Homework 4, Due May 3

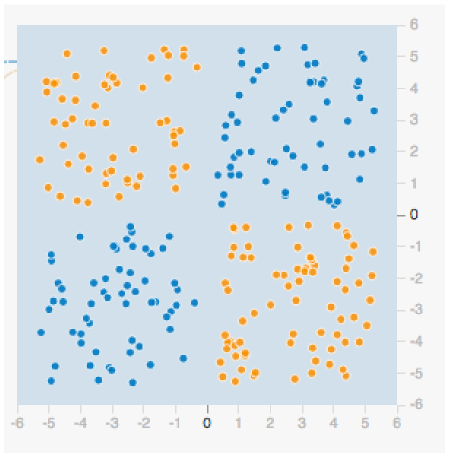
MPCS 53111 Machine Learning, University of Chicago

Practice problem, do not submit

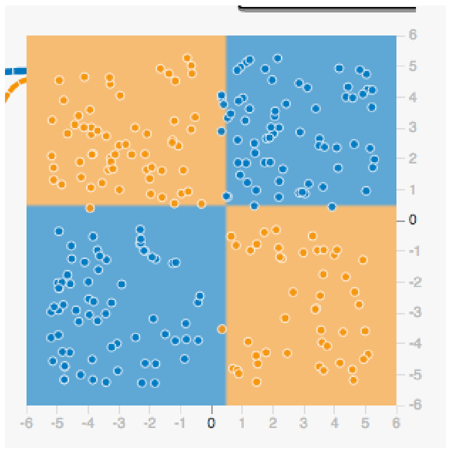
1. Russell-Norvig, Exercise 18.22 (page 767). Here it would be easier to first define a vector corresponding to the weights on the inputs of a node, and a matrix corresponding to the weights on the inputs of nodes in a layer. (See, e.g., Andrew Ng's Coursera course.)

Graded problems, submit

2. Design a neural network that would classify the dataset in Figure 1a, and verify your design at playground.tensorflow.org. The goal is to get crisp decision boundaries as in Figure 1b.



(a) Dataset for classification.



(b) Desired decision boundary.

Hint: Unlike the examples worked out in class, the input values in our dataset are not limited to 0 and 1. So first devise a neural network that outputs 1 for $x_1 > 0.5 + \epsilon$ and 0 for $x_1 < 0.5 - \epsilon$, where ϵ is a very small quantity, such as 0.001.

3. Consider a neural network for a binary classification problem in which the output is a class 0 or 1. The log-likelihood of a dataset of m

examples for a given set of weights $\mathbf{w}$ is—

$$\sum_{i=1}^{m} y^{(i)} \log h_{\mathbf{w}}(x^{(i)}) + (1 - y^{(i)}) \log(1 - h_{\mathbf{w}}(x^{(i)})).$$

(This is similar to Andrew Ng's expression for the log-likelihood in logistic regression, page 18.) Now suppose there is a probability δ that the class label on a training data point has been incorrectly set, and assume it is independent of all other events. Derive the log-likelihood for this scenario. (Verify that your expression reduces to the original expression above when $\delta = 0$.) The cost corresponding to this modified log-likelihood will also be robust to incorrectly labelled data.

4. Russell-Norvig, Exercise 18.23 (page 767). The trick here is that although in gradient descent we differentiate with respect to a weight w, here we differentiate with respect to the output $\hat{y} = h_{\mathbf{w}}(\mathbf{x})$. Briefly explain the reasoning.

5. Implement a Python class `ANN` for an artificial neural network. In particular implement the following methods—

   - `__init__(h, s)`: Initializes an ANN object where `h` is the number of hidden layers and `s` is the number of hidden units in each hidden layer.
   - `fit(X, y, alpha, t)`: Trains the network using back propagation as described in Russell-Norvig (but note the correction below), in which `X` is an (m, n)-shaped numpy input matrix, `y` is an $(m, 1)$-shaped numpy output vector, `alpha` is the training parameter, and `t` is the number of iterations. In particular, use the logistic activation function and the square loss, along with bias inputs.
   - `predict(T)`: Returns the class probabilities for a (q, n)-shaped numpy array `T` of test examples. Assume `T` has the same columns as `X` used in training. The return value should be an (q, k)-shaped numpy array, say `P`, in which k is the number of distinct classes in `y` during training, and `P[i,j]` is the model's probability of example `i` belonging to class `j`.
   - `print()`: Prints the current weights from the input layer to the output layer.

Correction. The back-propagation pseudocode on page 734 of Russell-Norvig contains a small error; namely, the lines initializing the weights should come *before* the **repeat** statement, not after.

Practical considerations. There are some practical considerations when implementing back propagation.

- The first is an easy way to test your gradients. For any weight w you can approximate $\partial J(w)/\partial w$ by

$$\frac{J(w+\epsilon) - J(w-\epsilon)}{2\epsilon,}$$

  in which $J(\cdot)$ is the loss function, and ϵ is a small value.

- Second, you should choose different random weights for each neuron, otherwise two neurons in the same layer will receive the exact same updates and will always have the same weights. Please use the Xavier initialization, where, if w is the weight of a synapse going from a layer with n_{in} nodes to a layer of n_{out} nodes, then w is initialized uniformly at random from the interval

$$[-\sqrt{6/(n_{in} + n_{out})}, \sqrt{6/(n_{in} + n_{out})}].$$

- If you find that your network is giving very strong, incorrect predictions, your value for α may be too large. Also, please work with very small networks during the debugging stage because training can take several minutes.

- Lastly, it is important you use vectorization for efficient code—execute simple operations in parallel between elements of matrices. See numpy tutorial at Stanford or longer numpy tutorial.

The first two are further explained in Andrew Ng's lectures on back propagation in practice.

6. Use your implementation above to build an artificial neural network for the MNIST dataset of handwritten digits. Use the methods you have learned so far to build the best model you can and estimate your generalization error. Avoid testing for networks with more than one

hidden layer, unless your code runs particularly fast. For a reasonable sized network, our code take as long as 15 minutes to train. Describe the steps you have taken in a discussion, and include any relevant plots.

As usual please submit your homework as `.py` and `.txt`/`.pdf` files on SVN.