in much the same manner as perceptrons. First a weighted sum of the activations of the previous layer is computed, and

then the weighted sum is passed through an activation function g. The weight on the edge from unit k to unit j is denoted w_{kj}. In multi-layer feed forward neural networks, the sigmoid activation function, defined by $g(x) = \frac{1}{1+e^{-x}}$ is normally used. The graph of the sigmoid activation function is shown below:



Although the sigmoid activation function doesn't have a threshold term, recall that we can simulate a threshold by adding an extra input I_0, whose value is always -1. There will be an edge from I_0 to every non-input unit, and the weight w_{0j} corresponds to the threshold of node j.

There are a number of reasons for using the sigmoid activation function. First, it is differentiable, which is required for the back-prop algorithm. Second, it behaves like a threshold unit at the extremes, converging to 0 on the left and 1 on the right. Third, it is approximately linear around 0, which provides a smooth transition between the left and right extremes.

It is easy to see that one can represent more functions with a multi-layer network than with a perceptron. For example, one can easily represent the xor function (how?). In fact, with a single complete hidden layer one can approximate any continuous function from the inputs to the outputs to an arbitrary degree of accuracy. Before you get too excited by this, there is a caveat — the

number of hidden units required may be very large. Just to represent all Boolean functions on n inputs, the number of hidden units required is exponential in the

number of inputs.

3 The Back-Propagation Algorithm

In developing the perceptron and adaline learning rules, we used the difference between the true output and the predicted output to determine how to update the weights. We use the same idea for learning in multi-layer networks, but there is a challenge — how should we distribute the error in the output over the different hidden units, and how should we then update the weights leading into the hidden units? The back-propagation algorithm provides an answer to this question.

Back-propagation is a gradient descent algorithm. The error function to minimize is the total squared error of the training data. We define $err_{d,i}$ to be the error of output O_i on training instance d. The error function is then

$$E(w) = 1/2 \sum_{d \in D} \sum_{i} (T_{d,i} - O_{d,i})^2 = 1/2 \sum_{d \in D} \sum_{i} (err_{d,i})^2$$

There are two kinds of weights in w: weights w_{ji} from a hidden node H_j to an output O_i, and weights w_{kj} from an input I_k to a hidden node H_j. We'll develop an update rule for the first kind of weight.

$$\begin{aligned}
\frac{\partial E}{\partial w_{ji}} &= 1/2 \sum_{d \in D} \sum_{l} 2(T_{d,l} - O_{d,l}) \frac{\partial}{\partial w_{ji}} (T_{d,l} - O_{d,l}) \\
&= \sum_{d \in D} (T_{d,i} - O_{d,i}) \frac{\partial}{\partial w_{ji}} (T_{d,i} - O_{d,i}) \\
&\quad \text{because } \frac{\partial}{\partial w_{ji}} (T_{d,l} - O_{d,l}) = 0 \text{ for } l \neq i \\
&= \sum_{d \in D} err_{d,i} \frac{\partial}{\partial w_{ji}} (T_{d,i} - g(in_{d,i})) \\
&= \sum_{d \in D} err_{d,i} (-g'(in_{d,i}) \frac{\partial in_{d,i}}{\partial w_{ji}}) \\
&= -\sum_{d \in D} err_{d,i} g'(in_{d,i}) a_{d,j}
\end{aligned}$$

This leads to the update rule

$$w_{ji} \leftarrow w_{ji} + \alpha \sum_{d \in D} a_{d,j} err_{d,i} g'(in_{d,i})$$

Normally we train on one instance at a time, so we can drop the summation over instances and the superscript d. We also write $\delta_i = err_i g'(in_i)$ to get the following rule:

$$w_{ji} \leftarrow w_{ji} + \alpha a_j \delta_i$$

The derivation for weights between the input and hidden layer is similar. Without going into details, it turns out that if you define err_j to be $\sum_i w_{ji}\delta_i$, and then define δ_j analogously to δ_i by $\delta_j = err_j g'(in_j)$, you get the update rule:

$$w_{kj} \leftarrow w_{kj} + \alpha a_k \delta_j$$

This is beautiful — the rule for updating weights between the input and hidden units looks exactly like the rule for updating weights between the hidden and output units. The key to making this work is the way err_j is defined. Intuitively, the back-propagation algorithm works out how much each hidden unit H_j contributes to the actual error, and puts this amount into err_j.

It turns out that the back-propagation algorithm generalizes to arbitrary feed-forward networks. We now allow the network to be any directed acyclic graph, and store the set of children and parents of each node. The general form of the back-propagation algorithm for processing a single training instance is as follows:

Repeat until all nodes have been processed
  Pick a node j all of whose children have been processed
  If j is an output
    $err_j = (T_j - a_j)$
  Else
    $err_j = \sum_{\text{children i of j}} w_{ji}\delta_i$
  $\delta_j = g\,(in_j)err_j$
  For each parent k of j
    $w_{kj} \leftarrow w_{kj} + \alpha a_k \delta_j$

This is the algorithm for training on a single instance. A neural network is trained by running back-propagation on each of the instances in a training set repeatedly. One round of running back-prop on each of the training instances is called an epoch.

4 Comments on Back-Propagation

4.1 Implementation Details

The above back-propagation algorithm applies for any differentiable activation function g. For the sigmoid function $g(x) = \frac{1}{1+e^{-x}}$, it has a particularly nice form.

Let us compute the derivative of g:

$$g\,(x) = -\frac{-e^{-x}}{(1+e^{-x})^2} = \frac{1}{1+e^{-x}}\,\frac{e^{-x}}{1+e^{-x}} = g(x)(1-g(x))$$

If we substitute in_j for x, we get

$$g\,(in_j) = g(in_j)(1-g(in_j)) = a_j(1-a_j)$$

We can implement back-propagation efficiently by computing a_j once for each node and storing it, and replacing the term $g\,(in_j)$ with $a_j(1-a_j)$ in the above algorithm. The a_j values can be computed in a forward-propagation phase, as follows:

Repeat until all nodes have been processed
  Pick a node j all of whose parents have been processed
  If j is an input
    $a_j = I_j$
  Else
    $in_j = \sum_{\text{parents k of j}} (w_{kj} a_k)$
    $a_j = \frac{1}{1+e^{-in_j}}$

This forward-propagation phase looks very similar to back-propagation, except that parents are processed before children rather than the other way round.

When implementing back- and forward-propagation in practice, we would rather not have to search for a node to process next, but be able to choose the next node automatically. This can easily be achieved by numbering the nodes in such a way that parents always have smaller index than their children. This can either be forced in the design of the network, or a topological sort algorithm can be used to sort the nodes appropriately. Once this has been achieved, the first two lines of forward-prop can be replaced by for j = 1 to M (where M is the total number of units). Similarly, the first two lines of back-prop can be replaced by for j = M to 1 step -1.

4.2 Complexity

What is the cost of running forward and backward propagation on a single training instance? In forward propagation, each edge contributes one term to one of the in_j sums, while each non-input node results in one computation of g. The total running time is O(M + E) (where E is the number of edges). A similar calculation results in the same cost for back-propagation. Thus the total cost is O(M + E). If we consider networks with a single complete hidden layer, with m inputs, n hidden units and l outputs, then the network has m + n + l nodes and (m+l)n edges, so the total cost is O(m+n+l+(m+l)n). Normally the number of inputs and outputs is fixed, while we allowed to vary the number n of hidden units. Looked at that way, the cost is O(n). Note that this is only the cost of running forward and backward propagation on a single example. It is not the cost of training an entire network, which takes multiple epochs. It is in fact possible for the number of epochs needed to be exponential in n.

4.3 Weight Sharing

One final tweak to the algorithm. If there is a lot of symmetry in a domain, it

might make sense to force some of the weights in the network to be the same. For example, consider an image processing problem. For concreteness, let's say the input data consists of a 3x3 image of nodes, numbered as follows:

I_1 I_2 I_3
I_4 I_5 I_6
I_7 I_8 I_9

Suppose we want to build an array of hidden units that detect features of 2x2 regions. There will be four hidden units, corresponding to the top left, top right, bottom left and bottom right of the image. We want each of these hidden units to compute the same function of its inputs, so we need to force some of the weights in the network to be the same. E.g., the weights from I_1 to the first hidden unit, from I_2 to the second hidden unit, from I_4 to the third hidden unit and from I_5 to the fourth hidden unit should all be the same. Forcing weights on different edges to be the same is called weight sharing. In an implementation of back-prop, weight-sharing can easily be implemented by storing a pointer to the actual weight with each edge, rather than the weight itself. Edges that share weights will point to the same weight.

Weight sharing was used very successfully be leCun et al. in their digit recognition system. There are two good reasons to do it. One is that it exploits prior knowledge that we have about symmetry in the domain. The other is that it reduces the number of weights that have to be learned, and as we shall see, it takes more samples to learn more weights reliably.

4.4 Convergence

So far, we've talked mainly about running one phase of the algorithm on a single example. Training a network requires training on each example many times. The basic form of the full training algorithm is as follows:

```
Repeat until convergence:
  For each training example d
    Run forward propagation on d
    Run back propagation on d
```

What does convergence mean here? It means that the difference in weights

between successive iterations is smaller than some pre-defined tolerance ϵ. As we said, back-propagation is a gradient descent algorithm, so it will eventually converge,[2] (although it may converge to a local minimum).

The number of epochs required until convergence depends on a number of factors. One, of course, is the learning rate. As we discussed earlier, if the learning rate is too small too many epochs may be required, but if it is too big the algorithm may end up oscillating around a minimum. Unfortunately, the right learning rate is problem dependent, and choosing a good one can take some trial and error. Another thing the number of epochs depends on is the function being learned. Simpler functions can generally be learned in fewer epochs than more complex ones. The network structure, and in particular the number of weights to learn, can also determine the number of epochs required.

[2]To be precise, because we are updating the weights after each training example, and not after seeing the entire data set, we are not exactly following the direction of the gradient. Actually, the algorithm performs a stochastic gradient descent, which may follow a slightly different direction from the actual gradient. This doesn't change the basic behavior of the algorithm.

The reason is that the same function can be fitted in very different ways in different hypothesis spaces. So a function that looks simple in one hypothesis space can look very complex in another hypothesis space that is rich enough to model the complexities of the function. (For a simple example, a quartic function looks like a simple quadratic in a hypothesis space that can represent quadratics but not quartics, but it looks like a quartic in a richer space that can represent quartics.)

Local minima can be a really big problem for back-propagation. However, some people claim they are not that much of an issue in practice. One reason is that if there are enough weights in the network, the extra weights can provide for escape routes away from local minima. A local minimum has to be a minimum in all the weights — even if a point is a minimum in some dimensions, other dimensions can allow it to change. Another reason given is that local minima can often be pretty good. Even if a neural network learns an "incorrect" function due to local minima, if the function has reasonably good performance on the training set it will often still generalize reasonably well.

Of course, a good way to deal with local minima is to use random restarts, where each run is started with different initial weights. Using this approach, you end up with a set of networks. You can then choose the one with the best performance on the training set. Alternatively, you can use a voting hypothesis,

in which the weight of each network is determined by its score on the training set. Finally, if you're already using a voting hypothesis, you can go ahead and do boosting, reweighting each of the training examples on each round according to the AdaBoost formula. Weighted training examples can easily be taken into account by the back-prop algorithm, by factoring the weight of an example into the learning rate.

4.5 Overfitting

Although the above algorithm says that you train the network until the weights converge, it turns out that you don't necessarily want to run until convergence. The reason is that running until convergence can lead to overfitting. In fact, if we draw a graph with the number of epochs of training on the x axis, and the error rate on the y axis, we will often see the standard overfitting phenomenon. The error rate on the training set will continue to decrease as the number of epochs increases. However, there will be some point after which the error rate on the test set increases. This is the point at which overfitting happens.

How can we find the optimal time at which to stop training? One possibility is to use a validation set. The training set is divided into two parts, one for actual training of the network and one for validation. The network is trained on the first part, and at the end of each epoch, its error on the validation set is computed. As soon as the validation set error rate starts increasing, training is stopped, and the previous network is returned.

In order to better understand why overfitting is an issue in neural networks, and to gain insight on other ways to prevent it, consider a network with a single complete hidden layer, in which the number of hidden units is equal to the

number of training examples. This network can get an error arbitrarily close to zero on the training set, because each hidden unit can learn to recognize exactly one of the training examples! This is the extreme case of overfitting, where the network just memorizes the training data and has an extremely hard time generalizing to other examples.

One remedy for overfitting suggested by this example is to decrease the number of hidden units. In fact, the tendency of a network to overfit the data depends on the number of weights that have to be learned compared to the number of training examples. In fact, as we shall see, the sample complexity of a neural network depends directly on the number of weights to be learned. In addition to removing hidden units, one could decrease the number of weights

by removing edges, or by implementing weight sharing in which different edges use the same weights.

Another remedy is suggested by the following insight. In our example, the training set error (in terms of the error function) will never quite reach zero, because the sigmoid function never quite reaches zero or one at the extremes. However, the error converges closer and closer to zero the larger the magnitudes of the weights. In fact, small weights in neural networks tend to correspond to simpler hypotheses, while overfitting usually involves large weights. This suggests that one might try to use some sort of minimum description length (MDL) criterion and penalize large weights. The error function to be minimized would then be composed of two terms, one describing the training set error and one measuring the complexity of the network as expressed in the magnitudes of the weights.

4.6 Input and Output Encodings

One final point in regard to the implementation and use of back-propagation. A neural network is a function from continuous inputs to continuous outputs. If you want to use it for representing discrete functions, you have to find a way to encode the discrete inputs or outputs. There are several possible encoding methods. One is to use a local encoding, where a single node is used to represent the different possible values of a discrete attribute. Each discrete value is simply mapped to a real number. Another idea is to use distributed encoding, where there is a separate node corresponding to each possible value of the discrete attribute. In representing the value v of the discrete attribute, the node corresponding to v will have value 1, while other nodes will have value 0 or -1. Another possibility is to use a binary encoding, in which the different possible discrete values are represented as binary numbers. In this encoding, the number of nodes used to represent an attribute with n possible values will be $\lceil \log_2 n \rceil$. A value v is represented by taking the binary representation of v, and setting the values of the nodes according to the values of the bits in the binary representation.

In choosing which encoding to use, the main question to consider is how well does that encoding fit the quantity being measured. Suppose the possible classes are 0 to 9. If these classes represent “inflation rate”, a local encoding

is quite appropriate, because the classifications all fall on a linear scale. If

they represent digits in character recognition, a local encoding is inappropriate, because visually 8 is not between 7 and 9. A secondary question to consider when choosing an encoding is how many weights will be required to learn it, and how much data is available.

An issue that arises when using a distributed output encoding is how to classify when there are multiple output nodes, each representing one possible class. One possibility is to simply choose the output node with the highest activation level as the classification, no matter what the actual activation levels. So if all outputs are 0 and one is 0.1, or all are 0.9 and one is 1, that one will be chosen. An alternative is to require that there be one output node with high activation and all the rest are low in order to classify. All other situations are considered ambiguous, and no classification is given. One can use various options between these extremes, and also provide a notion of "confidence" of the classification.