Massachusetts Institute of Technology

6.034 Artificial Intelligence

Assignment # **10** Due: **10 Nov 98** 6.034 Item #

PLEASE WRITE YOUR TA'S NAME ON YOUR PAPER.

Goals

- You learn about training neural nets, and how to interpret what the weights and the hidden units "mean".
- You learn about the effects of various modifications to the backpropagation algorithm and to initial conditions for the algorithm.
- You learn how to partly overcome issues that arise due to these modifications.

Code and Answer sheet

- The Scheme code for this problem set is in `/ps10/back.scm`.
- **PLEASE READ. Important note 1: you NEED not run any code unless you want to carry out your own tests of backpropagation, as indicated below. All the neural net simulations and results are already given below.**
- **Important note 2: you should simply tear off the last several pages of this problem set handout (starting at page 13) to use as an answer sheet. Fill in your answers and turn this in. There is NO need for a separate answer sheet.**

Background: Neural networks

Following up on the recent election results and the failure of professional political pundits, you quit school to take a job as a political consultant. Your clients are eager to use neural nets to provide advice. They expect you to be an expert, noting that you were taking 6.034 when you dropped out.

On the first day, they ask you to demonstrate the power of neural nets by training a neural net on some focus-group data. Basically, they have determined, from the focus-group data, that for a certain population, when married people belong to different parties, they do not vote, but when they belong to the same party, they do vote, and they want your neural net to reach the same conclusion on the basis of training data. Accordingly, you start to fool around with a neural net program taken from a 6.034 problem set.

Overview: Neural networks

The file `back.sc` contains the basic neural net learning program. It works by repeatedly changing weights so as to make the network a better predictor of outputs given in a set of input-output samples.

The backpropagation algorithm is used to determine how weights change. That is, the program computes partial derivatives, multiplies them by a "learning" rate constant, and adds the product to the existing weights.

The rate constant is bound to the global variable `*rate*`. The following is a representative example of a rate constant suited to the particular problem you will work on:

```
(define *rate* 2.0)
```

The input-output samples are bound to `*samples*`. The following is a representative example of a sample set for a two-input, one-output net that you decide to work with. The first element represents sample inputs; the second represents sample outputs. The first element of each sample input represents one partner's party (1 means republican, 0 means democrat); the second represents the other partner's party (again, 1 means republican, 0 means democrat). The first and only element of each sample output represents whether the partners vote (1 means yes, 0 means no).

```
                        *-- Inputs (two)
                        |     *-- Output (just one)
                        |     |
                        v     v
(define *samples*  '(((1 0) (0.0))
                     ((0 1) (0.0))
                     ((0 0) (1.0))
                     ((1 1) (1.0))))
```

The structure of the net, and its current state, is determined by an expression bound to `*weights*`. The following is a representative example, showing how structure is encoded for a two layer net. The first element in the list describes the weights between the two inputs and the two neurons in the first layer of neurons. The second element describes the weights between the neurons in the first layer and the second layer.

Each list of weights contains one particular weight, the first, that is taken to be the weight ¡connected to a permanent `-1` value. Accordingly, that weight embodies a neuron's threshold value. The values shown below are those values in the code filed, rounded-off.

;; use other weights

```
(((-.26 .70 -.25)        <-- between inputs and neuron 1,  layer 1
  (-.40 .91 -.36))       <-- between inputs and neuron 2,  layer 1
 ((-.13 -.54 .58))))     <-- between layer 1 and neuron 1, layer 2
```

Normally, you would initialize a neural net by using a random number generator buried in the function `initialize-weights`. So as to assist in debugging, and grading, you are usually asked to work from a given set of initial weights, bound to `*initial-weights*`.

```
(define *weights* *initial-weights*)
```

Training is initiated by calling the `train` function. The argument specifies the maximum number of *training epochs*. For example, the following adjusts the weights bound to `*weights*` for 1000 epochs.

```
(train 1000)
```

In each epoch, the weights are adjusted once for each training sample. Every 25 epochs, `train` prints information about how the training is going in terms of rms error.

As written, the `train` function stops when that maximum is reached or when the root-mean-square error of the net, when working with the training samples, goes below 0.1.

Problem 1 Warmup: perceptrons

Before plunging in on general neural nets, you decide to familiarize yourself with the simpler networks known as perceptrons.

- Read Chapter 23 in the Winston text, on perceptrons, focusing on the recognition of the digits 0–9 represented as seven line segments (see Figures 23.6 and 23.7).

As indicated in the text, the weight vector (-10000010) can be used to recognize the digit 0. (The first number in the weight vector is -1 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. (Please use the answer sheet at the end of this problem set.)

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 digits. Suppose that $w^3 = w^2 + w^1$. Can w^3 be used to recognize the same digit? Answer YES or NO, and explain WHY or WHY NOT. (Please use the answer sheet at the back).

Problem 2 Network picture

Next you turn to multi-layer perceptrons, or neural nets, as described in the introduction to this problem set, using the backpropagation training algorithm `back.scm`.

You run `train` on the training data from the overview above, with the initial weights, `*weights*`, bound to `*initial-weights*`, and a rate factor of 2.0. The results and final weights output from the program are shown below:

```
*initial-weights*
;Value: (((.26 -.7 .25) (.4 .91 .36)) ((.13 -.54 -.58)))

(define *weights* *initial-weights*)

(train 1000)
Epochs: 0  Average rms error: .49937397065063605
Epochs: 25  Average rms error: .499930389656755
Epochs: 50  Average rms error: .4998473899984932
Epochs: 75  Average rms error: .49951664499953463
Epochs: 100  Average rms error: .49877918529765997
Epochs: 125  Average rms error: .4967497145256198
Epochs: 150  Average rms error: .4899893083490814
Epochs: 175  Average rms error: .46804124906367406
Epochs: 200  Average rms error: .42902897184791944
Epochs: 225  Average rms error: .39800974082239193
Epochs: 250  Average rms error: .37903497407727477
Epochs: 275  Average rms error: .3474145055330959
Epochs: 300  Average rms error: .22595639013569552
Epochs: 325  Average rms error: .15311651449912544
Epochs: 350  Average rms error: .119702066172998
Epochs: 375  Average rms error: .10048185111618285
Epochs: 400  Average rms error: .08780574646181707
;Value: #t

*weights*
;Value:
(((3.2720128168482985 -5.445364898669252 5.4509402246392336)
  (2.3871472242075535 4.209238958450872 -4.409431549504485))
 ((-3.007022488348488 -6.061600694283009 -6.171130483682283)))
```

Part A

Use the two figures on the answer sheet at the end of this problem set. 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 down these results down in the boxes provided.

Part B

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

(**Note**: If you decide to run `back.scm` yourself, then in order to avoid spurious Scheme interrupts, you may find it more convenient to break the 600 epochs into units of 100 or 50 each: that is, simply run `(train 100)`, and do so 6 times total; each time you will start the next epoch with the `*weights*` retained from the previous epoch. Be patient! Alternatively, MIT Scheme on PCs (Windows NT or Windows 95) works fine without such interrupts.)

Problem 3 What differences do initial conditions make?

Your political friends, having been a bit bruised by the last election, are still cautious. Experts in sensitivity themselves, sniffing out even the smallest whiffs of political opinion change, they inquire whether your results are solid. What happens, they ask, if network training started off someplace else? You decide to find out.

You modify `*initial-weights*` as follows:

```
(define *weights* *very-large-initial-weights*)
;Value: "*weights* --> ((... ...) (...))"

*weights*
;Value: (((260. -700. 250.) (400. 910. 360.)) ((130. -540. -580.)))

(train 1000)
Epochs: 0  Average rms error: .5
Epochs: 25  Average rms error: .5
Epochs: 50  Average rms error: .5
Epochs: 75  Average rms error: .5
Epochs: 100  Average rms error: .5
```

```
Epochs: 125  Average rms error: .5
Epochs: 150  Average rms error: .5
Epochs: 175  Average rms error: .5
Epochs: 200  Average rms error: .5
Epochs: 225  Average rms error: .5
Epochs: 250  Average rms error: .5
Epochs: 275  Average rms error: .5
Epochs: 300  Average rms error: .5
Epochs: 325  Average rms error: .5
Epochs: 350  Average rms error: .5
Epochs: 375  Average rms error: .5
Epochs: 400  Average rms error: .5
Epochs: 425  Average rms error: .5
Epochs: 450  Average rms error: .5
Epochs: 475  Average rms error: .5
Epochs: 500  Average rms error: .5
Epochs: 525  Average rms error: .5
Epochs: 550  Average rms error: .5
Epochs: 575  Average rms error: .5
Epochs: 600  Average rms error: .5
Epochs: 625  Average rms error: .5
Epochs: 650  Average rms error: .5
Epochs: 675  Average rms error: .5
Epochs: 700  Average rms error: .5
Epochs: 725  Average rms error: .5
Epochs: 750  Average rms error: .5
Epochs: 775  Average rms error: .5
Epochs: 800  Average rms error: .5
Epochs: 825  Average rms error: .5
Epochs: 850  Average rms error: .5
Epochs: 875  Average rms error: .5
Epochs: 900  Average rms error: .5
Epochs: 925  Average rms error: .5
Epochs: 950  Average rms error: .5
Epochs: 975  Average rms error: .5
;Value: #t

*weights*
;Value: (((260. -700. 250.)
          (400. 910. 360.))
         ((130. -540. -580.)))
```

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.)

Part B

The results of the previous section worry you, so naturally you try to see what happens when the initial weights are set to be very small numbers. You reset `*weights*` to the values given by `*very-small-initial-weights*`, as follows, and the call `train` as usual, with the following results.

```
(define *weights* *very-tiny-initial-weights*)
;Value: "*weights* --> ((... ...) (...))"

*weights*
;Value: (((2.6000000000000003e-4 -.0007 .00025)
          (.0004 .00091 3.5999999999999997e-4))
         ((1.3000000000000002e-4 -.00054 -.00058)))

(train 1000)
Epochs: 0  Average rms error: .4999999999999992
Epochs: 25  Average rms error: .5000033472957588
Epochs: 50  Average rms error: .5000134151799465
Epochs: 75  Average rms error: .5000211726587204
Epochs: 100  Average rms error: .5000156575339116
Epochs: 125  Average rms error: .4999847990302526
Epochs: 150  Average rms error: .49990804484952434
Epochs: 175  Average rms error: .4997409104884716
Epochs: 200  Average rms error: .49938184603954094
Epochs: 225  Average rms error: .49858891552576934
Epochs: 250  Average rms error: .4967499913285107
Epochs: 275  Average rms error: .4921901385874044
Epochs: 300  Average rms error: .480058277529503
Epochs: 325  Average rms error: .45123543791655124
Epochs: 350  Average rms error: .4144491293956575
Epochs: 375  Average rms error: .3910890162505197
Epochs: 400  Average rms error: .37875428211533685
Epochs: 425  Average rms error: .3714912767640485
Epochs: 450  Average rms error: .36667017149417214
Epochs: 475  Average rms error: .36320250665260656
Epochs: 500  Average rms error: .3605718618208198
Epochs: 525  Average rms error: .3584992082463595
Epochs: 550  Average rms error: .3568189395383733
```

```
Epochs: 575  Average rms error: .3554259293212078
Epochs: 600  Average rms error: .3542499864123705
Epochs: 625  Average rms error: .3532423467780492
Epochs: 650  Average rms error: .35236802071579043
Epochs: 675  Average rms error: .35160121757293
Epochs: 700  Average rms error: .3509224886397029
Epochs: 725  Average rms error: .35031687712291615
Epochs: 750  Average rms error: .34977268274375417
Epochs: 775  Average rms error: .34928061453551057
Epochs: 800  Average rms error: .34883319620227804
Epochs: 825  Average rms error: .3484243401047837
Epochs: 850  Average rms error: .34804903643967944
Epochs: 875  Average rms error: .34770312273113707
Epochs: 900  Average rms error: .34738311034927916
Epochs: 925  Average rms error: .34708605219526084
Epochs: 950  Average rms error: .3468094405522867
Epochs: 975  Average rms error: .34655112734542926
;Value: #t

*weights*
;Value: (((-9.684660599957758e-3 -5.994163641576111 -5.834529187947255)
          (.20273174584087003 -5.762969785164754 -5.603218796662956))
         ((.6467796630420627 4.46855263384308 4.114249638560575)))
```

Part C

In a sentence or two, describe the difference between using `*very-tiny-initial-weights*` and `*very-large-initial-weights*` and why such a difference might arise, again by referring to the backpropagation algorithm.

Problem 4 Faster, Better, Learning

Professor Hacker drops by and comments that the "learning rate" parameter, described by the variable `*rate*`, dictates "how fast the network will learn — the higher the learning rate, the better." Inspired by this comment, you re-set the learning rate from its current value, 2.0, to a much higher value, 10000.00 — "that will increase the learning rate 5000 times," the Professor cries.

You go ahead and try this experiment, with the following results:

```
(define *weights* *initial-weights*)
;Value: "*weights* --> ((... ...) (...))"
```

```
(define *rate* 10000)
;Value: "*rate* --> 10000"

(train 1000)
Epochs: 0  Average rms error: .49937397065063605
Epochs: 25  Average rms error: .5
Epochs: 50  Average rms error: .5
Epochs: 75  Average rms error: .5
Epochs: 100  Average rms error: .5
Epochs: 125  Average rms error: .5
Epochs: 150  Average rms error: .5
Epochs: 175  Average rms error: .5
Epochs: 200  Average rms error: .5
Epochs: 225  Average rms error: .5
Epochs: 250  Average rms error: .5
Epochs: 275  Average rms error: .5
Epochs: 300  Average rms error: .5
Epochs: 325  Average rms error: .5
Epochs: 350  Average rms error: .5
Epochs: 375  Average rms error: .5
Epochs: 400  Average rms error: .5
Epochs: 425  Average rms error: .5
Epochs: 450  Average rms error: .5
Epochs: 475  Average rms error: .5
Epochs: 500  Average rms error: .5
Epochs: 525  Average rms error: .5
Epochs: 550  Average rms error: .5
Epochs: 575  Average rms error: .5
Epochs: 600  Average rms error: .5
Epochs: 625  Average rms error: .5
Epochs: 650  Average rms error: .5
Epochs: 675  Average rms error: .5
Epochs: 700  Average rms error: .5
Epochs: 725  Average rms error: .5
Epochs: 750  Average rms error: .5
Epochs: 775  Average rms error: .5
Epochs: 800  Average rms error: .5
Epochs: 825  Average rms error: .5
Epochs: 850  Average rms error: .5
Epochs: 875  Average rms error: .5
Epochs: 900  Average rms error: .5
Epochs: 925  Average rms error: .5
Epochs: 950  Average rms error: .5
Epochs: 975  Average rms error: .5
;Value: #t

*weights*
```

```
;Value: (((-83.96352468158129 83.52352468158129 .25)
          (-105.51763873456777 106.82763873456777 .36))
        ((779.1335558993336 -216.22909835908894 -487.30646535983254)))
```

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.

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.

Problem 5 What do the hidden units do?

Still exploring the mysteries of neural nets, you decide to set `*weights*` to a different set of initial weights, as shown below.

```
(define *weights* *new-initial-weights*)
;Value: "*weights* --> ((... ...) (...))"

*weights*
;Value: (((-.26 .7 -.25) (-.4 .91 -.36)) ((-.13 -.54 .58)))
```

The results of training on this new set of initial weights, `(train 1000)`, are as follows:

```
(train 1000)
Epochs: 0  Average rms error: .4995114591982719
Epochs: 25  Average rms error: .49978477709422076
Epochs: 50  Average rms error: .4999051351729767
Epochs: 75  Average rms error: .4997808879326071
Epochs: 100  Average rms error: .49959533828417807
Epochs: 125  Average rms error: .4991877164817323
Epochs: 150  Average rms error: .497958520651512
Epochs: 175  Average rms error: .49302410515285783
Epochs: 200  Average rms error: .470738525195781
Epochs: 225  Average rms error: .4220421938882244
Epochs: 250  Average rms error: .3868982629663781
Epochs: 275  Average rms error: .3694233275818257
Epochs: 300  Average rms error: .3594614695895283
Epochs: 325  Average rms error: .3522425859464214
Epochs: 350  Average rms error: .3436560430931501
Epochs: 375  Average rms error: .3074126207810446
Epochs: 400  Average rms error: .20727217301259485
Epochs: 425  Average rms error: .14225857657176444
Epochs: 450  Average rms error: .11158101324475836
Epochs: 475  Average rms error: .0940376564759476
;Value: #t

*weights*
;Value: (((3.735890989683978 5.994992796481607 -5.83301368895564)
         (-1.7706613203879007 4.1506799196662625 -3.9021861261823805))
         ((2.5337776563941024 -6.393306555948539 5.811117151330713)))
```

Part A

Describe the differences between this set of solution weights and the weights obtained from the previous problem.

Suppose we think of each unit in the network we have designed as computing a Boolean logical combination — a combinational circuit with `and` ($\wedge$) and `or` ($\vee$) gates, along with a negation operator, $\neg$. Further, all inputs are either 0 or 1 (false or true). In terms of our original circuit, addition corresponds to logical `or`; that is, when we add two node inputs together, we take the logical `or` ($\vee$) of their values (the output is 1 if either of the inputs or both is 1). For example, if the two inputs to a node, x_1 and x_2, were 1 and 0, then $x_1 + x_2 = x_1 \vee x_2$. Multiplication corresponds to logical `and`, $\wedge$: $w \times x_1 = w \wedge x_1$ (the output is 1 if and only if both inputs are 1).

Part B

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:

Type 1: $(x \wedge y) \vee (\neg x \wedge \neg y)$

Type 2: $(x \vee \neg y) \wedge (y \vee \neg x)$

Part C

Now consider the network learned starting with `*initial-weights*`, from Problem 2, part (A). Is that network of Type 1 or Type 2? Why?

Part D

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.

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. (The first number in the weight vector is -1 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:

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 digits. 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:

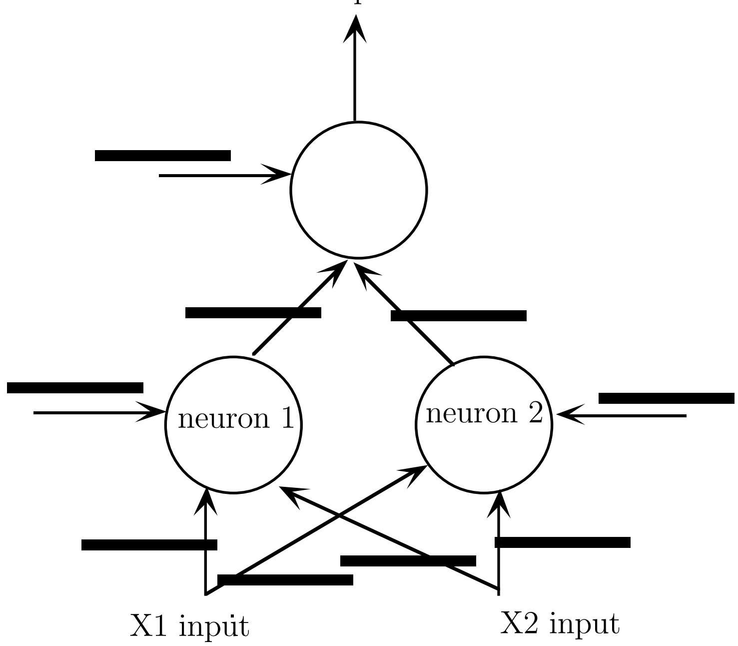
YES NO

WHY or WHY NOT:

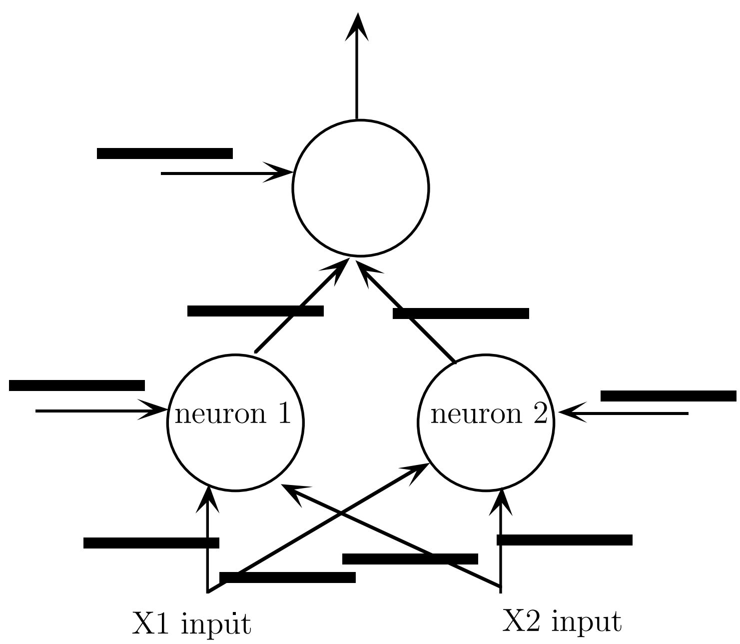
Problem 2 Network picture

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.



weights
before training



weights
after training

Network's output= $\boxed{}$

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

Part B

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

WHY NOT:

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.)

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.

Problem 4 Faster, better learning

Now we run train on the training data, with the initial weights, *weights*, bound to *initial-weights*, and a rate factor of 2.0.

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.

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.

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:

Type 1: $(x \wedge y) \vee (\neg x \wedge \neg y)$

Type 2: $(x \vee \neg y) \wedge (y \vee \neg x)$

Why:

Part B

Now consider the network learned starting with `*initial-weights*`, from Problem 2, part (A). Is that network of Type 1 or Type 2? Why?

TYPE 1 TYPE 2

Why:

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.