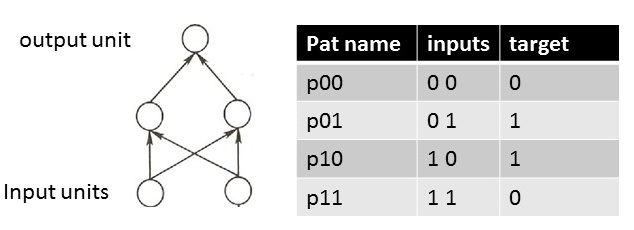
Psychology 209  
Winter 2019  
Homework number 2. Due January 28, 2018

The software for this exercise is written in Python, using the PyTorch neural network package. The homework is predicated on the assumption that you understand the PDP Handbook text for Chapter 5.1. The material below picks up from there. Note that there are slight changes of conventions that have been necessitated by our shift to PyTorch, described as we go through setting up the first exercise. It is discussed on the third page of this homework, just before you proceed into the questions you are asked to answer.

There are two exercises you will run. **Exercise 1** takes you through the XOR problem and is intended to allow you to test and consolidate your basic understanding of the back propagation procedure and the gradient descent process it implements. **Exercise 2** involves exploring the model, and is described later in this handout.

The network architecture is shown in the figure to the right. In this network configuration there are two input units, one for each “bit” in the input pattern. There are also two hidden units and one output unit. The input units project to the hidden units, and the hidden units project to the output unit; there are no direct connections from the input units to the output units. Thus, there are a total of six weights, one in a 2x2 input-to-hidden matrix, and one in a hidden to output matrix. The hidden units and the output unit also have learnable bias weights (not shown).



Architecture of the XOR network used in these exercises, and the training patterns for the XOR problem.

**Running the program**

You will run the exercise in the jupyter notebook. After software installation, you will need to start the jupyter notebook according to the instructions in ‘installing\_software.txt’ that apply to your platform. The key steps are to open a command window, activate the PDPyFlow environment, change directory to the xor top-level directory, and then enter ‘jupyter notebook’ at the command prompt.

Your browser should open a window with a tab called ‘Home’. In this tab, you should find folder called ***xor***. Click on that to open it, and you will see the ***xor\_exercise*** ipython notebook. Click on that and the exercise will open in a new tab. Also click on the ***xor\_visualize*** notebook, to open another tab with the visualization window. If your screen permits, you can move the visualize tab to a new window.

For **Ex. 1**, you will run the program using a pre-specified set of connection weights, so that you will obtain exactly the same results as other students.

To run the homework, you will run the ***xor\_exercise*** ipython notebook. We will briefly discuss the sections of the code that will run one after the other, so you know what will be happening.

First import necessary support packages, create the dataset, the model and trainer objects, and visualization utilities, in the **Import**, **Dataset,** **Model**, and **Visualization** code blocks.

The next code block, called **Run**, runs a single *runset* (where a runset is one or more runs of the model, as we explain below). In each runset, for each run, we create our dataset, create a model, train the model, save the results, and plot the learning curve. You will see that in the `main` function, we specify the number of runs to be included in the runset (`num\_runs`), as well as the parameters of our model, optimizer, and trainer.

For the model, we specify the number of units in the hidden layer of the network (`hidden\_size`; use 2 for Exercise 1), as well as how to initialize the network weights: either using saved parameters (`use\_saved\_params`; set to True for Exercise 1) or by random initialization within a particular range (`weight\_initialization\_range`).

For the optimizer, we specify the learning rate (`learning\_rate`) and momentum term (`momentum`; described in the Handbook text).

Last, we denote the number of epochs for which to train the model (`num\_training\_epochs`), as well as a stopping criterion (`stopping\_criterion`): if the training loss (the total Sum Squared Error, summing across all four patterns), reaches the stopping criterion, training stops early (this corresponds to an average squared error per pattern of .01).

In Ex. 1, the runset only contains one run – in Ex. 2, your runsets will contain several runs, for reasons we will explain below. In any case, for each run in your runset, the **Run** section of the code carries out the initialization of all of the weights and biases in the network (either by restoring then from saved parameters as in Ex 1, or choosing random initial values), sets up queues that present the training examples to the network, then sets up a **for** loop to run a series of training epochs (max, which may terminate early if the stopping criterion is reached). At test points (`test\_freq`) , the set of test patterns will be presented one after the other with learning turned off, so that we can record details of the network’s behavior for detailed visualization an analysis.

You run the network by clicking on the ‘Kernel’ menu item at the top of the notebook, then selecting ‘Restart & Run All’. You will be asked to confirm – to do so, hit enter or click on the red box.

For each run of the network, it will be trained according to the settings of several of the specifiers (often called *hyperparameters*) we mentioned above. The options that are set by default are consistent with the statements in the handbook, Chapter 5. Everyone is using the same set of weights and biases that were initialized many years ago, using a uniform random distribution in the range between -.5 and .5 (use\_saved\_params=True). Note that, during learning, weight error derivatives are accumulated across all patterns in the training set and then the weights are updated once at the end of the epoch (full batch mode training). The learning rate is .25, the momentum is .9, the loss function is the Sum Squared Error, and there is no weight decay. The stopping criterion value for the loss is set to .04. That means that when the total Sum Squared Error, summing across all four patterns, reaches .04, training stops (this corresponds to an average squared error per pattern of .01).

After training finishes, you will see printouts and plots at the bottom of the notebook showing the training progress. This shows the value of the loss (also called *tss,* for total sum of squares) for each training epoch until the error criterion is reached. You will see some strange phenomena – some oscillations at first, followed by a long plateau, then followed by a relatively rapid drop in the *tss* until it falls below 0.04.

To understand what has happened in the network in more detail, you will use the **xor\_visualize** **notebook**. In this notebook, you must specify the correct logdir to read this data from, providing the correct pathname. The pathname is printed when you run the **Run** code block (“Results saved in logdirs/logdir\_XXX”). You will only enter the last two elements of this path in quotes in the xor\_visualize notebook. The first time you run the network, you will see that the results have been placed in logdirs/logdir\_000. Thus, the relevant line in the xor\_visualize window should be:

logdir = ‘logdirs/logdir\_000’

Each time you run the **Run** code block, a new logdir will be created containing the results for that runset. Indeed if you ran the notebook to test your installation, and then run it again to actually do the homework, you will be saving results in a new logdir. In any case***,*** ***whenever you use the visualizer, be careful that you are always visualizing the one you intend to visualize***. Note that multiple runs within a runset will be stored in the same logdir; you select which run you want to visualize in the visualization window. For now we have only one run to worry about.

**Navigating the network visualization window**

The visualization window can display the values of the weights and biases from each run of the network, for each of the separate times the network was tested (in our case, before training and at `test\_freq = 30` epoch intervals, and after the error criterion was reached). It can also display the full set of values computed in processing each of the four input patterns and in backpropagating error based on the corresponding target. The run identifier, relevant if you ran an ensemble of separate runs, can be selected from a pulldown at the top left but we ran only one run in Ex. 1. The network displays state variables (inputs, targets, net (linear layer output), act (activations) and relevant derivatives) for a particular pattern at a time – the pattern to display can be selected near the top right (Initially this will be p00). A horizontal selector allows you to move through the set of saved test steps by clicking at different points along the selector. The Epoch number for the current test step is shown at left below the selector (should be 0 at the first test time point). You can shift to another epoch by clicking on the slider – you have to experiment a bit to find the position that corresponds to a particular test step you want.

OK, now, you can select another pattern using the drop down menu called Pattern. Select p11.

The display shows what happened when pattern p11 was processed before any training occurred. The input units activation values were set to 1 1. This is why they both have activation values of 1.0, shown as a fairly saturated red in the first two entries of the sender activation vector. You can verify these values by hovering over the colored tile. The value is in [] at the right, following the x and y position of the cursor, which you should ignore.

The display may be confusing at first – let’s make sure you understand it. There are two parts to the display itself (image below). The bottom one shows information related to the hidden layer and the top one shows information related to the output layer. Note that the input to the hidden layer is just the input pattern itself, while the input to the output layer is the vector of activations of the hidden units. This vector is shown twice, once as the ***vertical*** vector labeled *a* within the hidden layer and once as the ***horizontal*** vector labeled *input* within the output layer.



With the given activations of the input units, coupled with the values of the weights from these units to the hidden units and the values of the bias terms, the net inputs to the hidden units were set to 0.604 and -0.404, which show up as pale reddish and pale blueish respectively. To be sure you understand the net input, hover over the appropriate weights and biases, record their values, and combine them arithmetically to obtain the net input to each of these units. As previously noted, you can hover over a tile to view its numerical value at the bottom of the window. Plugging the net input values into the logistic function, which you can do in the jupyter notebook code block we have placed below the visualization window, you can check that the program correctly calculated the activation values of these units. For example, if you replace the value 0 in the line that reads “logistic(0)” with the net input to hidden unit 0, which is .604, and run the code block by pressing ctrl-Enter, you will see the resulting activation value which (to three decimal places) is equal to the activation of hidden unit 0 that you will see if you hover over the first element of the column marked **a** in the display (the underlying values are full precision – the values we show are rounded for easier reading). You could also carry out the analogous check to see that the activation of hidden unit 1 (second hidden unit, we count from 0!) has been computed correctly. Given these activations for the hidden units (which are repeated as the input to the output layer, as explained above) coupled with the weights from the hidden units to the output unit and the bias on the output unit, the net input to the output unit is 0.488, showing as a faint red. This leads to an activation *a* of about 0.62, which is much larger than the target value of 0 (neutral gray).

A note about the color scale used: Initially, saturated blue corresponds to -1, saturated red corresponds to +1, and light grey corresponds to 0. Thus, activations range from gray to red (0 to 1), and targets are either grey or red (0 or 1). Weights, biases and net inputs all initially fall in a narrow range, but later on in learning they will go outside this range. You can adjust this range adjusting the V-range slider, but we will leave this value at 1 for now.

We now consider the signals that are used to adjust the connection weights, referring back to section 5.1.2 in the PDP handbook. Since the target is 0 (showing as grey), the error at the output, (i.e., target – activation, or t-a), is -0.62. The loss or squared error (shown as “pattern loss” in the controls section) is the square of this number, about 0.38. During learning we use the derivative of the squared error with respect to the activation of the unit, which is equal to -2(*t-a*). We use the expression[[1]](#footnote-1) *dE/da* to refer to the quantity in the test, but to save space in the display, it is shown there as **a’**. Thus, the value shown for *dE/da* is –(–1.24), or 1.24. The next quantity to the left, *dE/dnet* (shown as **net’**) is equal to *dE/da* times the derivative of the activation function (which is equal to (*a*)(1 - *a*)). Plugging the output unit’s activation value (*a* = .62) into this results in a value of *dE/dnet* equal to about 0.292.

(Note that in the PDP handbook, the term *delta* is used to refer to *-dE/dnet,* the negative of the derivative of the error with respect to the net input.Because we are following PyTorch’s conventions rather than Rumelhart’s, we are keeping track of the error derivative itself, leaving off the extra negation step for now.)

We can now calculate the partial derivative of the error at the output unit with respect to a weight from a given hidden unit to the output unit, represented *dE/dw* (the array of these quantities is labeled **W’** in the figure). This quantity is equal to *dE/dnet* on the receiving unit (denoted with subscript *r*) times the activation of the sending unit (subscript *s*) projecting to the weight:

We can also calculate the quantity, *dE/db*, the partial derivative of the error with respect to the bias, labeled **b’**. Since the bias is equivalent to a weight coming from a unit that is always on, *dE/db* is simply equal to *dE/dnet*.

The propagation of learning signals to the hidden layer depends on the **BP equation** given in the PDP handbook, which we re-write in the following form using our current notation:

**BP equation**

We now have a *dE*/*dnet* term for each hidden unit. How do we use it to determine the gradient for the weight coming to the hidden unit from a unit that projects to it? We simply note that the hidden unit is now the receiving unit and the input unit is now the sending unit. So we apply the rule we had above again, noting that the roles of sending and receiving units have shifted back through the net:

We could also use the BP equation to propagate error back to the units we are now treating as sending units. However, if they are the input units, we do not take that additional step, since these are fixed inputs to the network, rather than values that depend on modifiable connection weights.

We now have an equation for *dE*/*dw* that applies to each weight in the network, and a *dE/db* term corresponding to each bias, as described above. What do we do with this information? When we learn, we perform gradient *decent* – we adjust the weights by an amount proportional to -*dE/dw*, and we adjust the biases by an amount equal to -*dE/db*.

*Full batch learning and momentum.* There are two more points to be aware of before you try to understand what is going on as the network tries to learn.

In this exercise we are using what we will call the *full batch learning method*, which means that adjustment to each weight is performed once per epoch, based on the sum across all of the patterns in the training set of the partial derivative of the error on each pattern with respect to the weight, which corresponds to . Below we will call this the *summed gradient* for each weight, and there is also a summed gradient for each bias weight. We display the summed gradient for the batch of test items for weights and for biases at both the output level and the hidden level, at the extreme right of the visualization window, in the cells labeled **sW’** and **sb’**.

The actual *weight step* taken at the end of epoch *n* is represented in the equation below as . The Δ (delta) signifies that this is a change to the weight, so that the value of the weight after the step is simply what it was before plus the delta:

The expression for the weight step is given below:

The step includes the summed gradient for the batch, scaled by the learning rate *є* (‘epsilon’, specified by `learning\_rate` in the software) and combined with a momentum term here called  (‘alpha’) specified by `momentum` in the software. What is momentum? It is a tendency to continue in the same direction as before, i.e. to make the current weight step go in the same direction as the previous weight step (represented as . To remind you, we are using a `learning\_rate`of .25, and momentum of .9.

Ok, let’s hope this is all clear, and you are ready to answer some questions, based on the information in the network viewer, after processing pattern p11 at epoch 0, prior to any learning in the network.

**Exercise 1**

Prior to class on Jan 24, you should try to obtain tentative answers to Q1.1 through Q1.9 below, and note that many of the Q’s have several parts. Start by trying to answer Q.1.1. We show our answer for this question so you can see the kind of answers we are looking for.

Q.1.1.

Show the calculations of the values of *dE/dnet* for each of the two hidden units for pattern p11, using the activations and weights as given in this initial screen display, and the BP Equation above. Explain why these values are so small, referring to the values of the contributing quantities (Keep explanations as short as possible, e.g. ~100 words).

Answer: To answer this question, we need to use the **BP equation**, which simplifies in our case since there is just one output unit. We show the result for hidden unit 0 (you should do the same calculation for hidden unit 1). The value of hu 0’s *dE*/*dnet* is the product of the value of dE/dnet for the output unit (.292), times the weight from hu0 to the output unit (.272), times the derivative of the activation function for hidden unit 0; this derivative is equal to the product of its activation (.647) times one minus its activation (1-.647). Doing the calculations we obtain:

dE/dnet for hu 0 = .292\*.272\*(.647)\*(1-.647) = .0181

This is close to the value (.0182) you should see in the display for this unit’s dE/dnet. The discrepancy is due to our use of only 3 decimal places in our calculations.

*Explanation:* This value is small because it is the product of a number of values that are all fairly small. The derivative of the activation function for the hidden unit reduces the dE/dnet from the output level by about a factor of four, and so does the connection weight, so what we end up with is a very small learning signal. For hu 1, the result is even smaller mostly because the weight from the unit to the output unit is very small.

At this time point, prior to training, the total loss or *tss* for ‘total sum of squares’ is 1.051 as printed in the panel at the bottom of the **xor\_visualize** window. Step through the four input patterns (p00, p01, p10, and p11) to understand more about what is happening.

Q.1.2.

Report the output the network produces for each input pattern and explain why the values are all so similar, referring to the strengths of the weights, the logistic function, and the effects of passing activation forward through the hidden units before it reaches the output units. Calculate the *tss* from these activations and the target values, showing the four squared terms that add up to the initial *tss*.

Now you are ready to explore learning. Look at the first few epochs of the *tss* graph in the **xor\_exercise** window to see how the value changes, shift over to epoch 30 in the visualization window (step index will be equal to 3, Epoch will show as 30), and step through the four training patterns to answer the next question.

Q.1.3.

(a) The total sum of squares is smaller at the end of 30 epochs, but is only a little smaller. Describe what has happened to the weights and biases briefly, and report the resulting activations of the output unit for each of the four patterns. Why do these output activations result in a lower *tss* than the initial value?

(b) Report the small sizes of the *dE/dnet* value for the second hidden unit after processing each of the four patterns, and explain briefly why they are so small, referring to the weights from the hidden to the output units. You will see by examining the *tss* graph that learning proceeds very slowly from this point. Explain why the weights from the input units to the hidden units will change slowly using the *dE/dnet* values for these units.

(c) Now consider the activation of the second hidden unit and the values of *dE/dnet* from this unit to the output unit, for each of the four training patterns. Report the four pairs of numbers. These numbers are small, but not tiny. Yet the weight from the second hidden unit to the output unit will change very slowly over the next several epochs. Can you explain this? Using the numbers you have reported, show the calculation of ***the summed gradient across the batch*** for this weight (check it against the values shown in the visualization window). Given the current value of the weight, and ignoring momentum, what would the new value of the weight be after the connection adjustment?

Over the next 90 epochs or so (out to epoch 120), you will see that there has been very little further change. Run through the four test patterns at epoch 120 and observe both the values of the activations and the values of *dE/dnet* (we do not ask you to report this, but take a look anyway).

Run through the four patterns again after another 60 epochs (epoch 180), and note that some of the weights in the network have begun to build up. At this point, one of the hidden units is providing a fairly sensitive index of the number of input units that are on. The other is very unresponsive.

Q.1.4.

Explain briefly why the more responsive hidden unit will continue to change its weights from the input units more rapidly than the other unit over the next few epochs. [HINT: Consider the *dE/dnet* terms for these two hidden units for each of the four patterns, noting how they depend on the weights from these units to the output unit.]

Shift forward another 30 epochs. At this point, after a total of 210 epochs, one of the hidden units is now acting rather like an OR unit: its output is about the same for all input patterns in which one or more input units is on.

Q.1.5.

Explain this OR unit in terms of its incoming weights and bias term. What is the other hidden unit doing at this point?

Shift forward another 30 epochs to epoch 240. Testing the four patterns, you will see that the second hidden unit becomes more differentiated in its response.

Q.1.6.

Describe what the second hidden unit is doing at this point, and explain why it is leading the network to activate the output unit most strongly when only one of the two input units is on, referring to the weight from this unit to the output unit.

You will see the *tss* drops very quickly over the next 30 epochs, to about epoch 270.

Q.1.7.

Explain the rapid drop in the *tss*, referring to the forces operating on the second hidden unit and the change in its behavior. With pattern p11, report the value of *dE/dnet* for this hidden unit and the value of *dE/net* for the output unit. They have about the same magnitude, but the opposite sign. Explain the factors that are compensating for each other so that these two magnitudes are about the same.

The value of *tss* drops quickly for a few more epochs then begins to level off, reaching the `stopping criterion` (.04) at epoch 289, and training stops. We consider the XOR problem solved at this point.

Let us characterize the solution the network has found using Boolean logic. Use IN0 and IN1 as Boolean variables corresponding to the two input units; HU0 or HU1 as Booleans for the hidden units; and OUT as a Boolean for the output unit. Compose these with AND, OR and NOT. Write each hidden unit as an expression of the inputs and write the output unit’s value as an expression of the hidden units.

By stepping through the four patterns at the final test point, we see that the first hidden unit comes on when one or both of the input units are on, so we can write: HU0 = IN0 OR IN0

Considering the other Hidden Unit, it comes on when both input units are on, but not when only one of them is on so we can write: HU1 = IN0 AND IN1

Finally, the output unit comes on when HU0 is on and HU1 is NOT on, yielding OUT = HU0 AND NOT HU1

We will use this approach to characterize the solutions the network finds in the first part of Ex. 2.

Q.1.8. **(300 words maximum**)

(a) Summarize the course of learning in terms of the emerging roles of the hidden units in computing the XOR function, referring to the logical expressions as discussed above and their implementation by the weights and biases in the network. (b) Compare the final state of the weights with their initial state. Can you give an approximate intuitive account of what has happened, relating the continuous changes driven by weight change to the emergence of a logic-like computation? How might the initial weights have affected the final outcome?

**Exercise 2: Further explorations in the XOR network**

The second exercise is to explore the effects of changing different things in the XOR model. First we consider the effects of different starting weights, then we consider effects of changing some of the hyper-parameters and other details of the learning procedure, in search of improvements in speed and robustness of solutions found. At the end, you will have a chance to present a brief summary evaluation of the back-propagation algorithm.

*Ex 2.1. Effects of initial weights*

Q.2.1.a

Explore the effect of different random initial weights on the time course of learning and the solution found by the network, recording your results in a table. Summarize the range of outcomes you observe in a brief descriptive paragraph.

To respond to this question, conduct a single runset of 8 runs, each using different random weights, without changing other network parameters. Then, use the *xor\_visualize* notebook to look at the final solutions found for each run. You will report your results from three of these runs by filling in the Homework Table that accompanies this assignment. You may complete it using pencil and paper if you prefer. We have filled in the table with the results from Ex 1. to help you see how to fill in the table.

To run this part of the exercise: Edit the **Run** block of *xor\_exercise* to set ’num\_runs’ to **8** and to set `use\_saved\_params` to **False** (first letter uppercase)**.** Then, re-run the exercise notebook (**Note:** this always means select **Restart & Run All** from the Kernel dropdown). Each run will stop when `stopping\_criterion` is reached or after 500 epochs. We will call reaching the stopping criterion a success. The model reaches criterion within 500 epochs about 3/4 of the time, so typically you would observe 5-7 successes out of 8 runs. If in your case less than all of the runs are successes, re-run the exercise. Repeat until you have 5-7 successful runs.

Now, your successful runs will tend to have qualitatively different solutions. There are many different ones, and we can capture the differences using the Boolean approach discussed above. By looking at the final states of the networks at the end of training you should be able to find 2 runs out of your successes that are different from the solution in Ex 1 and also different from each other. We will focus on these two runs, and on one of your runs that was not successful. (To select runs to focus on: In the *xor\_visualize* notebook, edit the logdir entry in the first code block to correspond to the new log that was created for the runset you are recording, and then re-run the visualize notebook (*remembering what this always means*). For each run: select it from the run log selector, select the last saved step by clicking the right end of the step index selector, and then step through the four patterns using the pattern selector. Repeat this until you have chosen the runs you want to record.)

For your three chosen runs, record the epoch number of the final test (where the loss fell below the `stopping\_criterion` value as shown in the xor\_excersise notebook) in the second column, first row of the set of four rows for the run, as illustrated for the run from Ex 1. If the `stopping\_criterion` value was not reached, record the final loss in this cell instead. Now, let us record the ‘Truth Table’ for each of your chosen three runs, and then characterize the solutions found, using Boolean logical expressions using the conventions we established before[[2]](#footnote-2). Using the visualizer again, for each pattern in each of your chosen runs, record the activation value for each hidden unit and for the output unit (in cases where the network did not converge, the output activation will not be correct for at least one entry). Use 1’s for values greater than .9 and and 0’s for values less than .1; if value is between .1 and .9; record the actual value to 2 decimal places (omit 0 before the decimal, as in .39). Finally, record the Boolean expressions computed by H0, H1 and the output unit in the indicated columns. For the case where the network does not converge, Boolean expressions may not be adequate to describe some units. In this case, write a brief descriptive statement in the set of four cells in the appropriate column. If you need more space, write any additional brief comments under Q.2.1.a in the main body of your homework document, referring to the run by the run identifier in the first column in the table.

Now that you have completed the table (which you should submit as part of your homework), consider the diversity of solutions (and failures to reach a solution). Examine the starting weights from the different runs in the visualizer, and consider whether you see anything that might explain either the solution or the time to reach the solution. Then answer this question:

Q.1.b (**200 words**)

Does the solution the network finds or the time it takes to reach a solution depend on the starting weights? Briefly compare 2 of your runs and discuss how the starting weights might have affected the outcome of learning and/or the time required to reach criterion.

*Ex 2.2. Hyper-parameters and other aspects of the implementation*

From your experiments and observations above, you can see that the network is sometimes quite slow to learn and sometimes seems to get stuck. In the final part of the homework, we invite you to explore how to improve its performance.

Q.2.2

Explore the effect of varying some of the hyper-parameters or other aspects of the network on the time course of learning, and the likelihood of success. See if you can find settings that allow the network to learn quickly and reliably, as measured by the time it takes the slowest run in a runset to reach the stopping criterion value of 0.04. (Detailed instructions for responding to this question are below).

Below (next page) we provide a list of some of the hyper-parameters and other aspects of the network that you can change.

Because there is variability from run to run, change `num\_runs` to 10 (in the **RUN** codeblock), and make sure that **‘use\_saved\_parameters’** is set to **False**. These settings will give a sense of the range of outcomes for a particular choice of hyperparameter settings. You will find the current settings of each of these parameters specified near the top of the **RUN** codeblock under the *PARAMETERS* heading. Make edits to those values in your explorations.

Do as much exploration as you would like, and see if you can find parameters that give a worst case result of less than 200 (that is, all runs succeed, and the slowest finishes in about 200 epochs). Some of you might beat this, but that’s not crucial for a good grade. It would be tedious to report all of your runs, but please report about 5 that you can use to support your conclusions in the second part of the question as stated below. A good approach is to use the default values as a baseline and explore deviating in individual ways before considering combinations; reset what you changed back to the default (provided in a comment for ease of reference) before making further changes to explore one change at a time. However, changes to two parameters might not have simple additive effects. [[3]](#footnote-3)

Your answer to this question should consist of two parts:

1. A listing, in which you provide, for each runset: (a) What the differences were from the default (e.g. ‘learning rate increased to 1.0’, (b) a sentence describing why you changed this and what you expected to see and (c) the result of the worst-case performance for that runset. Your best run is the run with the best worst-case performance. **Highlight this entry in bold.**   
*Definition of worst-case performance:* If all runs in a run set succeed, the worst-case performance is the number of epochs to criterion for the slowest run. A failure to reach criterion in 500 epochs (after epoch 499) is worse than any success, so if one or more runs fails the worst-case performance can be recorded as 499, together with both the number of failures and the worst final loss from all the failures.

2. A brief discussion (**200 words maximum**) of your conclusions about how parameters affect performance and what the important factors are in allowing your network to learn quickly and reliably.

Below we describe the different things you can change and how you can change them. We hope you enjoy experimenting with these things in this part of the assignment!

***Optimization parameters***

**learning rate (`learning\_rate` (.25))**: Use a value greater than 0. Changes of about a factor of 2 should give a feel for how this parameter affects performance.

**momentum (`momentum` (.9))**: Must stay in [0,1); .9 or 0 are the most common choices.

***Initialization and scheduling***

**weight range (`weight\_initialization\_range` [-.5, .5])**: The range of the initial random weights.  Weights are initialized to uniform random values in the range specified.  The upper and lower bound can be different, but the lower bound must be smaller than the upper bound, and one can explore varying the range by keeping the bounds symmetric around 0.

**Batch size (`train\_batch\_size`) (4)):** So far we have accumulated weight error derivatives over all 4 training patterns before changing the weights. You can also explore a mode of learning called ***stochastic gradient descent,*** in which a randomly chosen subsets of patterns are placed in batches smaller than the number of patterns in the training set*.* In this mode, all patterns are presented once per epoch, but the patterns are presented in permuted order and the weights are updated after every batch.  In this mode we still add the loss for all of the batches in the epoch to measure the total loss for the epoch. To explore this, change the `train\_batch\_size` (in the **Run** code block) to be 1 or 2 (value must divide evenly into the total number of patterns).  If you set this variable to 1, the weights are updated after every pattern, something that might make sense in a biological system.

***Model parameters***

**Number of hidden units (`hidden\_size` (2)):** You can explore the effect of changing the number of hidden units.  One is not enough (can you say why?) but what happens when you use more than 2? You can explore this by changing the `hidden\_size` value in the specification within the **Run** code block.

**Hidden Layer Non-linearity (`hidden\_nonlinearity` (‘sigmoid’)):** When backpropagation was first developed by Rumelhart, Hinton and Williams (1986) they tended to rely on the sigmoid or logistic non-linearity for all units in a network. We will retain the sigmoid for the output but allow you to explore other non-linearities for use at the hidden layer. Here you can select from ‘**sigmoid’**, **‘tanh’**, and **‘relu’**. The have defined the sigmoid non-linearity in the background Handbook text for backpropagation. The **tanh** function is similar, but its output ranges from -1 to 1 instead of 0 to 1. The **relu** function simply outputs 0 if its input is less than or equal to zero; if the input it positive, it simply outputs its input.

*Final evaluation*

Q.2.3. (**300 words maximum**)

Write a brief evaluation of the computational strengths and weaknesses of the back propagation learning procedure. For any weaknesses you identify, consider how thy might be addressed.

1. We type ascii *d* in *dE*/*da* instead of *∂* even though this is a partial derivative. Note also that in the PDP Handbook, the factor of 2 is ignored. This also affects the size of the *delta* terms used in the PDP Handbook, which strictly speaking correspond to *–(dE/dnet)/2*. [↑](#footnote-ref-1)
2. It may be necessary to use parentheses in some cases. For example NOT (A AND B) (where A and B are Boolean Variables) inverts the Boolean value of (A AND B) – the expression has value 0 when A and B are both 1, and otherwise it has value 1. It is not the same as NOT A AND NOT B, which has value 1 when A and B are both 0, and otherwise has value 0. NOT A AND B means (NOT A) AND B, but it is better to include the () for clarity. [↑](#footnote-ref-2)
3. **TIP**: It is fairly easy to copy and paste from the notebook output into a document, so you should be able to create a kind of lab notebook for each run set, with a page for each listing the parameter deviations from the defaults, the performance summary printed after each run, and the graph of the loss by epoch so that you have clarity about what you did and what the results were. [↑](#footnote-ref-3)