Massachusetts Institute of Technology

6.034 Artificial Intelligence

Solutions # **10** 6034 Item # **17 Nov 98**

Answer Sheet

Your names: ____________________

Your TA: ____________________

Problem 1 Warmup: perceptrons

As indicated in the text, the weight vector (-10000010) can be used to recognize the digit 0. (Correction: the last number in the weight vector is 0 to replace a threshold, as described in the text; the remaining numbers say whether a particular line segment is turned *on* or *off*, forming the digit shapes 0 through 9).

Part A

Explain in a few sentences why the weight vector (-30000030) must work as well:

Answer:

For the original network with the given weights to work, then $\Sigma_i w_i^1 \times x_i > 0$. But then, a network with weights multiplied by 3 means that $\Sigma_i 3 \times w_i \times x_i > 0$ when $3 \times \Sigma_i \times w_i \times x_i > 0$, which is true exactly when $\Sigma_i w_i \times x_i > 0$. So, the perceptron with altered weights must recognize exactly the same configurations recognized by the ordinary perceptron.

Part B

Suppose that you are given two weight vectors for a digit-recognition perceptron. Both vectors w^1 and w^2 can be used to recognized some particular digit. Suppose that $w^3 = w^2 + w^1$. Can w^3 be used to recognize the same digit? Circle YES or NO, and explain WHY or WHY NOT:

The answer is YES. Since $w^3 = w^2 + w^1$, then $\Sigma_i w_i^3 \times x_i = \Sigma_i (w_i^1 + w_i^2) \times x_i$. Now we separate this latter sum into two parts, $\Sigma_i w_i^1$ and $\Sigma_i w_i^2$.

Because the original weight vectors w^1 and w^2 work, both $\Sigma_i w_i^1$ and $\Sigma_i w_i^2$ are greater than 0 exactly when the input is to be recognized, and less than 0 when the input is not to be recognized. This means that the sum of the two is greater than 0 or less than 0 as appropriate, so their sum w^3 works as well.

Problem 2 Network picture

Part A

On the left, write down the numbers associated with the initial state of each weight and connection link, as given by `*initial-weights*`, before training. On the right, write down the numbers associated with the weights and links *after* training. Please fill in the numbers by writing in the underlined slots provided. Compute the answer the network gives for the input $(1, 1)$, and compare the network's computed output value to that of the actual value. Write both of these results down in the boxes provided.

(WE HAVE OMITTED THIS PICTURE IN THE SOLUTIONS: according to the value for `*initial-weights*`, .26, from input 1; -0.7, from input 2; and threshold .25 feed into neuron 1; 0.4 from input 1; 0.91 from input 2; and 0.36 from the threshold feed into neuron 2; and 0.13 from neuron 1; -0.54 from neuron 2; and a threshold of -0.58 feed into neuron 1, layer 2 (the output neuron).

With an input of 1 to input 1 and 1 to input 2, we plug the numbers in the network as above. Assuming -1 input on the threshold, we have -0.69 feeding out of neuron 1, and 0.95 feeding out of neuron 2. We input these values into the second layer neuron, and get an outut of 1.544

Difference between network and true output= $\boxed{}$

True output should be 0; the computed output is , so the difference is: -1.544.

Part B

Explain why a perceptron (even with weights) could not learn the function learned by this multi-layer network.

WHY NOT:

The given function is Not-XOR (the negation of exclusive or). If we plot the four (x, y) points (0,0), (0,1), (1,0), (1,1) corresponding to the inputs for this function, and note where the $+1$ outputs are, we see that they correspond to (0,0) and (1,1). This region is not linearly separable from the region where the perceptron should report a 0 — there is no single line that divides the plane into two half-planes, with the 1 outputs on one side, and the 0 outputs on the other. Since a perceptron with 2 inputs can only compute such a half-plane (consider its form which is the equation of a half-plane in 2-place, $w_1x_1 + w_2x_2 - w_3 > 0$), a perceptron cannot compute this more complex region.

Problem 3 What difference do initial conditions make?

Part A

Referring to the backpropagation algorithm, explain in simple mathematical terms *why* you see this behavior. (Note: the behavior is not a problem due to "round-off", and this is not the explanation of the phenomenon in question.)

Answer: The network never converges; the rms stays at 0.5 forever. The reason is that the backpropagation algorithm uses the slope (derivative) of the "squashed s" sigmoid function at the currently estimated weights for a given unit as a multiplier to determine how much to change the weights, and in what direction — that is, this is part of the estimated local "gradient" in the gradient descent algorithm.

For "large" values of 200-500, the slope of the sigmoid ($= o(1-o)$ where $o =$ the sigmoid function) is virtually zero, so any "learning" changes dictated by the backpropagation algorithm, which for these units must necessarily be zero. Thus, if we start off an estimated weight at such a "flat" point, the network will change very little if at all from its initial position, and the root mean squared error (rms) never changes. (In the neural net literature, such units are said to be "saturated").

Part B

In a sentence or two, describe the difference between using `*very-tiny-initial-weights*` and `*very-large-initial-weights*` and why, in terms of the backpropagation algorithm, these differences might arise.

For very small weights, convergence is slow, but the system will eventually reach a solution. The difference is that by randomly starting at a point very near zero with small perturbations, and assuming a sigmoid threshold centered on zero, then the inflection point of the sigmoid is also at zero, and there is a chance of moving off in the correct direction, to the right. Given enough iterations (about 3500 in this case), the network will converge on an answer. In contrast, with large enough weights, and a learning rate set to 2.0, there is no chance of ever taking a step large enough in the right direction to make a difference.

Problem 4 Faster, better learning

Part A

Briefly explain *why* the network with the changed learning rate behaves as it does. Hint: look at the "error deltas" — the difference between what the network is predicting as the output vs. the actual, true output desired, as one proceeds from epoch to epoch.

With the learning rate set too high, then the direction of the error gradient can change too much from one training epoch to the next: remember, this is the "stepsize" that the backprop algorithm uses to do its gradient descent search. If this is too large, one can overshoot and move past a local minimum to a region of the search space where the next step points in the opposite direction — perhaps over and over again. The system can oscillate and never converge. Geometrically, the weight change in one training step can be greater than 90 degrees. It is as if the weights are being told in one training (epoch), to "go west" and, at the next iteration, "Go northeast". If this keeps up, the weights won't have a consistent direction to travel in for their search.

Part B

Propose a solution to this problem by means of adjusting an initially high learning rate `*rate*` in the face of such behavior. Please describe your proposed solution in mathematical and/or geometric terms.

One way to detect a too-large learning rate is to measure the actual angle between two successive update error deltas — the directions of the steps the system uses to update its weights during successive passes of the backprop algorithm. If we call these successive "error deltas" d_1 and d_2 then we may recall that the cosine of the angle between the two is defined as $d_1 \cdot d_2/(|d_1||d_2|)$. To see if the angle is greater than 90 degrees, all we have to do is see if this quantity is negative. But this will be so if the dot product of the two vectors $d_1 \cdot d_2$ is negative. If this quantity is negative, one simple approach is simply to halve `*rate` for the next iteration, to see if the oscillation is thereby fixed.

Problem 5 What do the hidden units mean?

Part A

Classify the solution network (=network plus final weights) the system learns with `*new-initial-weights*` above as either a combinational circuit of Type 1 or Type 2 by circling the correct type below, and explain your answer:

Both solutions are essentially of Type II.

Weights to first layer neurons are equal and opposite in sign, corresponding to the negation of one input value and a logical AND; weights to output neuron are equal and of the same sign, corresponding to logical OR, i.e., Boolean addition.)

Part B

Part C

What do these results this suggest about the regions of the weight space for each of the "hidden units" that implement a solution? Your answer should take no more than a few sentences.

A TYPE 1 solution might have the following weights

- Weights to neuron 1, layer 1: (1/2, 1, 1)

- Weights to neuron 2, layer 1: (1/2, 1, 1)
- Weights to output neuron: (3/2, −1, 1)

- The weights leading to each first-layer neuron are all positive, equal, and of the same sign (corresponding to a logical OR) — like Boolean addition.
- The weights leading to the output unit are equal and opposite in sign (corresponding to a logical AND).

TYPE 2 solutions have weights in the form:

- The weights leading to each first layer neuron are equal and opposite in sign, corresponding to the negation of one input value and combination via logical OR.
- The weights leading to the output neuron are equal and of the same sign, corresponding to logical OR.

The bias weights (weights leading into each neuron multiplied by −) are re-scaled depending on what these other weights are — it depends where you start.

The relative frequency of Type 1 solutions vs. Type 2 can be understood from considering the weight constraints described above and the relative probabilities of starting out with weights to neurons 1/2 and the output neuron as either of opposite or the same sign. The answers also depend on the learning rate that is set — in general, neural networks can be sensitive to all of these initial conditions.

In our particular case, there are three basic possibilities for initial weight assignments to the first neuron layer: weights with the same sign or different leading to each neuron, and weights with the same sign and weights with opposite signs leading to the other; and, similarly, two possibilities for initial weight assignments to the output neuron, either weights of the same sign or different signs, for a total of six qualitatively different initial weight assignments. Of these six initial weight region assignments, the one where all weights are the same sign (say, all positive), will have difficulty converging. Of the remaining five, 3 result in Type 1 solutions, and 2 in Type 2 solutions, so this gives a rough ratio of solution types.