CS181 Lecture 10 — Perceptrons

Today we shift to a new topic, on which we shall spend several lectures: neural networks. The topic is covered in Russell & Norvig Chapter 19, Sections 19.1–19.5. A good in-depth book on neural networks is "Neural Networks for Pattern Recognition" by Bishop.

1 Introduction to Neural Networks

In one sentence, a neural network is a computational structure formed from a network of individual processing units (neurons) that send messages to each other. Neural networks are modeled after the human brain. They are important both as a framework for machine learning and as a model for understanding human intelligence. Some people believe they provide the most promising approach to developing intelligent agents.

In fact, neural networks have been around longer than AI itself. Here's a brief timeline, to provide some cultural and historical perspective.

1943 McCulloch & Pitts inaugrated the study of neural networks by introducing a calculus of weights and threshold units.

1948 Norbert Wiener's cybernetics introduced negative feedback loops as a model for learning.

1949 Hebb developed the first update rule for learning neurons.

1950 Turing proposed his test for determining whether a computer is intelligent.

1951 Minsky build the first working neural net system.

1956 AI ”officially inaugrated” by McCarthy, Minsky, Shannon, Newell & Simon, Samuel and others.

1957 Rosenblatt develops the perceptron (a model for a single processing unit).

1960 Widrow & Hoff develop the adaline (another model for learning a single unit).

1960-1970 Lots of work on building systems of perceptrons and adalines.

1969 Minsky & Papert publish ”Perceptrons”. This book describes the state of perceptron research at that time, but also points out some fundamental limitations of perceptrons.

1969 Bryson & Ho, two applied mathematicians, develop the back-propagation algorithm. This work addresses the limitations of perceptrons, but was not known in the AI and neural network communities.

1970s The “perceptron winter” — loss of interest and funding for neural network research as a result of Minsky and Papert’s book.

1980s There is a resurgence of interest in “connectionism” (as neural network research became known). The back-propagation algorithm is rediscovered by several researchers.

1986 Rumelhart & McClelland’s influential “Parallel Distributed Processing” collection popularizes connectionism in the AI community.

1990s There is a gradual increase in applications and technical understanding of neural networks; by the end of the decade, neural network research is reintegrated with mainstream AI.

2000s Neural networks are viewed as just another machine learning technique, and not necessarily the best technique for many tasks.

1.1 Brain Networks

As we said earlier, neural networks are inspired by the workings of the brain. Studying how the brain works might help us understand how computational intelligence could work. Furthermore, brains are our best example of working intelligent systems, so it makes sense to try to mimic the brain in designing intelligent systems. The study also goes in the other direction: by studying computational models of artificial neural networks, we might develop a better understanding of how the brain works. I will describe very briefly some of the basic features of the brain that are mimicked by artificial neural networks.

The main processing unit in the brain is the neuron, or nerve cell. Some of the parts of a neuron are: the soma or cell body; the axon, a long fibre leading out of the soma; a number of dendrites, shorter fibres branching out of the soma; and synapses that connect the axon to dendrites of other neurons.

Information flows from the soma of one neuron through the axon, and is transmitted to dendrites of other neurons via the synapses. Signals are propogated by electrochemical reactions. When the electric potential in the soma exceeds some threshold, a pulse or action potential is sent down the axon. This triggers a reaction in the synapses, releasing transmitters into the dendrites that alter the electric potential in the next cell. The reactions at the synapses may be excitatory or inhibitory depending on whether they increase or decrease the potential in the next cell.

When we abstract away from the biochemistry, we see that there are two basic features that determine how a network functions. One is the connectivity structure of the network, which specifies which neurons feed into which other neurons. The second is the nature of the synaptic reactions. When both of these are specified, a computational structure is defined, that is capable of information processing.

In addition, both the connectivity structure and the strength of connections can be learned. The strengths of the synaptic reactions are plastic; there can be long-term changes in the strengths of connections based on patterns of stimulation. Also, new connections are often formed, changing the structure of the network itself. As the network changes, so do the computations performed by the network.

It is important to emphasize that neural networks are a micro-level model of the brain. At a higher level, the brain is divided into regions that perform particular functions. Our understanding of what parts of the brain perform different tasks and how they perform it is quite incomplete. Nevertheless, one might still believe that modeling the overall architecture of intelligent agents on the macro-strucure of the brain is a good idea. However, artificial neural networks do not model the higher-level structure of the brain at all. They take the neuron as the model of how information is stored and communicated, but do not follow the brain in determining what information is stored, or how it is organized.

In fact, even on the lower level, artificial neural networks are not exact models of the brain. Neuroscientists tell us that the types of computations performed in the brain are different from the ones performed by artificial networks. Nevertheless, artificial neural networks are an attractive design methodolgy for intelligent systems, because they mimic a number of attractive properties of the brain:

Massive Parallelism The "clock speed" of the brain is rather slow: less than 1 kHz, as compared to several thousand mHz for a modern PC. However, there are about 10^{11} neurons, each of which functions as both a processing unit and a storage unit. This compares favorably with today's computers. One key reason why brains are able to process so much data is that they are massively parallel — all the neurons process the data at once.

Graceful degradation The brain continues to function even as individual neurons die.

Plasticity or ability to adapt, based on experience.

2 Perceptrons

A perceptron is a single processing unit of a neural net. Here’s a general picture of a perceptron, with notation:

```
I1 ---------\
         w1          \
       ...              > in --> g --> OUTPUT
         wn           /
In ---------/
```

A perceptron takes n inputs I_1 to I_n. Each input I_j has an associated weight w_j. The output of the perceptron is produced by a two-stage process. The first stage computes the quantity in, which is the weighted sum of the inputs: $in = \sum_{j=1}^{n} w_j I_j$. The second stage applies an activation function g to in. For perceptrons, the activation function used is the threshold activation function:

$$g(in) = \begin{cases} 1 & \text{if } in > \text{threshold} \\ -1 & \text{otherwise} \end{cases}$$

The threshold is a parameter of the activation function. An alternative way of encoding a threshold activation function that we shall find useful is to create a special input I_0. This input will always be fixed at −1 for every instance. The weight associated with this input can then be used in place of the threshold. Now, we compute $in = \sum_{j=0}^{n} w_j I_j$, and set

$$g(in) = \begin{cases} 1 & \text{if } in > 0 \\ -1 & \text{otherwise} \end{cases}$$

What we have achieved by this trick is to turn the threshold into a weight, so that we can treat it in the same way as the other weights, which are the other parameters of the perceptron.

A perceptron is a classifier, that takes n continuous inputs, and returns a Boolean classification (either +1 or -1). What is the hypothesis space of perceptrons? I.e., what subset of functions from continuous inputs to Boolean classifications can a perceptron represent?

Let us answer this question by analyzing the form of a perceptron. A perceptron on n inputs is defined by the set of weights $w_0,\dots,w_n$. A given set of weights w defines a function in_w on inputs as follows: $in_w(I) = \sum_{j=0}^{n} w_j I_j$. The perceptron then assigns class +1 if $in_w(I)$ is positive, −1 otherwise. Let us look at the points I where $in_w(I) = 0$. This set of points defines a hyperplane in the space of inputs. On one side of this hyperplane, all points will be classed as +1, on the other side they will be classed as −1. A surface that divides the input space into regions of different classification is called a decision boundary.

From this analysis, we see that perceptrons always have a linear decision boundary. A classification function with a linear decision boundary is called linearly separable. Perceptrons can represent linearly separable functions.

We can use a perceptron as a classifier on Boolean inputs if we restrict the inputs to be +1 (for true) and -1 (for false). If we consider the hypothesis space of perceptrons on two Boolean inputs, what subset of Boolean functions on two inputs can they represent? An equivalent question is: what Boolean functions on two inputs are linearly separable?

The answer is that perceptrons can represent many but not all of the Boolean functions on two inputs. For example, a perceptron can represent the and function by setting the weights to be +1 and the threshold to 1.5. Similarly, the or function can be represented with weights of +1 and threshold -0.5. However, the xor function is not linearly separable and cannot be represented. We can see this intuitively by looking at the values of the function in the 2-d plane:

```
                          |
                          |
           +              |          -
                          |
                          |
---------------|-------------
                          |
                          |
                          |
           -              |          +
                          |
                          |
```

We cannot draw a line so that both pluses are on one side and both minuses are on the other. To prove formally that xor is not linearly separable, we can make the following argument. Suppose we can represent xor using weights w. When I_1 and I_2 are both -1, the output is -1, so $-w_0 - w_1 - w_2 \leq 0$. On the other hand, when I_1 is -1 but I_2 is +1, the output is +1, so $-w_0 - w_1 + w_2 > 0$. This shows that w_2 is positive. However, when I_1 and I_2 are both +1, the output is -1, so $-w_0 + w_1 + w_2 \leq 0$, while when I_1 is +1 and I_2 is -1 the output -is -1, so $-w_0 + w_1 - w_2 > 0$. This shows that w_2 is negative, which is a contradication.

In general, as the number n of Boolean inputs increases, the set of linearly separable functions is an increasingly small subset of the complete set of Boolean functions on n Boolean inputs. In fact, the fraction of linearly separable functions decreases exponentially in n. However, there are still many interesting and useful linearly separable functions. An example is the majority function, in which the output is +1 iff the majority of inputs are +1.

Recall that the majority function is extremely difficult for a decision tree to represent. In fact, this function gives us a good characterization of the difference between the inductive biases of decision trees and perceptrons. Decision trees prefer functions that only depend on a small number of inputs, even if those functions are not linearly separable. On the other hand, perceptrons can depend equally on all the inputs, but require that each input votes independently toward the final classification — the inputs cannot interact in interesting ways, as they do in xor. The inductive bias of a learning algorithm gives us a good idea as to when that algorithm might perform well, and we should choose an algorithm whose inductive bias we believe is appropriate to a problem.

3 Learning perceptrons

The perceptron learning algorithm was developed by Rosenblatt in 1957. It is quite simple. First, we define the error of an example d to be the difference between the true output T_d and the predicted output O_d. The error is +2 for an example that was mistakenly classified as negative, 0 for a correctly classified example, and -2 for an example that was mistakenly classified as positive. Assuming the data is linearly separable, we want to find a perceptron that has error 0 on every example.

In order to develop the learning rule, we will consider each of the different cases. Suppose the error was 2, i.e., the correct answer is true, but we predicted false. This means that $\sum_{j=0}^{n} w_j I_j$ was ≤ 0 when it should have been positive. This suggests that we should tweak each of the w_j so as to make $w_j I_j$ larger. If I_j is positive, we should increase w_j, while if I_j is negative we should decrease it.

On the other hand, if the error was -2, $\sum_{j=0}^{n} w_j I_j$ was too large, and we should adjust w_j so as to decrease each of the $w_j I_j$. If I_j is positive, we should decrease w_j, whereas if it I_j is negative we should increase it.

Finally, if the error was 0, the sum was fine, and we should not adjust the weights at all.

We can get the right effect in all cases with the following rule, called the perceptron rule:

$$w_j \leftarrow w_j + \alpha I^d_j (T_d - O_d). \qquad (1)$$

α is a small positive number called the learning rate of the algorithm. What this rule says is that whenever we see a training example, we adjust the weights in such a way as to nudge the sum in the right direction, if it needs to be nudged.

The above rule applies to a single instance. In order to reduce the error on the entire training set D, we may do two things. We may simply apply ??

repeatedly using each of the individual training instances in turn, or we may use the following "batch" version of the rule:

$$w_j \leftarrow w_j + \alpha \sum_{d \in D} I^d_j (T_d - O_d). \quad (2)$$

A perceptron is trained by repeatedly applying the perceptron learning rule. It is a fact that if the training data is linearly separable, and a small enough

learning rate is used, then repeated appliactions of the perceptron learning rule will eventually result in a set of weights that classifies every training instance correctly. On the other hand, if the training set is not linearly separable, the process of repeatedly applying the rule may never converge.

3.1 Gradient Descent and the Adaline Rule

There is an alternative learning rule for perceptrons that addresses this issue, and also forms the basis for the back-propagation algorithm we'll talk about later. This rule, called the adaline rule (also known as the delta rule and the Widrow-Hoff rule) was developed by Widrow and Hoff in 1960.

The adaline rule is based on the idea of gradient descent. Gradient descent is a general algorithm for minimizing a function. It can be applied to many different optimization tasks, not just machine learning. There are many variations and refinements of the gradient descent algorithm, but we'll just look at the basic version.

The task of gradient descent is as follows: given some differentiable function f, that takes an n-dimensional vector $w = [w_1,...,w_n]$ as its argument, find the values of w that minimize f. The idea of the algorithm is very simple. If we imagine the graph of the function f as defining a mountain above the w space, we can try to minimize f by starting from an arbitrary point and then "skiing downhill" as fast as possible. The direction in which to ski is determined by the gradient of the function. The gradient of f at w, denoted $\nabla f(w)$, indicates the direction of steepest increase in the value of f, and is equal to the vector $[\frac{\partial f}{\partial w_0}(w),..., \frac{\partial f}{\partial w_n}(w)]$. So in order to ski downhill, we just compute the gradient, and then adjust the weights in the opposite direction of the gradient. This leads to the following update rule:

$$w \leftarrow w - \alpha \nabla f(w).$$

Gradient descent has a very good property: given a small enough α, it will always converge to a minimum of the error space (in rare circumstances it may reach a saddle point). However, this minimum is not necessarily a global minimum. It may be a local minimum that is a lot worse than the global minimum. This is a serious issue that often needs to be dealt with in practical applications. One solution is random restart: gradient descent is run multiple times, from different random initial starting points, and only the best minimum reached is eventually returned.

Another issue that needs to be dealt with is choosing the learning rate. If the learning rate is too small, the algorithm will take a long time to move from the starting point to the minimum. On the other hand, if the learning rate is too large, the algorithm may oscillate around a minimum point instead of converging. One trick that people use is to begin with a higher learning rate, to get quickly into the region of the minimum, and gradually decrease the learning rate so as to guarantee convergence.

In order to apply gradient descent to learning perceptrons, we need some differentiable function to optimize. A natural candidate is the total squared

error of the perceptron on the training set, but unfortunately this is not a differentiable function — in fact, it is not even continuous, since the absolute error on an instance shifts abruptly from 2 to 0 when the instance becomes classified correctly. Instead, we use the difference between the in value and the target output as a surrogate for the true error, and define the function to be minimized as the total square error (measured this way) on the training set:1

$$E(w) = \frac{1}{2}\sum_{d\in D}(T_d - in_d)^2 = \frac{1}{2}\sum_{d\in D}\Big(T_d - \sum_{j=0}^{n} w_j I_{d_j}\Big)^2$$

Now, to derive the gradient descent update rule, we compute the derivative of this function with respect to each of the weights w_j:

$$\frac{\partial E}{\partial w_j} = \frac{1}{2}\sum_{d\in D} 2(T_d - in_d)\frac{\partial}{\partial w_j}\Big(T_d - \sum_{k=0}^{n} w_k I_{d_k}\Big)$$
$$= \sum_{d\in D}(T_d - in_d)(-I_{d_j})$$

This leads to the following update rule (remembering that we update in the direction of the negative gradient):

$$w_j \leftarrow w_j + \alpha \sum_{d \in D} I_j^d (T_d - in_d) \qquad (3)$$

This is a batch rule. We can also use the rule iteratively, applying it to each of the training instances in turn, without the summation operator.

If we compare Formulas ?? and ??, we see that they are very similar. The only difference is that in_d appears in the adaline rule where the predicted output O_d appeared in the perceptron rule. This small difference leads to a noticable difference in the behavior of the algorithm. Since the adaline rule is performing gradient descent, it will eventually reach a local minimum of the error function. Furthermore, this particular error function is a quadratic with a unique minimum, so we can conclude that the learning process using the adaline rule will in fact converge to the global minimum — even if the data is not linearly separable. This is a big improvement on the perceptron rule, which might not converge if the data is not linearly separable.

Another difference between the two rules lies in the way they treat different incorrectly classified points. In the perceptron rule, all points that are incorrectly classified in the same way have the same value for their predicted output, so they contribute the same amount to the total weight adjustment. By contrast, in the adaline rule, the amount an incorrectly classified point affects the weights depends on the value of $T - in$, i.e., on the degree to which the point was misclassified. Points that are further away from being classified correctly will have more effect on the weights, which seems like good behavior.

A third difference lies in the behavior of the two rules on linearly separable data sets, after a set of weights has been found that classifies the entire training set correctly. For the perceptron rule, all points will now have error 0, so the

[1]The 1/2 appears so we can get rid of it later.

algorithm will stop learning. This is the case even if the separator produced goes very close to one of the points, and only just manages to classify it correctly. We might expect that this separator will be less likely to generalize well than one that leaves a good “margin of error” from each of the training points (we shall see later that this is indeed the case). For the adaline rule, even though all points will have error 0, $T - in$ will generally not be zero, and the algorithm may not yet have reached a minimum in error space. So the algorithm will be able to continue to learn, and perhaps find a better separator, which again is a good thing.

On the other hand, it is quite possible that a model learned using the adaline rule will not classify all the training data correctly, even if the training data is linearly separable. The reason is that the error function minimized by the adaline rule is not the true error function corresponding to correct classification. It is possible that the weights that minimize the adaline rule's error function do not minimize the classification error on the training set.