UNIVERSIDADE DE AVEIRO
DEPARTAMENTO DE ELECTRÓNICA TELECOMUNICAÇÕES E INFORMÀTICA

Machine Learning (2018/19) – Lab work 5

Part 1 Neural Networks training

Objectives: Implementation of the backpropagation algorithm for neural network (NN) training to the task of hand-written digit recognition.

First, you need to download the files to the directory where you wish to complete the exercise.

Files included in part 1
ex4.m - Octave/MATLAB script that steps you through this exercise (the main program)
ex3data1.mat - Training set of hand-written digits
displayData.m - Function to help visualize the dataset
fmincg.m - Function minimization routine (similar to fminunc)
ex4weights.mat – Initial Neural Network parameters
predict.m - Neural network prediction function (take this function from lab work 4)
sigmoidGradient.m - Compute the gradient of the sigmoid function **(you need to complete this function)**
randInitializeWeights.m - Randomly initialize weights
nnCostFunction.m - Neural network cost function with regularization term. Backpropagation algorithm

1.1 Visualizing the data

In the previous lab work, you implemented feedforward propagation for neural networks (NN) and used it to predict handwritten digits with pre-trained weights. In this exercise, you will implement the backpropagation algorithm to learn the parameters for the neural network with the same data file (randomly chosen examples of hand-written digit images are shown in Fig.1)

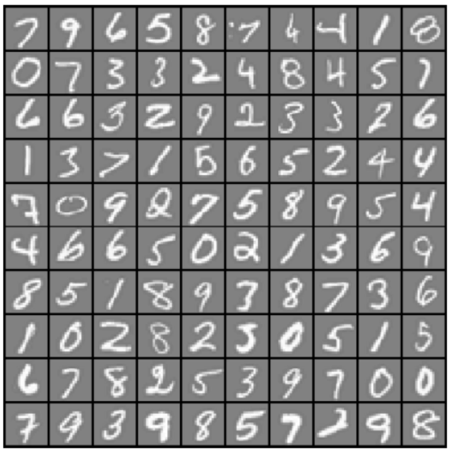


Fig.1 Examples from the dataset

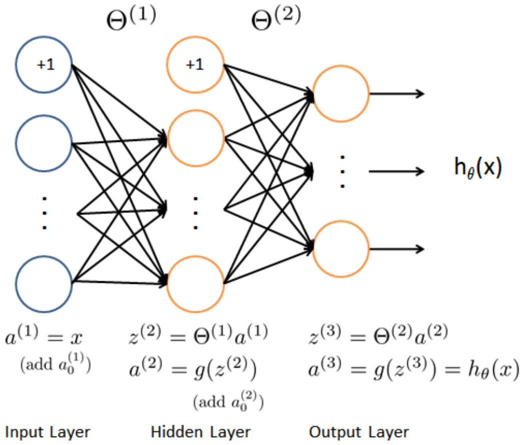


Fig. 2 Neural Network (NN) model

1.2 Model representation

The NN is shown in Fig. 2. It has 3 layers (an input layer, a hidden layer and an output layer). Recall that the inputs are all pixel values of the digit image. Since the images are of size 20x20, this gives 400 input layer units (excluding the extra bias input unit =1). As before, the training data will be directly loaded into the variables X and y. In file *ex4weights.mat* are stored pre-trained network parameters and will be loaded as matrices *Theta1* and *Theta2*. The parameters have dimensions that are sized for a neural network with 25 units in the second (the hidden) layer and 10 output units (corresponding to the 10 digit classes, 1,2, … 9, 0). *Theta1* has size 25x401 and *Theta2* has size 10x26.

1.3. Sigmoid gradient

Now you need to complete the sigmoid gradient function *sigmoidGradient.m*. The gradient for the sigmoid function can be computed as

$$g'(z) = \frac{d}{dz}g(z) = g(z)(1-g(z))$$

where

$$\mathrm{sigmoid}(z) = g(z) = \frac{1}{1+e^{-z}}.$$

When you are done, try testing a few values by calling *sigmoidGradient(z)* at the Octave/MATLAB command line. For large values (both positive and negative) of z, the gradient should be close to 0. When z = 0, the gradient should be exactly 0.25. Your code should also work with vectors and matrices. For a matrix, your function should perform the sigmoid gradient function on every element.

1.4 Feedforward and cost function

The cost function and the gradient for the NN are computed and returned by *nnCostFunction.m*. In this example the NN has 3 layers (an input layer, a hidden layer and an output layer), however, the code in *nnCostFunction.m* works for any number of input units, hidden units and outputs units. Recall that whereas the original labels (variable y) are 1, 2, ..., 10, for the purpose of NN training, the labels are recoded as vectors containing only values 0 or 1.

The cost function for neural networks without regularization is given by

$$J(\theta) = \frac{1}{m}\sum_{i=1}^{m}\sum_{k=1}^{K}\left[-y_k^{(i)}\log((h_\theta(x^{(i)}))_k) - (1-y_k^{(i)})\log(1-(h_\theta(x^{(i)}))_k)\right]$$

The cost function for neural networks with regularization is given by

$$J(\theta) = \frac{1}{m}\sum_{i=1}^{m}\sum_{k=1}^{K}\left[-y_k^{(i)}\log((h_\theta(x^{(i)}))_k) - (1-y_k^{(i)})\log(1-(h_\theta(x^{(i)}))_k)\right] + \frac{\lambda}{2m}\left[\sum_{j=1}^{25}\sum_{k=1}^{400}(\Theta_{j,k}^{(1)})^2 + \sum_{j=1}^{10}\sum_{k=1}^{25}(\Theta_{j,k}^{(2)})^2\right].$$

Using the loaded set of parameters for Theta1 and Theta2, the main script first calls *nnCostFunction* with *lambda*=0 (unregularized cost) and gets a cost of about 0.287629 and then calls *nnCostFunction* with *lambda* =1 and gets a cost of about 0.38377. Run this part of the code to verify these values.

1.5 Random initialization

When training neural networks, it is important to randomly initialize the parameters. One effective strategy for random initialization is to randomly select values for theta uniformly in the range [R_{init},-R_{init}]. This strategy is implemented in the function *randInitializeWeights.m*. In this case R_{init} = 0.12^2. This range of values ensures that the parameters are kept small and makes the learning more efficient.

1.6 Error Backpropagation

Now, you will run the backpropagation algorithm to train the neural network by minimizing the cost function *J* using the advanced optimization function *fmincg*. The intuition behind the backpropagation algorithm is as follows. Given a training example *($x^{(t)}$, $y^{(t)}$),* the backpropagation first runs a “forward pass" to compute all the activations throughout the network, including the output value of the hypothesis *h(x).* Then, for each node *j* in layer *l*, it computes an “error term" $\delta_j^{(l)}$ that measures how much that node was “responsible" for any errors in the NN output. For an output node, we can directly measure the difference between the network's activation and the true target value, and use that to define $\delta_j^{(3)}$ (since layer 3 is the output layer). For the hidden units, $\delta_j^{(l)}$ are computed based on a weighted average of the error terms of the nodes in the layer *(l + 1).* The backpropagation algorithm is schematically represented in Fig. 3 and Fig.4. Steps 1 to 4 are implemented in a for-loop that processes one example at a time. Step 5 will divide the accumulated gradients by *m* (the number of examples) to obtain the gradients for the NN cost function.

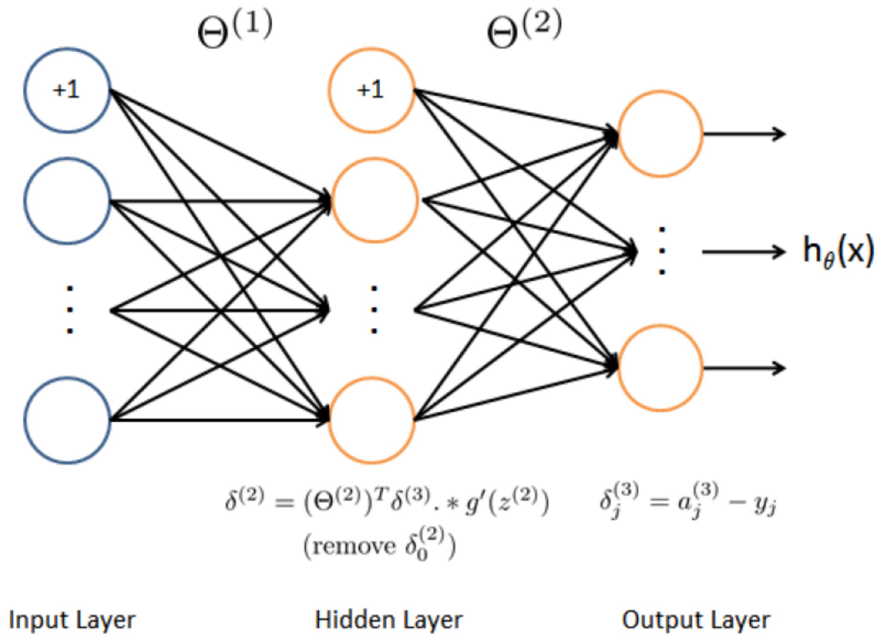
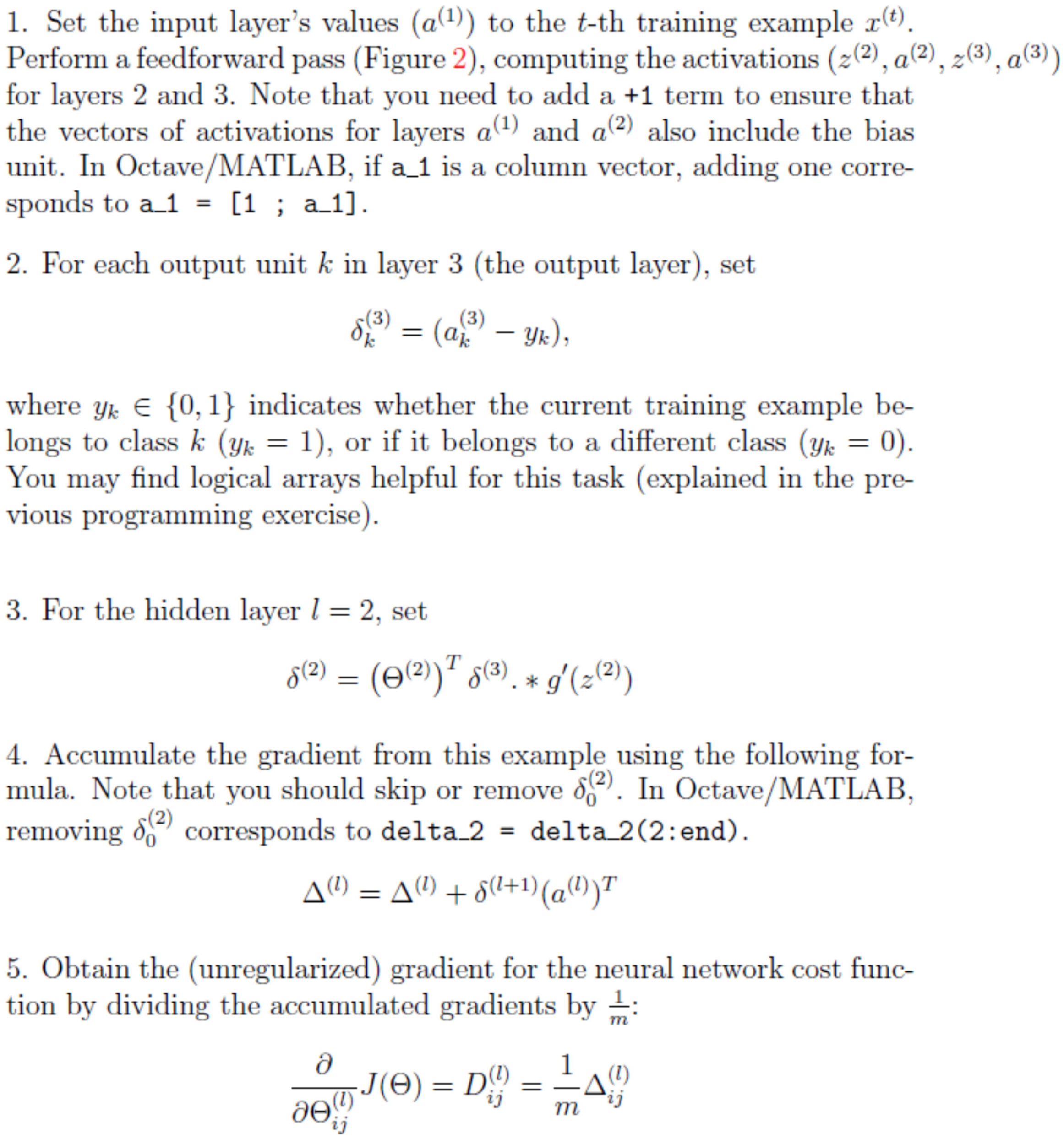

$\Theta^{(1)}$
$\Theta^{(2)}$
+1
+1
$h_\theta(x)$
$\delta^{(2)} = (\Theta^{(2)})^T\delta^{(3)} .* g'(z^{(2)})$
(remove $\delta_0^{(2)}$)
$\delta_j^{(3)} = a_j^{(3)} - y_j$
Input Layer
Hidden Layer
Output Layer

Fig. 3 Backpropagation update



1. Set the input layer's values ($a^{(1)}$) to the t-th training example $x^{(t)}$. Perform a feedforward pass (Figure 2), computing the activations ($z^{(2)}, a^{(2)}, z^{(3)}, a^{(3)}$) for layers 2 and 3. Note that you need to add a `+1` term to ensure that the vectors of activations for layers $a^{(1)}$ and $a^{(2)}$ also include the bias unit. In Octave/MATLAB, if `a_1` is a column vector, adding one corresponds to `a_1 = [1 ; a_1]`.

2. For each output unit k in layer 3 (the output layer), set

$$\delta_k^{(3)} = (a_k^{(3)} - y_k),$$

where $y_k \in \{0, 1\}$ indicates whether the current training example belongs to class k ($y_k = 1$), or if it belongs to a different class ($y_k = 0$). You may find logical arrays helpful for this task (explained in the previous programming exercise).

3. For the hidden layer $l = 2$, set

$$\delta^{(2)} = \left(\Theta^{(2)}\right)^T \delta^{(3)} .* g'(z^{(2)})$$

4. Accumulate the gradient from this example using the following formula. Note that you should skip or remove $\delta_0^{(2)}$. In Octave/MATLAB, removing $\delta_0^{(2)}$ corresponds to `delta_2 = delta_2(2:end)`.

$$\Delta^{(l)} = \Delta^{(l)} + \delta^{(l+1)}(a^{(l)})^T$$

5. Obtain the (unregularized) gradient for the neural network cost function by dividing the accumulated gradients by $\frac{1}{m}$:

$$\frac{\partial}{\partial \Theta_{ij}^{(l)}} J(\Theta) = D_{ij}^{(l)} = \frac{1}{m}\Delta_{ij}^{(l)}$$

Fig. 4 Backpropagation algorithm

1.7 Regularized Neural Networks

The regularization term is added as an additional term after computing the gradients using backpropagation. Note that the algorithm does not regularize the terms regarding the bias. For the matrices Theta1 and Theta2, this corresponds to the first column of each matrix.

$$\frac{\partial}{\partial \Theta_{ij}^{(l)}} J(\Theta) = D_{ij}^{(l)} = \frac{1}{m}\Delta_{ij}^{(l)} \qquad \text{for } j = 0$$

$$\frac{\partial}{\partial \Theta_{ij}^{(l)}} J(\Theta) = D_{ij}^{(l)} = \frac{1}{m}\Delta_{ij}^{(l)} + \frac{\lambda}{m}\Theta_{ij}^{(l)} \qquad \text{for } j \geq 1$$

The next step is to use *fmincg* to learn a good set of parameters. After the training completes, the ex4.m script will report the training accuracy of the classifier by computing the percentage of examples it got correct. You should see a reported training accuracy of about 95.3% (this may vary by about 1% due to the random initialization). It is possible to get higher training accuracies by training the NN for more iterations. Try training the neural network for more iterations (e.g., set *MaxIter* = 400) and also vary the regularization parameter lambda. With the right learning settings, it is possible to get the neural network to perfectly fit the training set (above 99.5%) .

1.8 Visualizing the hidden layer

One way to understand what the neural network is learning is to visualize the representations captured by the hidden units. For the neural network you trained, notice that the *i*th row of *Theta*$^{(1)}$ is a 401-dimensional vector that represents the parameter for the *i*th hidden unit. If we discard the bias term, we get a 400 dimensional vector that represents the weights from each input pixel to the hidden unit. Thus, one way to visualize the representation captured by the hidden unit is to reshape this 400 dimensional vector into a 20x20 image and display it. The next step of ex4.m does this by using the *displayData* function and it will show an image similar to Fig. 5 with 25 units, each corresponding to one hidden unit in the network. Note that the hidden units correspond roughly to detectors that look for strokes and other patterns in the input.

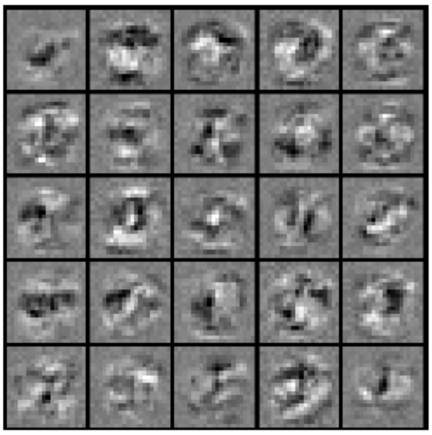


Fig. 5 Visualizing of hidden units

1.9 Learning parameters variation

In this part of the exercise, you will divide data into test and train sub sets and compute the training and testing errors. For example choose randomly 4000 images for training and 1000 images for testing.

Try out different learning settings for the neural network to see how the performance of the neural network varies with the regularization parameter *lambda* and number of training steps (the *MaxIter* option when using *fmincg*).

Neural networks are very powerful models that can form highly complex decision boundaries. Without regularization, it is possible for a neural network to overfit a training set so that it obtains close to 100% accuracy on the training set but does not as well on new examples that it has not seen before. You can set the regularization parameter *lambda* to a smaller value and the *MaxIter* parameter to a higher number of iterations to see this for yourself. Observe also the changes in the visualizations of the hidden units when you change the learning parameters *lambda* and *MaxIter*.

Part 2 Support Vector Machines (SVM) training

Objectives: Implement Support Vector Machines (SVM) algorithm for linearly and nonlinearly separable datasets. SVM with Gaussian kernels. Cross validation to select the best SVM parameters.

Files included in Part 2
ex6.m - Octave/MATLAB script for this exercise
ex6data1.mat - Dataset 1
ex6data2.mat - Dataset 2
ex6data3.mat - Dataset 3
svmTrain.m - SVM training function
svmPredict.m - SVM prediction function
visualizeBoundaryLinear.m - Plot linear boundary
visualizeBoundary.m - Plot non-linear boundary
linearKernel.m - Linear kernel for SVM
gaussianKernel.m - Gaussian kernel for SVM **(you need to complete this function)**
dataset3Params.m - Parameters to use for Dataset 3 **(you need to complete this function)**

2.1 Linearly separable Dataset 1 (linear SVM)

Load and plot data

File *ex6data1.mat* consists of 2D linearly separable dataset (i.e. with linear boundary between the two classes). Load and plot the data to get Fig.1. In this dataset, the positions of the positive examples (indicated with +) and the negative examples (indicated with o) suggest a natural separation indicated by the gap. However, notice that there is an outlier positive example + on the far left at about (0.1; 4.1). You will see how this outlier affects the SVM decision boundary.

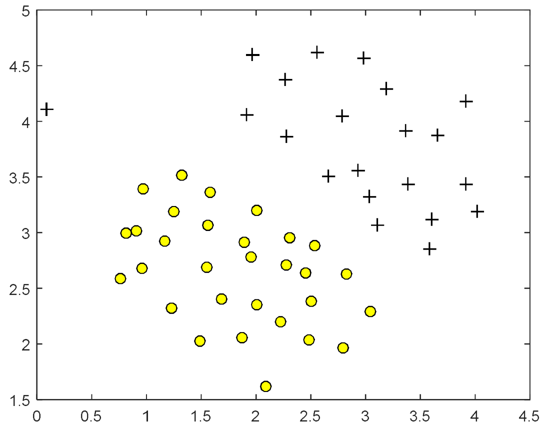


Fig.1 Dataset 1 (ex6data1.mat)

SVM training

Your task is to run the SVM training (function *svmTrain.m*) with different values of the C parameter (for example C=1, C=100) on this dataset. *C* parameter is a positive value that controls the penalty for misclassified training examples. A large *C* tells the SVM to try to classify all examples correctly. *C* plays a role similar to 1/lambda where lambda is the regularization parameter used for logistic regression. When *C* = 1, you should find that the SVM puts the decision boundary in the gap between the two datasets and misclassifies the data point on the far left. When *C* = 100, you should find that the SVM now classifies every single example correctly, but has a decision boundary that does not appear to be a natural fit for the data.

Remark: Most SVM software packages (including *svmTrain.m*) automatically add the extra feature x_0 = 1 and take care of learning the intercept term *theta*$_0$. So when passing the training data to the SVM software, there is no need to add this extra feature x_0 = 1.
svmTrain.m is not very efficient if you need to scale to a larger dataset, it is recommended to use the SVM toolbox such as LIBSVM.

2.2 Dataset 2 (nonlinear SVM with Gaussian Kernels)

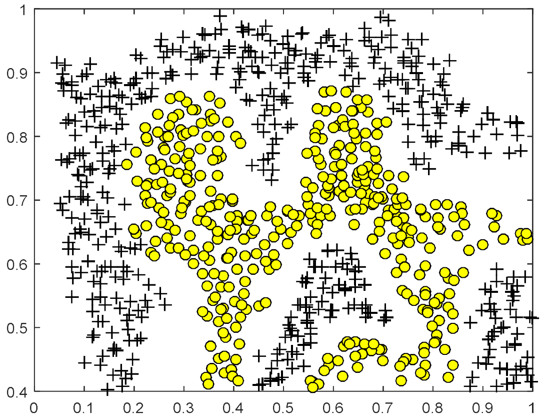


Fig.2 Dataset 2 (ex6data2.mat)

Load and plot data

Load and plot the data from file *ex6data2.mat* to get Fig.2. From the figure, you can observe that there is no linear decision boundary that separates the + and - examples for this dataset. However, by using the Gaussian kernel with the SVM, you will be able to learn a non-linear decision boundary that can perform reasonably well for this dataset.

Gaussian Kernel

To find non-linear decision boundaries with the SVM, we need to first implement a Gaussian kernel. You can think of the Gaussian kernel as a similarity function that measures the distance between a pair of examples ($x^{(i)}$, $x^{(j)}$). The Gaussian kernel is also parameterized by a bandwidth parameter (sigma), which determines how fast the similarity metric decreases to 0 as the examples are further apart. Complete the code in *gaussianKernel.m* to compute the Gaussian kernel between two examples ($x^{(i)}$, $x^{(j)}$). The Gaussian kernel function is defined as

$$K_{gaussian}(x^{(i)}, x^{(j)}) = \exp\left(-\frac{\|x^{(i)} - x^{(j)}\|^2}{2\sigma^2}\right) = \exp\left(-\frac{\sum\limits_{k=1}^{n}(x_k^{(i)} - x_k^{(j)})^2}{2\sigma^2}\right)$$

The main script will test the completed kernel function *gaussianKernel.m* on a couple of examples and you should expect to see a value of 0.324652.

Train the SVM with the Gaussian kernel on this dataset. Fig. 3 shows that the decision boundary is able to separate most of the positive and negative examples correctly.

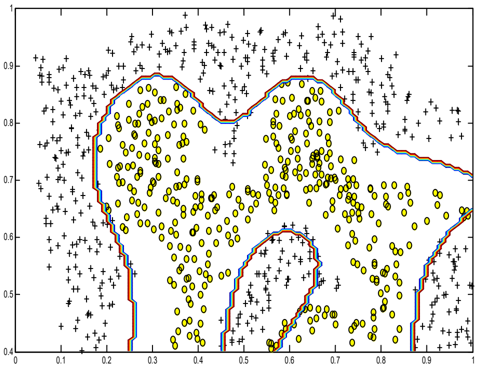


Fig.3 SVM (Gaussian Kernel) Decision Boundary (dataset2)

2.3 Dataset 3 (nonlinear SVM with Gaussian Kernels, optimization of *C* and *sigma*)

Load and plot data

File *ex6data3.mat* contains training data (*X,y*) and validation data (*Xval, yval*). Load the data and display the training data to get Fig.4. The task is to use the validation set *Xval, yval* to determine the best *C* and *sigma* parameters to use. You should modify the code in *dataset3Params.m*, to search over the parameters *C* and *sigma*. For both *C* and *sigma*, we suggest trying the following values (0.01; 0.03; 0.1; 0.3; 1; 3; 10; 30). The function *dataset3Params.m* tries all possible pairs of values for C and sigma. For example, for the 8 values listed above for *C* and for *sigma*, a total of 8^2 = 64 different models will be trained and evaluated (on the validation set). For the best parameters, the SVM will return a decision boundary shown in Fig. 5.

Remark: When implementing validation to select the best *C* and *sigma* parameters to use, the error is defined as the fraction of the validation examples that were classified incorrectly. In Octave/MATLAB, this error is computed as *mean(double(predictions ~= yval)),* where *predictions* is a vector containing all the predictions from the SVM, and *yval* are the true labels from the validation set. *svmPredict* function generate the predictions for the validation set.

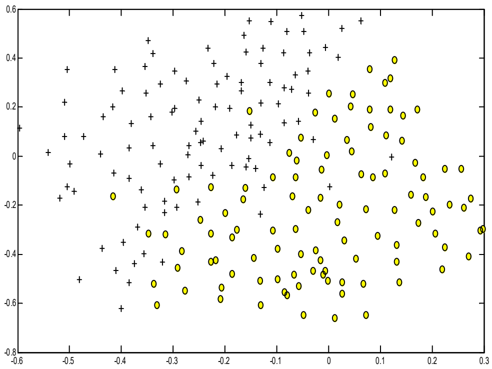


Fig. 4 Dataset3

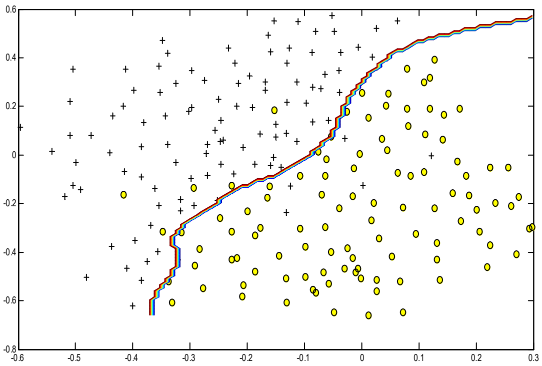


Fig.5 SVM (Gaussian Kernel) -Dataset3