Programming Assignment 3: Neural Networks[1]

See website for due date

What to submit (staple in this order):

1. grade3.txt
2. **Any modified files** under starter code folder, with your completed code
3. Screenshots of the output (try to reduce size)
4. Solutions to the written problem

Grading. We will randomly choose a subset of submissions and verify that 1) the code produces the answers and 2) the points computation in "grade.txt" is correct. There are three written problems, one of them will be chosen at random and will be worth 30 points. Total: 130 points.

1. Feedforward Propagation

In assignment 2, you implemented multi-class logistic regression to recognize handwritten digits. However, logistic regression cannot form more complex hypotheses as it is only a linear classifier.

In this part of the exercise, you will implement a neural network to recognize handwritten digits using the same training set as before. The neural network will be able to represent complex models that form non-linear hypotheses. For this part, you will be using parameters from a neural network that we have already trained. Your goal is to implement the feedforward propagation algorithm to use our weights for prediction. In the next part, you will write the backpropagation algorithm for learning the neural network parameters.

The provided script, ex3_fe.m, will help you step through this exercise.

1.1 Model representation

Our neural network is shown in Figure 1. It has 3 layers – an input layer, a hidden layer and an output layer. Recall that our inputs are pixel values of digit images. Since the images are of size 20×20, this gives us 400 input layer units (excluding the extra bias unit which always outputs +1). As before, the training data will be loaded into the

[1] Note: this assignment is adapted from programming assignments of Andrew Ng's online Machine Learning course(https://class.coursera.org/ml).

variables X and y.

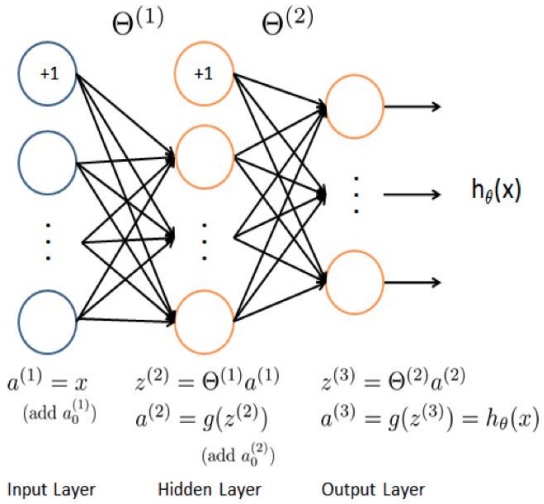


Figure 1: Neural network model.

You have been provided with a set of network parameters ($\Theta^{(1)}$,$\Theta^{(2)}$) already trained by us. These are stored in ex3weights.mat and will be loaded by ex3_fe.m into Theta1 and Theta2. The parameters have dimensions that are sized for a neural network with 25 units in the second layer and 10 output units (corresponding to the 10 digit classes).

```
% Load saved matrices from file
load('ex3weights.mat');

% The matrices Theta1 and Theta2 will now be in your Octave
% environment
% Theta1 has size 25 x 401
% Theta2 has size 10 x 26
```

1.2 Feedforward Propagation and Prediction

Now you will implement feedforward propagation for the neural network. You will need to complete the code in predict.m to return the neural network's prediction.

You should implement the feedforward computation that computes $h_\theta(x^{(i)})$ for every example i and returns the associated predictions. Similar to the one-vs-all classification (assignment 2) strategy, the prediction from the neural network will be the label that has the largest output $(h_\theta(x))_k$.

Implementation Note: The matrix X contains the examples in rows. When you complete the code in predict.m, you will need to add the column of 1's to the matrix.

The matrices Theta1 and Theta2 contain the parameters for each unit in rows. Specifically, the first row of Theta1 corresponds to the first hidden unit in the second layer. In Octave, when you compute $z^{(2)} = \Theta^{(1)} a^{(1)}$, be sure that you index (and if necessary, transpose) X correctly so that you get $a^{(l)}$as a column vector.

Once you are done, ex3_fe.m will call your predict function using the loaded set of parameters for Theta1 and Theta2. You should see that the accuracy is about 97.5%. [20 points for getting this answer]

After that, an interactive sequence will launch displaying images from the training set one at a time, while the console prints out the predicted label for the displayed image. To stop the image sequence, press Ctrl-C.

1.3 Feedforward and cost function

Now you will implement the cost function and gradient for the neural network. First, complete the code in nnCostFunction.m to return the cost.

The provided script, ex3_ba.m, will help you step through the following parts of this assignment.

Recall that the cost function for the neural network (without regularization) is

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

where $h_\theta(x^{(i)})$ is computed as shown in the Figure 1 and K = 10 is the total number of possible labels. Note that $h_\theta(x^{(i)})_k = a_k^{(3)}$ is the activation (output value) of the k-th output unit. Also, recall that whereas the original labels (in the variable y) were 1, 2, ..., 10, for the purpose of training a neural network, we need to recode the labels as vectors containing only values 0 or 1, so that

$$y = \begin{bmatrix} 1 \\ 0 \\ 0 \\ \vdots \\ 0 \end{bmatrix}, \quad \begin{bmatrix} 0 \\ 1 \\ 0 \\ \vdots \\ 0 \end{bmatrix}, \quad \ldots \quad \text{or} \quad \begin{bmatrix} 0 \\ 0 \\ 0 \\ \vdots \\ 1 \end{bmatrix}$$

For example, if $x^{(i)}$ is an image of the digit 5, then the corresponding $y^{(i)}$ (that you should use with the cost function) should be a 10-dimensional vector with $y_5 = 1$, and the other elements equal to 0.

You should implement the feedforward computation that computes $h_\theta(x^{(i)})$ for every example i and sum the cost over all examples. Your code should also work for a dataset of any size, with any number of labels (you can assume that there are always at least K ≥ 3 labels).

Implementation Note: The matrix X contains the examples in rows (i.e., X(i,:)' is the i-th training example $x^{(i)}$, expressed as a n × 1 vector.) When you complete the code in nnCostFunction.m, you will need to add the column of 1's to the X matrix. The parameters for each unit in the neural network is represented in Theta1 and Theta2 as one row. Specifically, the first row of Theta1 corresponds to the first hidden unit in the second layer. You can use a for-loop over the examples to compute the cost.

Once you are done, ex3_ba.m will call your nnCostFunction using the loaded set of parameters for Theta1 and Theta2. You should see that the cost is about 0.287629. [15 points for getting this answer]

1.4 Regularized cost function

The cost function for neural networks with regularization is given by

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

You can assume that the neural network will only have 3 layers – an input layer, a hidden layer and an output layer. However, your code should work for any number of input units, hidden units and outputs units. While we have explicitly listed the indices above for $\Theta^{(1)}$ and $\Theta^{(2)}$for clarity, do note that your code should in general work with $\Theta^{(1)}$ and $\Theta^{(2)}$of any size.

Note that you should not be regularizing the terms that correspond to the bias. For the matrices Theta1 and Theta2, this corresponds to the first column of each matrix. You should now add regularization to your cost function. Notice that you can first compute the unregularized cost function J using your existing nnCostFunction.m and then later add the cost for the regularization terms.

Once you are done, ex3_ba.m will call your nnCostFunction using the loaded set of parameters for Theta1 and Theta2, and λ = 1. You should see that the cost is about 0.383770. [15 points for getting this answer]

2. Backpropagation

In this part of the exercise, you will implement the backpropagation algorithm to compute the gradient for the neural network cost function. You will need to complete the nnCostFunction.m so that it returns an appropriate value for grad. Once you have computed the gradient, you will be able to train the neural network by minimizing the cost function $J(\Theta)$ using an advanced optimizer such as fmincg.

You will first implement the backpropagation algorithm to compute the gradients for the parameters for the (unregularized) neural network. After you have verified that your gradient computation for the unregularized case is correct, you will implement the gradient for the regularized neural network.

2.1 Sigmoid gradient

To help you get started with this part of the exercise, you will first implement the sigmoid gradient function. The gradient for the sigmoid function can be computed as

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

where

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

When you are done, try testing a few values by calling sigmoidGradient(z) at the Octave 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. [10 points for this part]

2.2 Random initialization

When training neural networks, it is important to randomly initialize the parameters for symmetry breaking. One effective strategy for random initialization is to randomly select values for $\Theta^{(l)}$ uniformly in the range $[-\epsilon_{init}, \epsilon_{init}]$. You should use $\epsilon_{init} = 0.12$. This range of values ensures that the parameters are kept small and makes the learning more efficient

Your job is to complete randInitializeWeights.m to initialize the weights for Θ; modify the file and fill in the following code:

```
% Randomly initialize the weights to small values
```

```
epsilon init = 0.12;
W = rand(L out, 1 + L in) * 2 * epsilon init − epsilon init;
```

2.3 Backpropagation

Now, you will implement the backpropagation algorithm. Recall that the intuition behind the backpropagation algorithm is as follows. Given a training example $(x^{(t)}, y^{(t)})$, we will first run a "forward pass" to compute all the activations throughout the network, including the output value of the hypothesis $h_\Theta(x)$. Then, for each node j in layer l, we would like to compute an "error term" $\delta_j^{(l)}$ that measures how much that node was "responsible" for any errors in our 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, you will compute $\delta_j^{(l)}$ based on a weighted average of the error terms of the nodes in layer $(l + 1)$.

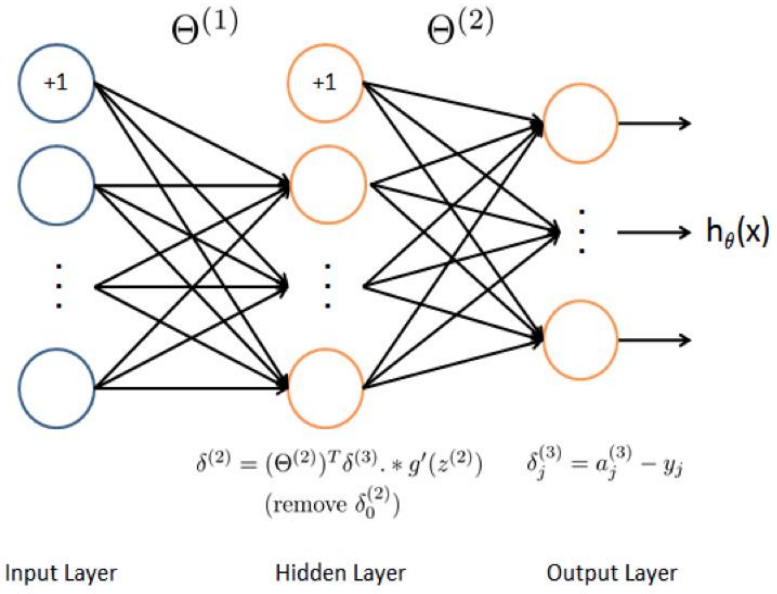


Figure 2: Backpropagation Updates

In detail, here is the backpropagation algorithm (also depicted in Figure 2). You should implement steps 1 to 4 in a loop that processes one example at a time. Concretely, you should implement a for-loop for t = 1:m and place steps 1-4 below inside the for-loop, with the t^{th} iteration performing the calculation on the t^{th} training example $(x^{(t)}, y^{(t)})$. Step 5 will divide the accumulated gradients by m to obtain the gradients for the neural

network cost function.

1. Set the input layer's values ($a^{(1)}$) to the t^{th} training example $x^{(t)}$. Perform a feedforward pass (Figure 1), 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, 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, 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 $\frac{1}{m}$:

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

Octave Tip: You should implement the backpropagation algorithm only after you have successfully completed the feedforward and cost functions. While implementing the backpropagation algorithm, it is often useful to use the size function to print out the sizes of the variables you are working with if you run into dimension mismatch errors ("nonconformant arguments" errors in Octave).

After you have implemented the backpropagation algorithm, the script ex3_ba.m will proceed to run gradient checking on your implementation. The gradient check will allow you to increase your confidence that your code is computing the gradients correctly.

2.4 Gradient checking

In your neural network, you are minimizing the cost function $J(\Theta)$. To perform gradient checking on your parameters, you can imagine "unrolling" the parameters $\Theta^{(1)}$, $\Theta^{(2)}$ into a long vector θ. By doing so, you can think of the cost function being $J(\theta)$ instead and use the following gradient checking procedure.

Suppose you have a function $f_i(\theta)$ that purportedly computes $\frac{\partial}{\partial \theta_i} J(\theta)$; you'd like to check if f_i is outputting correct derivative values.

$$\text{Let} \quad \theta^{(i+)} = \theta + \begin{bmatrix} 0 \\ 0 \\ \vdots \\ \epsilon \\ \vdots \\ 0 \end{bmatrix} \qquad \text{and} \quad \theta^{(i-)} = \theta - \begin{bmatrix} 0 \\ 0 \\ \vdots \\ \epsilon \\ \vdots \\ 0 \end{bmatrix}$$

So, $\theta^{(i+)}$ is the same as θ, except its i-th element has been incremented by ε. Similarly, $\theta^{(i-)}$ is the corresponding vector with the i-th element decreased by ε. You can now numerically verify $f_i(\theta)$'s correctness by checking, for each i, that:

$$f_i(\theta) \approx \frac{J(\theta^{(i+)}) - J(\theta^{(i-)})}{2\epsilon}.$$

The degree to which these two values should approximate each other will depend on the details of J. But assuming ε = $,10^{-4}$ you'll usually find that the left- and right-hand sides of the above will agree to at least 4 significant digits (and often many more).

We have implemented the function to compute the numerical gradient for you in computeNumericalGradient.m. While you are not required to modify the file, we highly encourage you to take a look at the code to understand how it works.

In the next step of ex3_ba.m, it will run the provided function checkNNGradients.m which will create a small neural network and dataset that will be used for checking your gradients. If your backpropagation implementation is correct, you should see a relative difference that is less than 1e-9. [10 points for getting this answer]

Practical Tip: When performing gradient checking, it is much more efficient to use a small neural network with a relatively small number of input units and hidden units, thus having a relatively small number of parameters. Each dimension of θ requires two evaluations of the cost function and this can be expensive. In the function checkNNGradients, our code creates a small random model and dataset which is

used with computeNumericalGradient for gradient checking. Furthermore, after you are confident that your gradient computations are correct, you should turn off gradient checking before running your learning algorithm.

Practical Tip: Gradient checking works for any function where you are computing the cost and the gradient. Concretely, you can use the same computeNumericalGradient.m function to check if your gradient implementations for the other exercises are correct too (e.g., logistic regression's cost function).

2.5 Regularized Neural Networks

After you have successfully implemented the backpropagation algorithm, you will add regularization to the gradient. To account for regularization, it turns out that you can add this as an additional term after computing the gradients using backpropagation.

Specifically, after you have computed $\Delta_{ij}^{(l)}$ using backpropagation, you should add regularization using

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

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

Note that you should not be regularizing the first column of $\Theta^{(l)}$ which is used for the bias term. Furthermore, in the parameters $\Theta_{ij}^{(l)}$, i is indexed starting from 1, and j is indexed starting from 0. Thus,

$$\Theta^{(l)} = \begin{bmatrix} \Theta_{1,0}^{(i)} & \Theta_{1,1}^{(l)} & \cdots \\ \Theta_{2,0}^{(i)} & \Theta_{2,1}^{(l)} & \\ \vdots & & \ddots \end{bmatrix}.$$

Somewhat confusingly, indexing in Octave starts from 1(for both i and j), thus Theta(2,1) actually corresponds to $\Theta_{2,0}^{(l)}$ (i.e., the entry in the second row, first column of the matrix $\Theta^{(1)}$ shown above)

Now modify your code that computes grad in nnCostFunction to account for regularization. After you are done, the ex3_ba.m script will proceed to run gradient checking on your implementation. If your code is correct, you should expect to see a

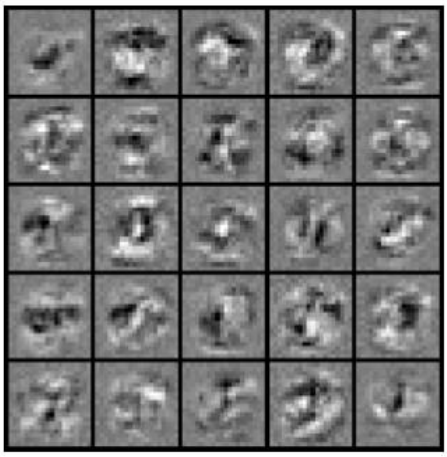
relative difference that is less than 1e-9. [10 points for getting this answer]

2.6 Learning parameters using fmincg

After you have successfully implemented the neural network cost function and gradient computation, the next step of the ex3_ba.m script will use fmincg to learn a good set parameters.

After the training completes, the ex3_ba.m script will proceed to report the training accuracy of your classifier by computing the percentage of examples it got correct. If your implementation is correct, you should see a reported training accuracy of about 95.3% (this may vary by about 1% due to the random initialization). [20 points for getting this answer]
It is possible to get higher training accuracies by training the neural network for more iterations. We encourage you to try training the neural network for more iterations (e.g., set MaxIter to 400) and also vary the regularization parameter λ. With the right learning settings, it is possible to get the neural network to perfectly fit the training set.

3. Visualizing the hidden layer



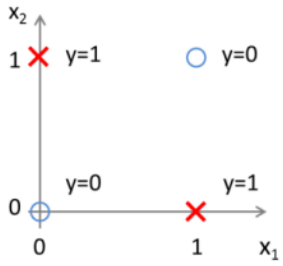
One way to understand what your neural network is learning is to visualize what the representations captured by the hidden units. Informally, given a particular hidden unit, one way to visualize what it computes is to find an input x that will cause it to activate (that is, to have an activation value ($a_i^{(l)}$) close to 1). For the neural network you trained, notice that the ith row of $\Theta^{(1)}$ is a 401-dimensional vector that represents the parameter for the ith 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.

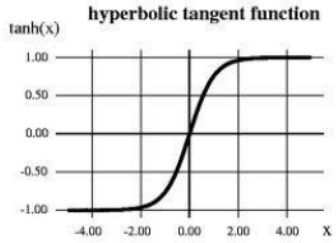
Thus, one way to visualize the "representation" captured by the hidden unit is to reshape this 400 dimensional vector into a 20 × 20 image and display it. The next step of ex3_ba.m does this by using the displayData function and it will show you an image (similar to above figure) with 25 units, each corresponding to one hidden unit in the network.

In your trained network, you should find that the hidden units corresponds roughly to detectors that look for strokes and other patterns in the input.

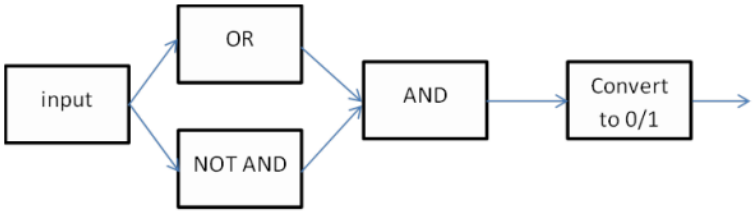
Written Problem 1: XOR Neural Net

Design a neural network to solve the XOR problem, i.e. the network should output 1 if only one of the two binary input variables is 1, and 0 otherwise (see left figure). Use the hyperbolic tangent, or *tanh,* activation function in all nodes (right figure), which ranges in [-1, +1].





Note that (A XOR B) can be expressed as (A OR B) AND NOT(A AND B)), as illustrated below:



In the diagrams below, we filled in most of the tanh units' parameters. Fill in the remaining parameters, keeping in mind that tanh outputs +1/-1, not 0/1. Note that we need to appropriately change the second layer (the AND node) to take +1/-1 as inputs. Also, we must add an extra last layer to convert the final output from +1/-1 to 0/1. *Hint: assume tanh outputs* -1 *for any input* $x \leq -2$, $+1$ *for any input* $x \geq +2$, 0 *for* $x = 0$.

Written Problem 2: Cost Functions (Quiz1)

Suppose we have a cost function

$$J(\theta) = \frac{1}{2m}\sum_{i=1}^{m} \theta^T A\theta + x_i^T\theta + by_i,$$

where $\theta \in \mathbb{R}^n$ is the parameter vector, $x_i \in \mathbb{R}^n$, $y_i \in \mathbb{R}$, $\{x_i, y_i\}$ are m training data points, $A \in \mathbb{R}^{nxn}$ is a symmetric matrix, and $b \in \mathbb{R}$. We want to find parameters θ using gradient descent.

a) Give the pseudo code for the gradient decent algorithm for a **generic** cost function $J(\theta)$ (not the specific one above).

b) For the specific function above, what is the vector of partial gradients of the cost function, i.e. the vector with the jth element equal to $\frac{\partial}{\partial\theta_j}J(\theta)$?

c) What is the design matrix? Describe its entries and give its dimensions.

d) Re-write the expression for the gradient without using the summation notation $\sum$. *Hint: use the design matrix X.*

e) Suppose we run gradient descent for two iterations. Give the expression for θ after two updates, with step size $\alpha = 1$ and initial value of $\theta = 0$.

f) How do we know when the algorithm has converged?

g) Give the closed-form solution for θ. You do not need to prove it is the minimum of the cost.

Written Problem 3: Probabilistic Linear Regression (Quiz 1)

Recall that probabilistic linear regression defines the likelihood of observing outputs $t^{(i)}$ given inputs $x^{(i)} \in \mathbb{R}^p$ as

$$p(t_1, \ldots, t_m | x_1 \ldots, x_m, \theta, \beta) = \prod_{i=1}^{m} N(t^{(i)} | h(x^{(i)}),\ \beta^{-1})$$

where θ, β are parameters.

(a) Find the maximum likelihood solution for β, β_{ML}

(b) What is the interpretation of β_{ML}?

(c) Suppose each data point had a different β_i. Show that the β_i can be set such that probabilistic linear regression becomes equivalent to localized linear regression from problem set 1.

(d) Draw the hypothesis h_θ for the dataset below, for the particular input query x^q. On the left, show hypothesis and predicted output $h_\theta(x^q)$ for regular regression, on the right, for localized regression.