Table of Contents

[Neural Network 1](#_Toc29805711)

[**Forward Propagation** 1](#_Toc29805712)

[**Motivation for neural network: fit non-linear hypothesis** 2](#_Toc29805713)

[**Multi-class classification: One-vs-all** 5](#_Toc29805714)

[**Cost function (classification)** 6](#_Toc29805715)

[**Backpropagation** 6](#_Toc29805716)

[**Implementation Issues** 9](#_Toc29805717)

[**Put it all together: build and train neural network** 11](#_Toc29805718)

# **Neural Network**

Neurons are basically computational units that take inputs (**dendrites**) as electrical inputs ("spikes") that are channelled to outputs (**axons**).

* Dendrites are like the input features
* Hypothesis output is sigmoid function applied to the sum of activation

sigmoid (logistic) activation function:

* input node is called the "bias unit." It is always equal to 1.

### **Forward Propagation**

A close up of a clock

Description automatically generated

: activation (output) of unit i in layer j

: matrix of weights controlling function mapping from layer j to j + 1

If network has units in layer j, units in layer j + 1, then is a matrix.

+1 because of bias nodes. Output nodes do not include bias nodes, but the inputs do.

In linear and logistic regression, is a vector. In neural network, (capital theta) is a matrix.

Forward Propagation:

is a 3x4 matrix, is a 1x4 matrix

Forward Propagation: Vectorized

Inputs:

Sum of activation of layer 1: 3x1 vector

Activation of layer 2: 3x1 vector, g applied to each element of

Add a bias unit to layer 2: 4x1 vector

Sum of activation of layer 2: a number

Activation of layer 3 (hypothesis output):

**Forward Propagation: Generalized Vectorization**

Inputs:

Sum of activation of layer j – 1:

Activation of layer j:

Add a bias unit to layer j:

Sum of activation of layer j:

Activation of layer j + 1 (hypothesis output):

### **Motivation for neural network: fit non-linear hypothesis**

When we need a non-linear hypothesis, we add polynomial terms to linear hypothesis.

However, this will increase the number of features dramatically, especially when n (# of features) is large.

Adding all quadratic terms increase the number of features from n to

This can be computationally expensive.

e.g. original 2500 features, quadratic features 3 million

Hidden layers in neural networks allow us to elegantly fit complex non-linear hypothesis.

Computation between layer 2 and 3 is the same as logistic regression.

Instead of using original features , **neural network learns its own features** (**activation of hidden layers)**as new features for logistic regression.

are learned functions of inputs , and are determined by parameters .

Neural network is not constrained to using raw features or their polynomial terms 🡪 more flexible in learning features and better hypothesis function.

A picture containing object, clock

Description automatically generatedA picture containing object

Description automatically generated

**Examples of Neural Network Computing Non-linear Hypotheses**

Logical operation: use negative weights for the variables that you want to negate.

EX 1: AND

Binary

Boolean function:

|  |  |  |
| --- | --- | --- |
| Truth Table | | |
|  |  |  |
| 0 | 0 |  |
| 0 | 1 |  |
| 1 | 0 |  |
| 1 | 1 |  |

A close up of a map

Description automatically generatedA picture containing object

Description automatically generated

EX 2: NOT

Binary

A close up of a clock

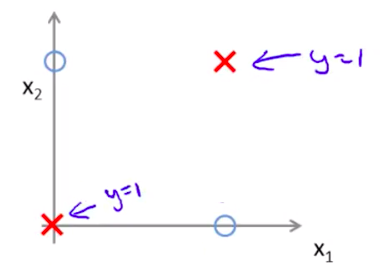
Description automatically generatedBoolean function:

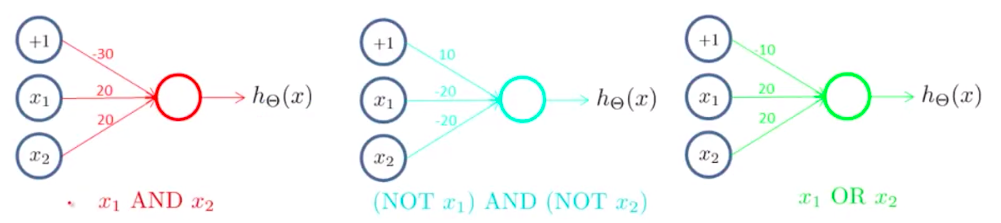
|  |  |
| --- | --- |
| Truth Table | |
|  |  |
| 0 |  |
| 1 |  |

EX 3: XNOR

XOR : true if exactly one of is true

XNOR : true if (inverse of XOR)





|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Truth Table | | | | |
|  |  |  |  |  |
| 0 | 0 | 0 | 1 | 1 |
| 0 | 1 | 0 | 0 | 0 |
| 1 | 0 | 0 | 0 | 0 |
| 1 | 1 | 1 | 0 | 1 |

A close up of a map

Description automatically generated

Intuition: To capture non-linear relations, we add hidden layers. Hidden layers build on top of each other to compute complex non-linear hypotheses.

### **Multi-class classification: One-vs-all**

To predict n classes, we have n nodes in the output layer.

Hypothesis output is a n-dimensional vector.

Training set:

Previously, . Now is one of

A close up of a map

Description automatically generated

For a multi-class classification problem:

m = # of training examples

n = # of features

h = # of nodes in the hidden layer (not including bias unit)

c = # of classes

Input X (m, n+1)

Theta 1 (h, n+1)

Theta 2 (c, h+1)

|  |
| --- |
| function p = predict(Theta1, Theta2, X)  % Predict the label of X given the trained weights of a neural network (Theta1, Theta2)  m = size(X, 1);  num\_labels = size(Theta2, 1);  % X(5000, 401) (m, n+1) n = # of features + 1 (bias unit)  % Theta1 (25, 401) (h, n+1) h = # of hidden units  % Theta2 (10, 26) (k, h+1) c = # of labels  X = [ones(m, 1) X]; % Add a bias unit  z\_2 = X \* Theta1'; % Sum of activation of layer 1  a\_2 = sigmoid(z\_2); % Activation of layer 2: mxh  s = size(a\_2, 1); % Add a bias unit to layer 2: mx(h+1)  a\_2 = [ones(s, 1) a\_2];  z\_3 = a\_2 \* Theta2'; % Sum of activation of layer 2  a\_3 = sigmoid(z\_3); % Activation of layer 3 (hypothesis output): mxc  % return p: a m-dimensional vector containing labels btw 1 to num\_labels.  [value indices] = max(a\_3, [], 2);  p = indices;  end |

### **Cost function (classification)**

n = # of features j = 0, ... n

m = # of training examples i = 1, ... m

L = total # of layers

= # of units (not including bias unit) in layer l

K = # of classes

* Binary classification: one output node, y = 0 or 1
* Multi-class classification (K 3): K output nodes, is a K-dimensional vector

Cost function for regularized logistic regression:

(do not regularize )

Cost function for neural network (classification):

(do not regularize bias unit)

* is a K-dimensional vector, is the k-th output
* The double sum adds up the logistic regression costs calculated for each cell in the output layer.
* The triple sum adds up the squares of all the individual s in the entire network.
* : matrix of weights controlling function mapping from layer l to l + 1, a matrix
* i in the triple sum does **not** refer to training example i

### **Backpropagation**

An algorithm to find that minimizes

To minimize , we want to compute the partial derivative

Consider one training example, one output unit, two hidden layers, without regularization

A close up of text on a white background

Description automatically generated

Forward propagation:

after adding bias unit

after adding bias unit

after adding bias unit

Backpropagation:

*Transposing is different when vectorized. Here only one example, is a vector, so transpose*

Cost for one example:

For output layer:

For hidden layer:

Let delta common term between output layer and hidden layer

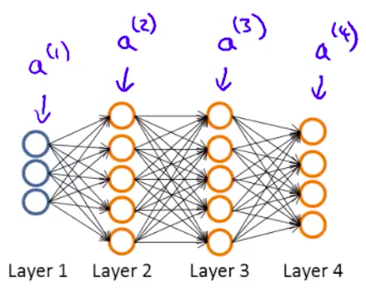
Let delta common term btw last hidden layer and hidden layer before it

For output layer:

For hidden layer:

For multi-class classification:

Backpropagation: vectorized:



**Backpropagation Algorithm:** backpropagate error starting from the output layer

Training set:

= a vector of activation of the output layer

= error in layer l, how much that layer is responsible for errors in output

(no because no error in feature layer)

D = gradient matrix

Set = 0 for all l, i, j ( is capital )

For i = 1:m

Set

Forward propagation: compute

Compute error term

Backpropagation: compute (do not calculate error of bias unit)

(do not regularize bias unit)

### **Implementation Issues**

1). Random initialization: symmetry breaking

Zero initialization works with logistic regression, but not with neural network.

A picture containing object

Description automatically generated

Same zero weights 🡪

🡪 Same update in every iteration of gradient descent: repeat until convergence {}

🡪 All hidden units compute the same function of input, redundant representation of the same feature

Symmetry breaking:

Initialize each to a random value in

= # of units (not including bias unit) in layer l

L = # of units in the layers adjacent to

% If the dimensions of Theta1 is 10x11, Theta2 is 10x11 and Theta3 is 1x11.

Theta1 = rand(10,11) \* (2 \* epsilon\_init) - epsilon\_init;

Theta2 = rand(10,11) \* (2 \* epsilon\_init) - epsilon\_init;

Theta3 = rand(1,11) \* (2 \* epsilon\_init) - epsilon\_init;

|  |
| --- |
| function W = randInitializeWeights(L\_in, L\_out)  % Randomly initialize the weights of a layer  % with L\_in incoming connections and L\_out outgoing connections  % Random initialization with symmetry breaking  % rand(x, y) returns a matrix (x,y) of real numbers btw 0 and 1  epsilon\_init = 0.12;  W = rand(L\_out, 1 + L\_in) \* (2 \* epsilon\_init) - epsilon\_init;  % Note that W should be set to a matrix of size(L\_out, 1 + L\_in)  % as the first column of W handles the bias unit  end |

2). Unrolling parameters

Advantage of using matrix: easy to implement vectorized forward propagation and backpropagation

Advantage of using vector: advanced optimization algorithms assume parameters are vectors

% unroll to get initialTheta (as a long vector) to pass to fminunc

thetaVector = [ Theta1(:); Theta2(:); Theta3(:); ]

% Get back original matrices

Theta1 = reshape(thetaVector(1:110),10,11)

Theta2 = reshape(thetaVector(111:220),10,11)

Theta3 = reshape(thetaVector(221:231),1,11)

3). Gradient checking

A close up of a map

Description automatically generated

When is a number:

Approximate gradient using

For small enough, this approximation will equal to gradient.

When is a vector (unrolled version of )

...

|  |
| --- |
| function numgrad = computeNumericalGradient(J, theta)  % Computes the numerical gradient of the function J around theta.  % This works for any function that compute cost and gradient.  % You can also use it for logistic regression's cost function.  % numgrad(i) is (a numerical approx. of) partial derivative of J w.r.t. i-th input, evaluated at theta.  % We will call computeNumericalGradient(costFunc, nn\_params);  % nn\_params is a long vector (10285,1)  numgrad = zeros(size(theta));  perturb = zeros(size(theta));  e = 1e-4;  for p = 1:numel(theta) % returns the number of elements in an array  % Set perturbation vector  perturb(p) = e;  loss1 = J(theta - perturb);  loss2 = J(theta + perturb);  % Compute Numerical Gradient  numgrad(p) = (loss2 - loss1) / (2 \* e);  perturb(p) = 0;  end  end |

### **Put it all together: build and train neural network**

# of input units = dimension of features

# of output units = # of classes

# of hidden layers: default 1, if >1, recommended to have same # of hidden units in each layer

# of hidden units: the more the better, but more computationally expensive

1. Initialize random weights
2. Forward propagation to compute
3. Compute cost function
4. Backpropagation to compute partial derivatives
5. Gradient checking: create a small neural network to check if = numerical gradients

Turn off gradient checking before training classifier (running gradient checking in every iteration of gradient descent is very slow).

1. Train neural network: use gradient descent or advanced optimization algo with backpropagation to find optimal that minimizes
2. Use optimal to make predictions

*is not convex due to the presence of hidden layers.*

*We may end up with a local minimum (not a big problem in practice).*

*Solution: run gradient descent multiple times. With random initialization of theta, we get a different local minimum each time. Select theta values that give the lowest cost.*

|  |
| --- |
| function [J grad] = nnCostFunction(nn\_params, ...  input\_layer\_size, ...  hidden\_layer\_size, ...  num\_labels, ...  X, y, lambda)    % Computes the cost and gradient of a two layer neural network classifier.  % Parameters for neural network are "unrolled" into the vector ‘nn\_params’  % and need to be converted back into the weight matrices.    % Reshape nn\_params back into weight matrices Theta1 and Theta2  Theta1 = reshape(nn\_params(1:hidden\_layer\_size\*(input\_layer\_size + 1)), ...  hidden\_layer\_size, (input\_layer\_size + 1));    Theta2 = reshape(nn\_params((1+hidden\_layer\_size\*(input\_layer\_size+1)):end),...  num\_labels, (hidden\_layer\_size + 1));    m = size(X, 1);    % ============ Part 1: Forward propagation =============  % Return the cost in the variable J.  % X(5000, 401) (m, n+1)  % Theta1 (25, 401) (h, n+1)  % Theta2 (10, 26) (k, h+1)    % Original labels in 'y' are 1, 2, ... 10  % We need 'y' as vectors with i-th unit = 1 for i-th class, and 0 elsewhere.  % Expand the 'y' output values into a matrix of single values  eye\_matrix = eye(num\_labels); % identity matrix  y\_matrix = eye\_matrix(y,:); % (m,k)    X = [ones(m, 1) X];  a\_1 = X;  z\_2 = a\_1 \* Theta1'; % Sum of activation of layer 1  a\_2 = sigmoid(z\_2); % Activation of layer 2: (m,h)    s = size(a\_2, 1); % Add a bias unit to layer 2: (m,h+1)  a\_2 = [ones(s, 1) a\_2];    z\_3 = a\_2 \* Theta2'; % Sum of activation of layer 2  a\_3 = sigmoid(z\_3); % Activation of layer 3 (hypothesis output): (m,k)      % ============ Part 2: Compute Cost J =============  % Note: Use element-wise multiplication for neural network  % because multiplication in cost equation is an element-wise scalar product.    % For linear and logistic regression, 'y' and 'h' were both vectors,  % so we could compute the sum of their products using vector multiplication.  % After transposing one of the vectors, we get a scalar value (1 x m)\*(m x 1).  % Why double sum: <https://www.coursera.org/learn/machine-learning/discussions/all/threads/AzIrrO7wEeaV3gonaJwAFA>    % Do not regularize bias unit (first column)  reg\_1 = sum(sum(Theta1(:,2:end) .^ 2));  reg\_2 = sum(sum(Theta2(:,2:end) .^ 2));    J = (1/m) \* sum(sum(- y\_matrix .\* log(a\_3) - (1 - y\_matrix) .\* log(1 - a\_3))) + (lambda / (2\*m)) \* (reg\_1 + reg\_2);      % ============ Part 2: Backpropagation =============  % Return the partial derivatives of the cost function  % w.r.t Theta1 and Theta2 as Theta1\_grad, Theta2\_grad  % Then check using checkNNGradients    delta\_3 = a\_3 - y\_matrix; % (m,k)    % Note we exclude the first column of Theta2 (bias unit)  % because input units do not connect to the bias unit in hidden layer  % so we do not use backpropagation for it.  delta\_2 = delta\_3\* (Theta2(:,2:end)) .\* sigmoidGradient(z\_2);%(m,k)(k,h)=(m,h)    Delta\_1 = delta\_2' \* a\_1; % (h,m)(m,n+1) = (h,n+1)  Delta\_2 = delta\_3' \* a\_2; % (k,m)(m,h+1) = (k,h+1)    % Do not regularize bias unit  Theta1\_reg = Theta1;  Theta1\_reg(:, 1) = 0;  Theta2\_reg = Theta2;  Theta2\_reg(:, 1) = 0;    Theta1\_grad = (1/m)\* (Delta\_1 + lambda \* Theta1\_reg); % (h,n+1)  Theta2\_grad = (1/m)\* (Delta\_2 + lambda \* Theta2\_reg); % (k,h+1)    % size(Theta1\_grad) = size(Delta\_1) = size(Theta1) = (h, n+1)  % size(Theta2\_grad) = size(Delta\_2) = size(Theta2) = (k, h+1)      % grad is a "unrolled" vector of the partial derivatives of neural network.  grad = [Theta1\_grad(:) ; Theta2\_grad(:)];    end |