APPM 4370/5370, Computational Neuroscience: Homework 6

Due Wed. Nov. 17, 2021
by 11:30am (on Canvas)

PAGES 1: MATH EXERCISE; PAGE 2: NO COLLABORATION PROBLEM; PAGE 3: PYTHON EXERCISES.

Math exercises.

1. **Logical operations in multilayer perceptrons.** The perceptron is a feedforward (FF) neural network model popularized by Rosenblatt (1957) and inspired by work of McCulloch & Pitts (1946). Minsky & Papert (1969) proved several shortcomings of *single layer* perceptrons, including the fact that they cannot implement the XOR logical function ($x_1 \oplus x_2 = 1$ if $x_1 = 1$ and $x_2 = 0$ or if $x_1 = 0$ and $x_2 = 1$ but not if both $x_1 = x_2 = 1$). This led to an AI winter in which computer scientists and statisticians largely abandoned the study of neural networks (however, mathematical modeling in neuroscience using neural networks still flourished in the 1970s). This winter did not thaw until the early 1980s when the notion of using backpropagation to train *multilayer* perceptrons became widely know, and people realized mulitlayer perceptrons could be used to approximate any function (universal approximation theorem: Cybenko, 1989).

   Here you will explore a bit about this distinction between single and multilayer networks in a small networks and examine the mechanics of backpropagation by hand.

   (a) Show that the single neuron perceptron $y = H(w_1x_1 + w_2x_2 + \theta)$ cannot implement the XOR function for x_1 and x_2 no matter how the parameters w_1, w_2, and θ are tuned, where $H(x)$ is the Heaviside function. (This can be proved for any increasing transfer function $f(x)$, but you need not show this).

   (b) Design a two layer perceptron network that implements the XOR function. Restrict yourself to two neurons in the *hidden* layer y_1 and y_2 and one neuron in the *output* layer z:

$$\begin{aligned} y_1 &= H(w_{11}x_1 + w_{12}x_2 + \theta_1), \\ y_2 &= H(w_{21}x_1 + w_{22}x_2 + \theta_2), \\ z &= H(J_1y_1 + J_2y_2 + \eta). \end{aligned}$$

   There are multiple solutions to the problem. I suggest you sit down with pen and paper and start with a guess, and then refine your guess iteratively.

   (c) Lastly, how to train the two layer neural network? This does not work as well with Heaviside transfer functions since they are insensitive to change unless inputs are near threshold. We thus turn to the sigmoid $f(x) = 1/(1 + \mathrm{e}^{-x})$ and consider again the network:

$$y_1 = f(w_{11}x_1 + w_{12}x_2 + \theta_1), \tag{1a}$$
$$y_2 = f(w_{21}x_1 + w_{22}x_2 + \theta_2), \tag{1b}$$
$$z = f(J_1y_1 + J_2y_2 + \eta). \tag{1c}$$

   Work out a single step of backpropagation (of all the weights) by hand in the case initially taking the weights $w_{11} = w_{12} = w_{21} = w_{22} = J_1 = J_2 = 1$. Take the case in which $x_1 = x_2 = 1$ and the desired output is $z = 0.05$, so the error is $E = (z - 0.05)^2$. Take a learning rate of $r = 0.5$. Note, only update the weights w_{jk} and J_k not the biases θ_j and η. Remember weight updates take the form

$$w_{jk} \mapsto w_{jk} - r \cdot \frac{\partial E}{\partial w_{jk}}, \qquad J_k \mapsto J_k - r \cdot \frac{\partial E}{\partial J_k},$$

   where the partial derivatives with respect to the current weights are found using the chain rule. Take biases to be $\theta_1 = -1$, $\theta_2 = -1$, and $\eta = -1$. Compare the output z before and after the weight update. Is the error reduced after this step? Could you speed up the learning process by changing the learning rate r?

2. ***<u>NO COLLABORATION PROBLEM!</u>*** **anti-BCM rule in a recurrent network.**

(a) Consider the following model of a self-coupled excitatory network subject to the 'anti-BCM' or 'homeostatic' plasticity rule. The neural population v and its self coupling w are given:

$$\frac{dv}{dt} = -v + wv, \qquad \frac{dw}{dt} = \gamma(1-v)v, \qquad \gamma > 0.$$

Describe what each of the terms in each differential equation means neurobiologically. In particular, what happens for large v versus small $v > 0$ in the w differential equation.

(b) Find the equilibria of the system of differential equations and classify their linear stability.

(c) Sketch the nullclines, fixed point(s), and two example trajectories in the phase plane consistent with your findings in (b). Explain what occurs in the model, and interpret your findings neurobiologically.

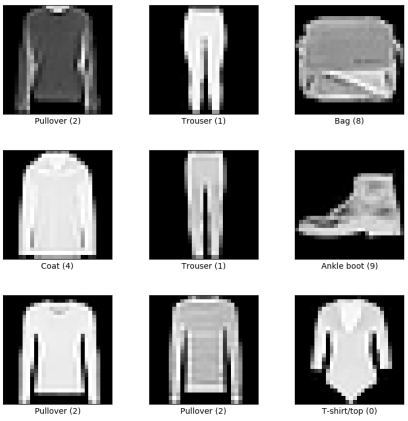
python exercises.

3. **Training the NOT AND function into a two layer network.** Write a python code (you can use numpy or pytorch) to train the weights of the two layer perceptron given in Eq. (1) in Problem 1c to yield a "soft NOT AND" function, so $x_1 = x_2 = 0$ implies $z = 0.95$; $x_1 = x_2 = 1$ implies $z = 0.05$; $x_1 = 1$ and $x_2 = 0$ implies $z = 0.95$; and $x_1 = 0$ and $x_2 = 1$ implies $z = 0.95$. Pick random initial weights and biases (each drawn uniform randomly between -1 and 1). Then train the weights and biases by presenting $\mathbf{x} = (0, 0)$, (1,0), (1,0), (1,1) in succession and backpropagating once based on the error $E = (z - z_{\rm targ})^2$ from the required NOT AND outputs $z_{\rm targ}$, and then repeating this process over and over (in a loop) until a required error tolerance is reached. Note, you should compute the functional form of all the error function derivatives (e.g., $\frac{\partial E}{\partial J_1}$, $\frac{\partial E}{\partial w_{11}}$, $\frac{\partial E}{\partial \theta_1}$) ahead of time and use these in your code.

   (a) Run your code, taking the learning rate $r = 0.5$. Terminate when the maximum error $E = (z - z_{\rm targ})^2$ across all four conditions is less than 0.05 or when you reach 10000 iterations. Plot the maximum error at each iteration as a function of iteration number. Does it always decrease? Report on what you see.

   (b) Run your code 10 times each for the cases where the learning rate is $r = 20$, $r = 10$, $r = 5$, $r = 1$, $r = 0.5$: Make sure you randomize the initial weights and biasesagain each time. Which learning rate leads to the most rapid convergence of the neural network? Explain why you think this is based on what you have learned about feedforward neural networks. What are some ways you could modify this neural network to try and ensure it converges to the desired output function more rapidly?

4. **Train a neural network to classify the MNIST Fashion Dataset.**



In class we have worked thoroughly through learning `pytorch` as a python package for implementing and training neural networks to perform image classification (though it can do much more). We learned how to train and test this on the MNIST handwritten digits data set. In this exercise, you will fill in the blanks on a jupyter notebook `fashion_pytorch_pt4.ipynb` in order to train a neural network to classify greyscale images as different articles of clothing. The reason this image dataset is considered more recently is because it is more difficult and a more natural set of images than handwritten digits.

The jupyter notebook `fashion_pytorch_pt4.ipynb` should provide adequate guidance for this exercise. Provide the outputs of the code and report on what you find by writing a couple of paragraphs in response to each of the following.

(a) You will start by defining the network architecture. Explain how many layers you will use and how many units per layer. In particular, explain what you think each subsequent layer does in a neural network when solving an image classification problem. Also, explain if and why you think classification accuracy might improve as you increase the number of units and layers. Do you think brains are like this? Does increasing the number of neurons and areas in the visual system improve visual computation? Why?

(b) After initializing, you will then train the network, and examine how performance improves as you change the different hyperparameters (learning rate, number of hidden units). Can you get the loss function below 0.4? Give an explanation as to how the changes to the hyperparameters you performed improved the performance of the neural network. Did you notice a speed up in the convergence too with changes to hyperparameters? Try a few different network conformations and choices for the hyperparameters to give a sense how the training run changes.

(c) **Bonus:** If you want to do more, figure out how to load an additional image dataset from torchvision, and see if you can also write code to train this. You may need to add on to the helpers.py file in order to do this, or write your own helper function. Is the dataset you trained on harder or easier to accomplish a high classification accuracy?