CS181 Lecture 12 — Neural Network Model Selection

1 Selecting the Neural Network Model

There’s a tradeoff in the number of hidden units used in a neural network. With too few hidden units, the hypothesis space might not be rich enough to represent the actual function being learned, which means that the network might underfit the data. On the other hand, more hidden units means more weights to learn, and makes it more likely that the network will overfit the data. This leads to the question: how many hidden units should we use? In addition, for a given number of hidden units, we could choose to remove some of the edges between the different layers — how many edges should we have in our network?

Suppose I say, ”I’m going to use a feed-forward neural network with a single complete hidden layer for my learning task”. I have specified all the details of the model, except for the number of hidden units. The number n of hidden units is a parameter of the model. If I have a set of models, parameterized by some set of parameters, choosing the best value for the parameters is a problem called model selection. Model selection is a problem encountered in many different machine learning frameworks. In general, it can be a very hard problem. Often, there is no better solution than searching through the space of possible models.

For example, for our space of models, we can search over the different possible numbers of hidden units, to find the one with the best performance. This approach requires a validation set. A simple minded algorithm is as follows:

```
n = number of training examples (clearly this n is too large)
PrevError = $\infty$
PrevNetwork = $\emptyset$
Repeat
    Network = network with n hidden units trained on training set
    Error = error of Network on validation set
    IfError > PrevError
        Return PrevNetwork
    Else
        n = n − 1
        PrevNetwork = Network
        PrevError = Error
```

Another, slightly more sophisticated example, is to say ”I’m going to use a feed-forward neural network with a single (not necessarily complete) hidden layer for my learning task”. Now the model space is larger — there are lots of different possible network topologies for a given number of hidden units. Consequently, the model selection problem is more difficult. One algorithm, called optimal brain damage, begins with a large, complete network. In each round, after training, it looks at each of the edges and determines the ones that are least ”salient” to the network, and removes them. Hence the name: “brain damage” means that some of the edges in the network are broken, while “optimal” means that there is an attempt to break the edges in a way that will cause least damage.

There are many details that need to be addressed in implementing such an algorithm. For example, how many weights to remove, and when to stop. The key question is, how does one determine which weights to remove? There are several options:

- Remove edges with very small weights. The problem with this is that it requires the inputs to be normalized, since a small weight with a large input may be more significant than a large weight with a small input. The absolute size of a weight is not a good measure of how important it is.
- For each edge, consider setting its weight to zero, and see the effect on the error function. This measures what we really want — how important is the edge to minimizing the overall error. The problem with this is that a huge amount of computation is required.
- There are various approximations to the effect of removing a weight on the error. The details are beyond the scope of this course — see Bishop, “Neural Networks for Pattern Recognition”. (A particular sophisticated algorithm for determining which edges to remove is called “optimal brain surgeon”.)

2 VC Dimension

Another answer to the model selection problem is derived from computational learning theory. The basic idea is that the complexity of the hypothesis space should be determined by the number of samples available. We should choose a hypothesis space such that there are enough samples to learn hypotheses in the space reliably. The fundamental concept in this approach is the VC dimension. “VC” stands for Vapnik-Chervonenkis, the two people who discovered the concept.

The setting is as follows: X is some instance space, and H is some hypothesis space consisting of Boolean functions from X to a Boolean classification. Let S be a set of points in X. A dichotomy on S is a partition of S into positive and negative instances. If $|S| = n$, then there are of course 2^n dichotomies on S. Now, each hypothesis $h \in H$ represents a dichotomy on S.

Definition 2.1: A hypothesis space H shatters a set S if, for every dichotomy on S, there is a hypothesis in H representing that dichotomy.

Now the question is, how large a set can H shatter?

Definition 2.2: The VC dimension of H, denoted V C(H), is the size of the largest set S shattered by H, if that number is finite, otherwise V C(H) = ∞.

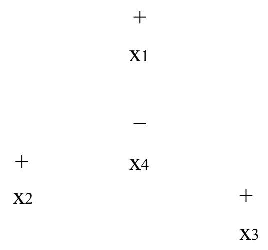
Example 2.3: X is the space of real numbers, H the space of closed intervals. What is V C(H)? The answer is 2. It is clear that any set of two points can be shattered, because we can represent each of the four dichotomies on two points with an interval, as follows:

$$\begin{array}{ccccc}
[\,] & - & & - & \\
 & x_1 & & x_2 & \\
 & & & & \\
 & - & [& + &] \\
 & x_1 & & x_2 & \\
 & & & & \\
[& + &] & - & \\
 & x_1 & & x_2 & \\
 & & & & \\
[& + & & + &] \\
 & x_1 & & x_2 &
\end{array}$$

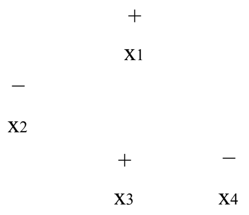
However, a set of three distinct points cannot be shattered. We may assume, without loss of generality, that $x_1 < x_2 < x_3$. The following hypothesis cannot be represented by an interval:

$$\begin{array}{ccc}
+ & - & + \\
x_1 & x_2 & x_3
\end{array}$$

Example 2.4: X is the x, y plane. H is the set of linearly separable hypotheses in the plane, which is the hypothesis space of a perceptron with two inputs. H can obviously shatter 2 points. What about 3? There are two cases to consider. In one case, the three points are collinear, and H cannot shatter them, for the same reason as in the previous example. In the other case, the three points are not collinear, and H can shatter them. So what do we know about the VC dimension at this point? Looking at the definition, we see that V C(H) is the size of the largest set that H can shatter. It does not matter that there are some sets of 3 points that H cannot shatter — V C(H) is at least 3. In fact, it is exactly 3. If we consider sets of 4 points, there are two cases to consider. In one case, one of the points is in the convex hull of the other three. In that case, H cannot represent the dichotomy where the central point differs from the other three:



In the other case, no point is in the convex hull of the other three, so the four points form a quadrilateral in which the four interior angles are all $< 180^\circ$. The diagonals of the quadrilateral intersect inside the quadrilateral. The following dichotomy cannot be represented:



In general, the VC dimension of linear separators in n dimensions, i.e., of perceptrons with n inputs is n + 1.

The VC dimension provides a measure of the complexity of the hypothesis space. One reason it is useful is because it provides us with good bounds on the sample complexity of a hypothesis space. In the lecture on PAC-learning, we gave a general upper bound on the sample complexity in terms of the size of the hypothesis space, but that only worked for finite hypothesis spaces. We also presented a special-purpose argument that works for some infinite hypothesis spaces. Many infinite hypothesis spaces have finite VC dimension, and the bounds presented here will work for them without any special arguments. Not only that, but VC dimension actually gives us lower as well as upper bounds on the sample complexity. The upper bound is

$$\frac{1}{\epsilon}\left(4\log_2\frac{2}{\delta} + 8VC(H)\log_2\frac{13}{\epsilon}\right),$$

while the lower bound is

$$\min\left(\frac{1}{\epsilon}\log_2\frac{1}{\delta}, \frac{VC(H)-1}{32\epsilon}\right).$$

We see from these two bounds that in general, the number of samples needed to reliably learn in a hypothesis space H grows linearly with V C(H).

Can we apply these ideas to neural networks? We already have the VC dimension for perceptrons. Suppose we have a multi-layer network of nodes, where each node behaves like a perceptron, i.e., each node uses a threshold activation function. Each node is capable of distinguishing some points. The question is, what sets of points can the entire network shatter? An answer is

provided by the following theorem. This theorem requires that the network be layered, which means that the nodes can be partitioned into layers in such a way that each edge goes from one layer to the next. Most neural network designs, including all the ones we have considered, are layered.

Theorem 2.5: (Baum & Haussler) Let G be a layered feed-forward network with $s \geq 2$ non-input nodes, such that each node has VC dimension at most d. Then $V C(G) \leq 2ds\log_2(es)$ (e is the base of the natural logarithm).

Consider a network with a single complete hidden layer, with m inputs, n hidden units, and l outputs. Suppose that the network uses threshold units for the hidden nodes. In the notation of the theorem, d is max(m, n) + 1, while s is n + l. If we assume $n \geq m$, which is true as n grows large, we get the bound $V C(G) <= 2(n + 1)(n + l)\log_2(e(n + l))$, which is $O(n^2 \log n)$. Of course, a multi-layer network typically uses sigmoid activation for the hidden nodes, not perceptrons, but the sigmoid function is approximately linear for small weights, so the bound gives a good qualitative idea of how the sample complexity grows as the number of hidden units increases. Also, for non-layered networks, the formula still provides a reasonable guideline as to the VC dimension. It is slightly more than linear in the number of weights. With shared weights, it is the number of actual weights that counts.

3 Discussion

We’ve now seen two major paradigms for supervised learning: decision trees and neural networks. How do they compare to each other? The following table summarizes some of the main points of comparison. It is by no means an exhaustive list.

Decision Trees	Neural Networks
discrete (but can be made continuous)	continuous
few attributes determine decision	all attributes contribute
fast classification	fast classification
fast learning	can take a long time to learn
	robust to noise
	detects hidden features
models easy to interpret	

4 Applications

One of the most famous applications of neural networks, that really put them on the map, was the NetTalk application published in 1987. NetTalk learned how to speak English words. More precisely, the task was to take an English word, and compute the sequence of phonemes used to pronounce that word. For example, given the word “cat” as input, the output is the sequence “k ae t”.

One thing to note about this application is that it is inherently hard in English. The same spelling may have multiple pronounciations. For example, how should “lead” be pronounced? Therefore, one could not expect any algorithm that took only the spelling of individual words as input to produce perfect output.

NetTalk used a neural network with 80 hidden units. The training set consisted of 1024 words. After 50 epochs, NetTalk achieved 95% accuracy on the training set, but only 78% on a test set.

In terms of performance, this is not so remarkable, but the developers of NetTalk had a fantastic demo. They produced a video showing how the performance of the network got better and better after more iterations. It began with babbling, and ended up with something that sounded like understandable speech. To sharpen the effect, they used a high-pitched voice saying “I want to go to grandmother’s house”. It really sounded like a child learning to speak.

This application really made neural networks quite popular, particularly in cognitive science. It combined a physically plausible neural network model with a convincing demonstration of gradually improving behavior.

There have been many applications of neural networks in the area of image processing. One of them — digit recognition — is being explored in your homework.

A similar application is to face recognition. The input is a database consisting of passport photos. Each photo is labeled with the name of the person, as well as other aspects such as which way the person is facing (pose) and whether the person is wearing sunglasses. The network was trained on 260 images with 20 different people appearing in them. Each image was 128x128 pixels, with each pixel taking on a greyscale value between 0 and 255. The network was able to achieve 90% accuracy in determining the pose of a photo, and over 70% accuracy in determining the identity of a person.

Another application of neural networks was to automated driving. A system called ALVINN was developed that could steer on a single lane road. The input to the network was a 30x32 pixel image, and the output was one of 30 steering directions. Training data was obtained by observing a human driver. So really, the function that ALVINN tried to learn was to predict what a human driver would do, given the visual data. ALVINN was able to travel at 70mph for 90 miles. A later system was able to drive “hands free across America”.

One interesting effect is that humans were too good drivers to provide ALVINN with good training data. Because a human would never get into bad situations, there would be no training data to tell ALVINN what to do in a bad situation. As a result, if it got into a bad situation, it was likely to crash. To get around this problem, the designers of ALVINN added skewed images of the road to the training data, to simulate the type of input that would be received in bad situations.

Another interesting point is that ALVINN didn’t need to be able to steer fantastically well. The system could correct errors very quickly, approximately every 10th of a second. As a classifier, the neural network was quick enough to make decisions very quickly. A system that did sophisticated image processing to decide which way to steer might have taken longer to make decisions, and therefore not be able to correct errors quickly enough.

A final area of application of neural networks is in conjunction with reinforcement learning. Recall that when we discussed reinforcement learning, we talked about the fact that if the state space is large, there is no way to represent the Q function or the value function explicitly. Instead, a generalization method is needed, that represents the function being learned more compactly. Neural networks are very commonly used to represent the Q function or the value function. One reason is that these functions are continuous. A famous example of this approach is the TD-Gammon backgammon player, that learned to play backgammon at a world-championship level by playing against itself.

A word of caution is in order. Although the Q-learning and temporal difference algorithms are guaranteed to converge in the long run to optimal policies, (assuming an appropriate exploration policy), this is only true if a complete, explicit representation of the Q function or value function is used. If a neural network is used to approximate the function, the algorithm may not converge. This is actually a serious problem in practice, and at this point researchers are unclear as to how well the method can be expected to perform in general, despite some conspicuous successes. The bottom line is that the method is worth trying, but a lot of engineering may be required to make it work well.