UNIVERSITY OF ILLINOIS AT URBANA-CHAMPAIGN
Department of Electrical and Computer Engineering

ECE 544NA Pattern Recognition

Solutions 5

Fall 2013

Assigned: Tuesday, October 8, 2013 Due: Thursday, October 17, 2013

Reading: NNPR Chapter 4

Problem 5.1

Given a binary vector $x = (x_1, ..., x_N)$, where $x_i \in \{0, 1\}$, $parity(x)$ is defined by

$$parity(x) = \left(\sum_{i=1}^{N} x_i \right) mod\ 2$$

That is, $parity(x) = 1$ if and only if the input vector x has an odd number of ones.

(a) It is easy to see that $parity(x) = (((x_1 \oplus x_2) \oplus x_3) \oplus ...x_N)$ where $\oplus$ denotes the XOR. The XOR can be implemented as a 2-layer neural network with 9 weights (some of which may be 0). This equivalent definition of parity is sequential, and requires $N - 1$ XORs cascaded together. Hence, the depth of an equivalent architecture is $2(N - 1)$ and the number of weights is $9(N - 1)$. That is, the depth is $O(N)$ and the number of weights is also $O(N)$.

(b) Any boolean function can be implemented as a sum-of-products (OR of ANDs) in two layers; one possible (although trivial) implementation is therefore a sum-of-products expansion of the parity function. The number of weights depends on the number of AND gates used in our representation (since an AND gate can be implemented in a 1-layer neural network with 3 weights). Clearly, the total number of subsets of a set of length N is 2^N; of these, we are interested in the subsets of *odd* cardinality, which is exactly half as many. Hence, the number of AND gates required is $O(2^{N-1})$ and the number of weights is equivalently $O(2^{N-1})$.

(c) A tree-like structure in which $parity(x) = (x_L \oplus x_R)$ where X_L and x_R denote the left and right children of a particular node, respectively. The leaves of the tree would consist of pairs of the input, $(x_1 \oplus x_2), ..., (x_{N-1} \oplus x_N)$. Such a tree would have depth $\log_2 N$ consisting of $N-1$ XOR gates. Hence, the number of weights required is $O(N)$ with a depth of $O(\log_2 N)$. We already see an improvement over (a) since we now have a shallower architecture with the same number of weights.

(d) We show that this can in fact be done in 2 layers. If we decompose the definition of *parity* into different modules, we would first want to compute $\sum_{i=1}^{N} x_i$ and then the *mod* function. We now describe a neural network that computes the following statistic at each hidden node: $m_j = \sum_{i=1}^{N} x_i \geq j$. For any j, if m_j is true, but m_{j+1} is not, we know that $\sum_{i=1}^{N} x_i = j$. To implement the *mod* function, it then suffices to check if m_j is true for *any* odd j. The hidden layer of the neural network is given by: $z_j = \sum_{i=1}^{N} x_i - j + 0.5$, where $j \in \{1, .., N\}$. After passing it through some non-linearity (e.g. $tanh(.)$), we obtain exactly the statistic m_j. Our decision is then based on $y = \sum_{j=1}^{N} (-1)^{j+1} z_j$. We need $O(N^2)$ weights to comppute z_j for $j = 0, 1, ..., N$ and $O(N)$ weights to compute y. Therefore, a depth-2 architecture can be implemented with just $O(N^2)$ weights. If the depth of the network is a constraint, then we see a clear improvement over (b).

(e) By introducing an additional layer that computes the sum $s = \sum_{i=1}^{N} x_i$, we can have an architecture that computes the *parity* function in just 3 layers. In this case, the first layer, denoted by $z^{(1)}$ contains only one node, $z^{(1)} = s$. The second layer computes the statistics m_j described in (c) by $z_j^{(2)} = s - j + 0.5$. Again, parity is computed as in (c): $y = \sum_{j=1}^{N} (-1)^{j+1} z_j^{(2)}$. The first, second, and third layers all require $O(N)$ weights and biases, and we have a depth 3 architecture with $O(N)$ weights.

(f) In most real settings, we want to *learn* a function from data, rather than specify it. We evaluate these approaches based on two criteria: computational complexity and number of local optima. As the number of hidden layers increases, the number of local optima increases expponentially (at least trivially from weight-space symmetry). Deep architectures therefore increase the risk of getting stuck in local optima. However, deeper representations are almost always more succinct – i.e., they require fewer number of parameters to express complex relationships. We saw that (a), (c), and (e) all require $O(N)$ weights. Shallow architectures, on the other hand, require many more parameters – sometimes exponentially many (b) – to represent the same function. Since the computational complexity directly depends on the number of weights, shallow architectures are costlier to train (but they are more likely to converge to good optima). A third criterion to consider is sensitivity to perturbations. That is, how close do the learnt weights have to be to the true value? Such an analysis is difficult, but some intuition is provided in Bengio & Lecun.

(g) Under the assumption that N can be quite large, an algorithm for which the number of weights is $O(N)$ is essential. Our solution to (e) – a minimum depth architecture (3 layers) with $O(N)$ weights – therefore seems to be the best choice.

Matlab Exercises

Problem 5.2

Figure 1 reports the average classification error on 100 random trials for the four methods from HW 4, as well as a 3-layer network with h hidden nodes, $h \in \{2, 4, 8, 16\}$.

(b) Intuitively, the hidden layer computes a nonlinear representation of the data in a supervised fashion (i.e. given the class labels). When $h < 8$, we are indeed reducing the dimensionality and computing a more succinct representation. When $h > 8$, we are projecting onto a higher dimensional space, which can have the effect of making the data more linearly separable. That $h = 2$ performed the best (and better than a single layer with $tanh(.)$) indicates that the data is actually well represented in some lower-dimensional space. As h increases, we see a gradual (although not significant) decrease in performance; hence the additional representation power does not help, and may even hurt given that we only have limited data and many more parameters to learn.

(c) Although still worse, a network with $h = 2$ hidden nodes is closer to linear least squares. Extensive tuning, better stopping criteria, and an adaptive η may improve the performance of these methods; but unless the improvement is significant, the heuristic nature of training a neural network is a major disadvantage. It is always good practice to therefore check how well linear classifiers perform on a particular dataset. In many real world applications, however, we tend to have almost an infinite supply of training data and deeper networks offer significant improvement. See the papers (posted online) on speech and vision applications.

Table 1: Average classification error across 100 random trials

classifier	average error	standard deviation
least squares	**0.23**	**0.03**
1-nearest neighbor	0.29	0.03
tanh(.) nonlinearity	0.28	0.04
perceptron	0.35	0.08
h=2	**0.26**	**0.04**
h=4	0.27	0.05
h=8	0.27	0.05
h=16	0.28	0.05

```
function results =  runexps(numtrials)
%runexps(numtrials) runs numtrials independent experiments for this
%problem set
%results contains results for 1) KNN, 2) least squares, 3) gradient descent
%with tanh(.) nonlinearity, and 4) the perceptron algorithm
results = zeros(8, numtrials);
load diabetes_normalized.mat
diab_labels = diabetes_normalized(:,1);
diab_features = diabetes_normalized(:,2:9);
numsamples = length(diab_labels);
numtrain = ceil(numsamples*0.8);
numtest = numsamples - numtrain;

for i = 1:numtrials
    randinds = randperm(numsamples); %random partitioning of the entire dataset
```

```
traininds = randinds(1:numtrain); %first 80% is training data
testinds = randinds(numtrain + 1: numsamples); %next 20% is test data

mytraindata = diab_features(traininds,:);
mytraindatabias = [mytraindata ones(numtrain, 1)]; %include the bias term

mytestdata = diab_features(testinds, :);
mytestdatabias = [mytestdata ones(numtest, 1)];

y_train = diab_labels(traininds);
y_test = diab_labels(testinds);
%Linear classifier
%learn the classifier
w_linear = ((mytraindatabias'*mytraindatabias)) \ (mytraindatabias'*y_train);
%classify test points
y_linear = 2.*(mytestdatabias*w_linear >= 0) - 1;
%compare with true labels
results(1,i) = sum(y_linear ~= y_test)/length(y_test); %compute and record the error

%Knn classifier with k = 1
y_1nn = knn_classify(1, mytestdata, mytraindata, y_train);
results(2,i) = sum(y_1nn ~= y_test)/length(y_test);

%gradient descent with tanh(.) activation function
w_tanh = graddesc(mytraindatabias, y_train, 0.001, 0.28);
y_tanh = 2.*(mytestdatabias*w_tanh >= 0) - 1;
%compare with true labels
results(3,i) = sum(y_tanh ~= y_test)/length(y_test); %compute and record the error

%the perceptron algorithm (hinge loss)
w_perc = perceptron(mytraindatabias, y_train, 0.001, 0.28);
y_perc = 2.*(mytestdatabias*w_perc >= 0) - 1;
%compare with true labels
results(4,i) = sum(y_perc ~= y_test)/length(y_test); %compute and record the error

%backprop with h = 2
[W1, W2] = backprop(mytraindatabias, y_train, 0.01, 0.30, 2);
a1 = mytestdatabias*W1;
z1 = tanh(a1);
z1 = [ones(numtest, 1) z1];
a2 = z1*W2;
y_backprop = 2.*(a2 >= 0) - 1;
%compare with true labels
results(5,i) = sum(y_backprop ~= y_test)/length(y_test);

%backprop with h = 4
[W1, W2] = backprop(mytraindatabias, y_train, 0.01, 0.30, 4);
a1 = mytestdatabias*W1;
z1 = tanh(a1);
z1 = [ones(numtest, 1) z1];
a2 = z1*W2;
```

```
    y_backprop = 2.*(a2 >= 0) - 1;
    %compare with true labels
    results(6,i) = sum(y_backprop ~= y_test)/length(y_test);

    %backprop with h = 8
    [W1, W2] = backprop(mytraindatabias, y_train, 0.01, 0.30, 8);
    a1 = mytestdatabias*W1;
    z1 = tanh(a1);
    z1 = [ones(numtest, 1) z1];
    a2 = z1*W2;
    y_backprop = 2.*(a2 >= 0) - 1;
    %compare with true labels
    results(7,i) = sum(y_backprop ~= y_test)/length(y_test);

    %backprop with h = 16
    [W1, W2] = backprop(mytraindatabias, y_train, 0.01, 0.30, 16);
    a1 = mytestdatabias*W1;
    z1 = tanh(a1);
    z1 = [ones(numtest, 1) z1];
    a2 = z1*W2;
    y_backprop = 2.*(a2 >= 0) - 1;
    %compare with true labels
    results(8,i) = sum(y_backprop ~= y_test)/length(y_test);

end
```

The back-propagation algorithm:

```
function [W1, W2] = backprop(traindata, trainlabels, eta, threshold, h)
%Given the training data and labels, and a test dataset,
%backprop trains a 3-layer feedforward network with h hidden layers
%and tests it on testdata
[numsamples, numfeats] = size(traindata);
%initialize weights
W1 = randn(numfeats, h);
W2 = randn(h + 1, 1); % one output label (+1 or -1) and a bias term (h + 1)

while 1
    %Note that this can be vectorized (and probably should be in Matlab)
    %but I simply did not have the time to work it out, sorry.
    classifyerror = 0;
    for i = 1:numsamples
        a1 = traindata(i,:)*W1; %forward prop layer 1
        z1 = tanh(a1); % apply the nonlinearity
        %repeat
        a2 = [1 z1]*W2; %introduce bias
        z2 = tanh(a2);

        %back prop
        delta2 = 2*(z2 - trainlabels(i))*(1 - z2^2);
        delta1 = delta2*(W2(2:h+1)'.*(1 - z1.^2))';
```

```
        %compute the gradients
        grad2 = delta2 * [1 z1];
        for j = 1:h
            grad1(j,:) = traindata(i,:)'.*delta1(j,:);
        end
        %update weights
        W1 = W1 - eta*grad1';
        W1 = W1./norm(W1); %normalize
        W2 = W2 - eta*grad2';
        W2 = W2./norm(W2); %normalize

        theclass = 2*(a2 >= 0) - 1;

        classifyerror = classifyerror + (theclass ~= trainlabels(i));
    end
    classifyerror = classifyerror/numsamples;

    if classifyerror < threshold
        break;
    end

end
```