CAP-5615 Intro to Neural Networks Summer 2018
Homework 4
Christian Garbin

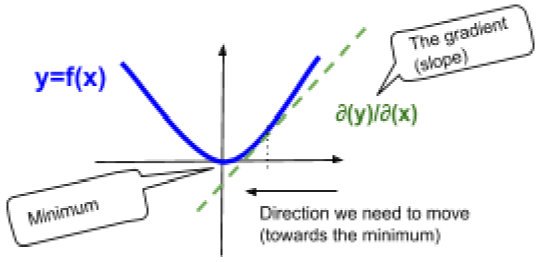
Question 1 [1.5 points: 0.25/each]

Please use your own language to briefly explain the following concepts (Must use your own language. No credit if descriptions are copied from external sources):

Gradient Descent

Gradient Descent is a method to find the local minimum of a function, by moving in the opposite direction of the gradient (slope) of a given point. In machine learning it is used to train neural networks, adjusting weights by iteratively reducing the network error.

In mathematical terms the gradient is the first derivative of a function in a given point:



In neural networks the gradient is used to calculate the network error, in relation to the weights:

$$\nabla E(W) = \left(\frac{\delta E}{\delta w_0}, \frac{\delta E}{\delta w_1}, \ldots, \frac{\delta E}{\delta w_m}\right)$$

Neural network learning rate

Learning rate is a parameter to control by how much we adjust the weights when training a neural network. It has to be chosen carefully, finding a balance between a small value (may take too long to converge) and too large (may overshoot the local minimum, failing to converge). The learning rate is represented by the greek lower-case letter eta (η).

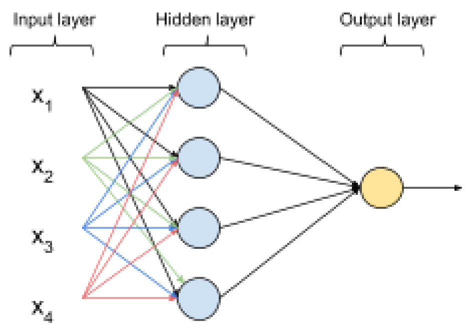
For example, this is the learning rate applied to the Gradient Descent Rule (batch mode) to control the rate of weight adjustments:

$$w_i(k+1) = w_i(k) + \eta \sum_{n \in D} (d(n) - o(n)) x_i(n)$$

Multi-Layer Feed Forward Neural Network

A multi-layer feed forward neural network is a neural network that has one or more hidden layers between the input and the output nodes. The number hidden layers in a network is determined by cross-validation, choosing a balance between too few hidden layers (may not fit the training set) and too many layers (may overfit the training set).

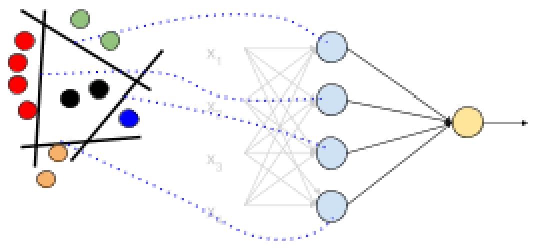
Multi-layer networks overcome limitations of single-layer networks: they are able to handle input that are not linearly-separable and arbitrarily shaped decision boundaries.



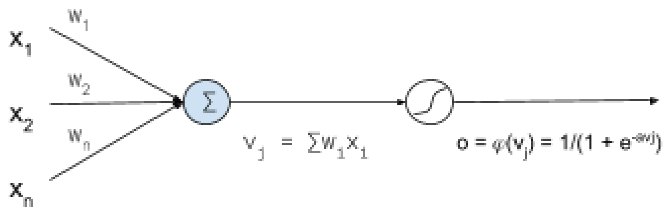
Hidden Nodes in Neural Network

Hidden nodes are the nodes in a multi-layer network that are connected between the input and the output layer. Their output is not directly visible.

The hidden layers captures pieces of information from the training data, which are then combined by the output layers to classify the instances. For example, each hidden layer in the picture below splits the training data into two sets, and their individual splits are combined in the output layer to define specific regions of the training data.



Each node is a neuron that calculates the weighted sum of the input values and has a continuous function as the activation function (usually a sigmoid function):



Output Nodes in Neural Network

The output nodes of a neural network combine the input from the hidden nodes to classify instances, resulting in a continuous value in the range [0,1]. The classification of an instance is determined by the output node with the maximum value.

As in the hidden nodes, each output node is a neuron that calculates the weighted sum of the input value and has a continuous function as the activation function (usually a sigmoid function). See picture in the previous question.

Backpropagation Rule

The Backpropagation Rule is an algorithm to train neural networks by minimizing the total error of the network using the training data.

The algorithm has two phases:

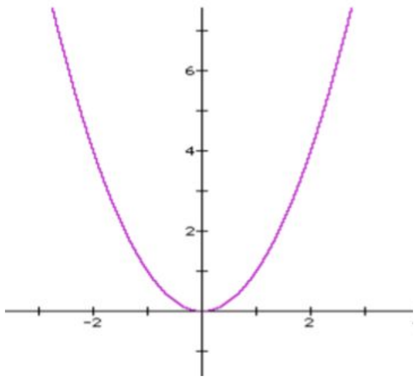
- Forward pass: calculate the output of each neuron and the error of the output neurons.
- Backward pass: starting at the output layer, propagate the error backwards, layer by layer by computing the local gradient of each neuron, with the goal of minimizing the network error (the total mean squared error). The process of distributing the error through the network elements is called "credit (or blame) assignment".

The pseudocode for backpropagation is:

```
initialize all weights to small random numbers
while the network error (E(W)) is not satisfactory and iterations < max iterations
   for each training instance do
      # This is the forward step
      # The output is the value of the sigmoid activation function for the nodes
      compute network output for that instance
      # These are the backward steps
      for each output unit $k$
         # This is the local gradient for the output unit
         # $o_k$ is the calculated output value for this output unit
         # $d_k$ is the desired value (label of the instance)
         $\delta_k \leftarrow o_k(1 - o_k)(d_k - o_k)$
      for each hidden node unit $h$
         # This is the local gradient for the hidden unit
         # $o_h$ is the calculated output value for this hidden unit
         $\delta_h \leftarrow o_h(1 - o_h)\sum_k w_{hk}\delta_k$ (where $k \in$ output units)
      update the network weights
         # $x_{ij}$ is the input value for this unit
         # note that the weight update formula is the same for hidden and output units
         $w_{ij} = w_{ij} + \Delta w_{ij}$    $(\Delta w_{ij} = \eta\delta_j x_{ij})$
   end for
end while
```

Question 2 [1.5 pts]

The following figure shows a quadratic function y=x^2. Assume we are at the point (2,4), and is searching for the next movement to find the minimum value of the quadratic function using gradient descent (the learning rate is 0.1). What is the gradient at point (2,4)? [0.5 pt] Following gradient descent principle, find the next movement towards the global minimum [1 pt]



$$\frac{\partial y}{\partial x} = \frac{\partial x^2}{\partial x} = 2x$$

At point (2,4): $\frac{\partial y}{\partial x} = 2x = 2 \cdot 2 = 4$

Next movement: $-\eta\frac{\partial y}{\partial x} = -0.1 \cdot 4 = -0.4$

Question 3 [2 pts]

Please derive Backpropagation weight update rules for a neural network with one hidden layer and one output layer (please show BP rules for both hidden nodes and output nodes). (You can borrow derivations in the course lectures, but please add detailed explanations)

The general form (for hidden and output nodes) of the backpropagation weight update rules is

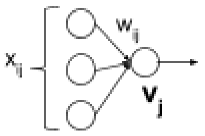
$$w_{ij} = w_{ij} + \Delta w_{ij}, \qquad \text{where} \quad \Delta w_{ij} = -\eta \frac{\partial E(W)}{\partial w_{ij}} \tag{1}$$

The partial derivative of Δw_{ij} can be solved more easily with the chain rule:

$$\frac{\partial E(W)}{\partial w_{ij}} = \frac{\partial E(W)}{\partial v_j} \frac{\partial v_j}{\partial w_{ij}} \tag{2}$$

The term v_j represents the output of a neuron:

$$v_j = \sum_{i=0\ldots m} w_{ij} x_{ij} \tag{3}$$



Whose partial derivative in respect to w_{ij} is easily calculated as

$$\frac{\partial \sum_{i=0\ldots m} w_{ij} x_{ij}}{\partial w_{ij}} = x_{ij} \tag{4}$$

Replacing (2) and (4) in Δw_{ij} from (1):

$$\Delta w_{ij} = -\eta \frac{\partial E(W)}{\partial w_{ij}} = -\eta \frac{\partial E(W)}{\partial v_j} \frac{\partial v_j}{\partial w_{ij}} = -\eta \frac{\partial E(W)}{\partial v_j} \frac{\partial \sum_{i=0\ldots m} w_{ij} x_{ij}}{\partial w_{ij}} = -\eta \frac{\partial E(W)}{\partial v_j} x_{ij} \tag{5}$$

We define the partial derivative of E(W) in (5) as the **local gradient** of neuron j as:

$$\delta_j = -\frac{\partial E(W)}{\partial v_j} \tag{6}$$

Replacing (6) in (5) we have the general definition of Δw_{ij}

$$\Delta w_{ij} = \eta \delta_j x_{ij} \tag{7}$$

To update the weight we need to compute the local gradient δ j for two different cases:

- The output neurons
- The hidden neurons

Output neurons

$$\delta_j = -\frac{\partial E(W)}{\partial v_j} = -\frac{\partial E(W)}{\partial e_j}\frac{\partial e_j}{\partial o_j}\frac{\partial o_j}{\partial v_j} \qquad (8)$$

Where:

$$E(W) = \tfrac{1}{2}(e_1^2 + e_2^2 + \cdots + e_n^2) \implies \frac{\partial E(W)}{\partial e_j} = e_1 + \cdots + e_n \text{ (n = number of output neurons)}$$

$$e_j = d_j - o_j \implies \frac{\partial e_j}{\partial o_j} = -1 \qquad (9)$$

$$o_j = \varphi(v_j) = \frac{1}{1+e^{-av_j}} \implies \frac{\partial o_j}{\partial v_j} = \varphi'(v_j) = o_j(1-o_j) \qquad (9a)$$

Substituting in (8):

$$\delta_j = -e_j(-1)\varphi'(v_j) = e_j\varphi'(v_j) \qquad (10)$$

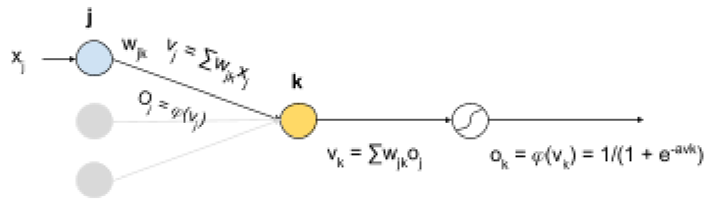
Substituting (10) in the definition of Δw_{ij} (7), using (9) to expand $e_j = (d_j - o_j)$ and (9a) to expand φ':

$$\Delta w_{ij} = \eta\delta_j x_{ij} = \eta e_j \varphi'(v_j) x_{ij} = \eta \underbrace{(d_j - o_j)}_{e_j} \cdot \underbrace{a \cdot o_j(1-o_j)}_{\varphi'(v_j)} \cdot x_{ij}$$

Where a is a factor to control the shape of the sigmoid function.

Hidden neurons

Visualization of some terms used in the following equations:



j = a hidden node
k = an output node
v_x = weighted output of xth node (before applying activation [sigmoid] function)
o_x = output of xth node (after applying activation [sigmoid] function)

$$\delta_j = -\frac{\partial E(W)}{\partial v_j} = -\sum_{k \in C} \frac{\partial E(W)}{\partial v_k}\frac{\partial v_k}{\partial v_j} \qquad \text{where C is the set of neurons of the output layer} \tag{11}$$

From (8):

$$\frac{\partial E(W)}{\partial v_k} = -\delta_k \tag{12}$$

And using the chain rule:

$$\frac{\delta v_k}{\delta v_j} = \frac{\delta v_k}{\delta o_j}\frac{\delta o_j}{\delta v_j} \tag{13}$$

Where:

$$v_k = \sum_{j=0,\ldots} w_{jk} o_j \implies \frac{\delta v_k}{\delta o_j} = w_{jk} \tag{14}$$

Substituting (9a) and (14) in (13):

$$\frac{\delta v_k}{\delta v_j} = w_{jk}\varphi'(v_j) \tag{15}$$

Then we can use (12) and (15) in (13) to get:

$$\delta_j = -\sum_{k \in C} \frac{\partial E(W)}{\partial v_k}\frac{\partial v_k}{\partial v_j} = \varphi'(v_j)\sum_{k \in C} \delta_k w_{jk} = o_j \cdot (1 - o_j)\sum_{k \in C} \delta_k w_{jk} \tag{16}$$

Substituting (16) in the definition of Δw_{ij} from (7) and using (9a) to expand φ':

$$\Delta w_{ij} = \eta \delta_j x_{ij} = \eta \cdot a \cdot x_{ij} \cdot o_j(1 - o_j) \sum_{\text{k in next layer}} \delta_k w_{jk}$$

Where a is a factor to control the shape of the sigmoid function.

Question 4 [3 pts]

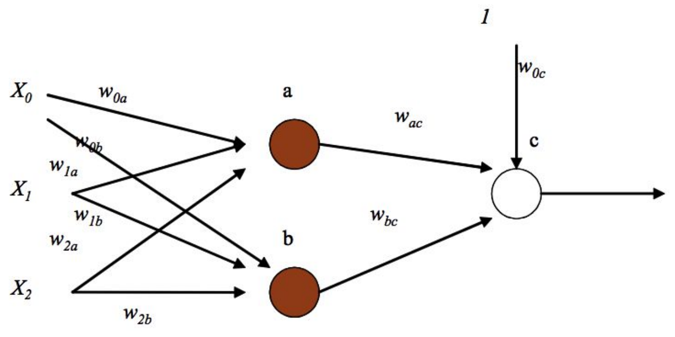
Given the following feedforward neural network with one hidden layer and one output layer, assuming the network initial weights are:

$$\begin{bmatrix} w_{0a} \\ w_{1a} \\ w_{2a} \end{bmatrix} = \begin{bmatrix} 1.0 \\ 1.0 \\ 1.0 \end{bmatrix}; \quad \begin{bmatrix} w_{0b} \\ w_{1b} \\ w_{2b} \end{bmatrix} = \begin{bmatrix} 1.0 \\ 1.0 \\ 1.0 \end{bmatrix}; \quad \begin{bmatrix} w_{0c} \\ w_{ac} \\ w_{bc} \end{bmatrix} = \begin{bmatrix} 1.0 \\ 1.0 \\ 1.0 \end{bmatrix}$$

All nodes use a sigmoid activation function with value a=2.0, and the learning rate is set to 0.5. The expected output is 1 if an instance's class label is "True", otherwise, the expected output is 0.

Given the following three instances I1, I2, I3 which are labeled as "True", "False", and "True", respectively, please calculate their actual outputs from the network? [0.5 pt]

$$I_1 = \begin{bmatrix} 1.0 \\ 1.0 \\ 0.5 \end{bmatrix}; \quad I_2 = \begin{bmatrix} 1.0 \\ 0.0 \\ 1.0 \end{bmatrix}; \quad I_3 = \begin{bmatrix} 1.0 \\ 0.5 \\ 0.5 \end{bmatrix}$$



Instance I$_1$: (1.0, 1.0, 0.5) $o_x = 1/(1 + e^{-2x})$ (for a = 2.0)

Hidden layer		Output layer
Neuron a	**Neuron b**	**Neuron c**
$v_a = w_{0a}x_o + w_{1a}x_1 + w_{2a}x_2$	$v_b = w_{0b}x_o + w_{1b}x_1 + w_{2b}x_2$	$v_c = w_{ac}o_a + w_{bc}o_b + w_{0c}(1)$
= 1(1) + 1(1) + 1(0.5) = **2.5**	= 1(1) + 1(1) + 1(0.5) = **2.5**	= 1(0.993) + 1(0.993) + 1(1) = **2.986**
$o_a = x_{ac} = 1/(1 + e^{-2va})$ = **0.993**	$o_b = x_{bc} = 1/(1 + e^{-2vb})$ = **0.993**	$o_c = 1/(1 + e^{-2vc})$ = **0.997**

Instance I$_2$: (1.0, 0.0, 1.0) $o_x = 1/(1 + e^{-2x})$ (for a = 2.0)

Hidden layer		Output layer
Neuron a	**Neuron b**	**Neuron c**
$v_a = w_{0a}x_o + w_{1a}x_1 + w_{2a}x_2$	$v_b = w_{0b}x_o + w_{1b}x_1 + w_{2b}x_2$	$v_c = w_{ac}o_a + w_{bc}o_b + w_{0c}(1)$
= 1(1) + 1(0) + 1(1) = **2**	= 1(1) + 1(0) + 1(1) = **2**	= 1(0.982) + 1(0.982) + 1(1) = **2.964**
$o_a = x_{ac} = 1/(1 + e^{-2va})$ = **0.982**	$o_b = x_{bc} = 1/(1 + e^{-2vb})$ = **0.982**	$o_c = 1/(1 + e^{-vc})$ = **0.997**

Instance I$_3$: (1.0, 0.5, 0.5) $o_x = 1/(1 + e^{-2x})$ (for a = 2.0)

Hidden layer		Output layer
Neuron a	**Neuron b**	**Neuron c**
$v_a = w_{0a}x_o + w_{1a}x_1 + w_{2a}x_2$	$v_b = w_{0b}x_o + w_{1b}x_1 + w_{2b}x_2$	$v_c = w_{ac}o_a + w_{bc}o_b + w_{0c}(1)$
= 1(1) + 1(0.5) + 1(0.5) = **2**	= 1(1) + 1(0.5) + 1(0.5) = **2**	= 1(0.982) + 1(0.982) + 1(1) = **2.964**
$o_a = x_{ac} = 1/(1 + e^{-2va})$ = **0.982**	$o_b = x_{bc} = 1/(1 + e^{-2vb})$ = **0.982**	$o_c = 1/(1 + e^{-2vc})$ = **0.997**

What is the mean squared error of the network with respect to the three instances? [0.5 pt]

$$E(W) = \frac{1}{N}\Sigma E(n) = \frac{1}{2N}\Sigma_n\Sigma_j(d_j(n) - o_j(n))^2 \text{ (network mean squared error)}$$

$$E(W) = \frac{1}{3}((1-0.997)^2 + (0-0.997)^2 + (1-0.997)^2)) = 0.332$$

Note: corrected calculation above: divided by N=3 (3 neurons), not 2x3.

Assuming instance I1 is fed to the network to update the weight, please update the network weight (using backpropagation rule) by using this instance (list major steps and results). [2.0 pts]

Instance I_1: {(1.0, 1.0, 0.5),1} η = 0.5
Note: calculations resulted in small numbers. To prevent zeroing out all values if only a few significant digits were used, the number below are in E-notation, using the full values for o_x for the calculations.

← Read from right (backpropagation) ← ←

Hidden layer		Output layer
Neuron a	**Neuron b**	**Neuron c**
$\delta_h = o_h(1 - o_h)\sum_k w_{hk}\delta_k$ (k = outputs)	$\delta_h = o_h(1 - o_h)\sum_k w_{hk}\delta_k$ (k = outputs)	$\delta_k = o_k(1 - o_k)(d_k - o_k)$
$\delta_a = o_a(1 - o_a)w_{ac}\delta_c$	$\delta_b = o_b(1 - o_b)w_{bc}\delta_c$	$\delta_c = o_c(1 - o_c)(d_c - o_c)$
= 0.993 · (1 - 0.993) · 1 · 0 = **4.28E-08**	= 0.993 · (1 - 0.993) · 1 · 0 = **4.28E-08**	= 0.997(1 - 0.997)(1 - 0.997) = **6.43E-06**
$\Delta w_{ij} = \eta\delta_j x_{ij}$	$\Delta w_{ij} = \eta\delta_j x_{ij}$	$\Delta w_{ij} = \eta\delta_j x_{ij}$
$\Delta w_{0a} = \eta\delta_a x_{0a}$ = 0.5 · 0 · 1 = **2.14E-08**	$\Delta w_{0b} = \eta\delta_b x_{0b}$ = 0.5 · 0 · 1 = **2.14E-08**	$\Delta w_{0c} = \eta\delta_c x_{0c}$ = 0.5 · 0 · 1 = **3.22E-06**
$\Delta w_{1a} = \eta\delta_a x_{1a}$ = 0.5 · 0 · 1 = **2.14E-08**	$\Delta w_{1b} = \eta\delta_b x_{1b}$ = 0.5 · 0 · 1 = **2.14E-08**	$\Delta w_{ac} = \eta\delta_c x_{ac}$ = 0.5 · 0 · 0.993 = **3.19E-06**
$\Delta w_{2a} = \eta\delta_a x_{2a}$ = 0.5 · 0 · 0.5 = **1.07E-08**	$\Delta w_{2b} = \eta\delta_b x_{2b}$ = 0.5 · 0 · 0.5 = **1.07E-08**	$\Delta w_{bc} = \eta\delta_c x_{bc}$ = 0.5 · 0 · 0.993 = **3.19E-06**
$w_{ij}(k+1) = w_{ij}(k) + \Delta w_{ij}$	$w_{ij}(k+1) = w_{ij}(k) + \Delta w_{ij}$	$w_{ij}(k+1) = w_{ij}(k) + \Delta w_{ij}$
$w_{0a}(k+1) = 1 + \Delta w_{oa}$ = **1.0000000214**	$w_{0b}(k+1) = 1 + \Delta w_{ob}$ = **1.0000000214**	$w_{0c}(k+1) = 1 + \Delta w_{0c}$ = **1.00000322**
$w_{1a}(k+1) = 1 + \Delta w_{1a}$ = **1.0000000214**	$w_{1b}(k+1) = 1 + \Delta w_{1b}$ = **1.0000000214**	$w_{ac}(k+1) = 1 + \Delta w_{ac}$ = **1.00000319**
$w_{2a}(k+1) = 1 + \Delta w_{2a}$ = **1.0000000107**	$w_{2b}(k+1) = 1 + \Delta w_{2b}$ = **1.0000000107**	$w_{bc}(k+1) = 1 + \Delta w_{bc}$ = **1.00000319**

Question 5 [2 pts]

In order to reduce the risk of neural network overfitting, one solution is to add a penalty term for the weight magnitude. For example, add a term to the squared error E that increases with the magnitude of the weight vector. This causes the gradient descent search to seek weight vectors with small magnitudes, thereby reduces the risk of overfitting. Assuming the redefined square error E is defined by

$$E(w) = \frac{1}{2}\sum_{n=1}^{N}\sum_{j \in output\,nodes}[d_j(n) - o_j(n)]^2 + \gamma\sum_{i,j} w_{i,j}^2$$

Where *N* denotes the total number of training instances, *$d_j(n)$* denotes the desired output of the *n*th instance from the *j*th output node. *$o_j(n)$* is the actual output of the *n*th instance observed from the *j*th output node. *$w_{i,j}$* is the

i^{th} weight value of the j^{th} output node. Assuming an output node *j* is using sigmoid activation function $\varphi(v_j) = \frac{1}{1+e^{-av_j}}$, please derive the backpropagation weight updating rule for the output node *j*. [Hint: use gradient descent]

The weight update rule takes a step in the opposite direction of the E:

$$w_{ij} = w_{ij} + \Delta w_{ij}, \text{ where } \Delta w_{ij} = -\eta \frac{\partial E(W)}{\partial w_{ij}}$$

The first part of this equation is the usual weight update rule. To simplify the expressions below, we will call it $E_0(W)$:

$$E_o(W) = \frac{1}{2}\sum\sum((d_j(n) - o_j(n))^2 \qquad (1)$$

E(W) derivative in relation to w_{ij} can be written as:

$$\frac{\partial E(W)}{\partial w_{ij}} = \frac{\partial E_0(W)}{\partial w_{ij}} + \frac{\partial \sum \gamma \sum w_{i,j}^2}{\partial w_{ij}} \qquad (2)$$

The second part of this equation can be developed as:

$$\frac{\partial \gamma \sum w_{i,j}^2}{\partial w_{ij}} = \frac{\partial \gamma \sum w_{i,j}^2}{\partial w_{ij}} = \gamma \frac{\partial \sum w_{i,j}^2}{\partial w_{ij}} = 2\gamma w_{i,j} \qquad (3)$$

Using (1), (2) and (3):

$$\Delta w_{ij} = -\eta \frac{\partial E(W)}{\partial w_{i,j}} = -\eta\left(\frac{\partial E_0(W)}{\partial w_{ij}} + 2\gamma w_{ij}\right) = -\eta \frac{\partial E_0(W)}{\partial w_{ij}} - \eta 2\gamma w_{ij}$$

$$w_{ij} = w_{ij} - \eta \frac{\partial E_0(W)}{\partial w_{ij}} - 2\eta\gamma w_{ij}$$

The first part of Δw_{ij} (in terms of $E_0(W)$) is the usual weight update rule, developed in question 3.

Note: corrected the answer to match the professor's answer. The second term should not be divided by ½, as I did originally.

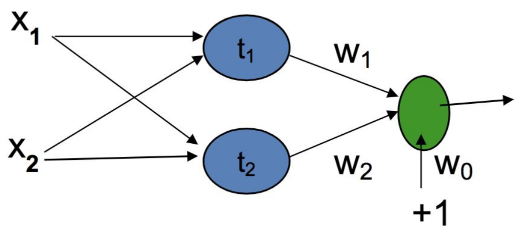
Question 6 [3 pts]

When building an RBF network to solve the XOR problem with input and output given as follows, where X_1 and X_2 are features and Y is the output.

X_1	X_2	Y

0	0	-1
1	0	1
0	1	1
1	1	-1

Assume the RBF network is showing as follows, and we use Gaussian RBF function with σ=1, and t_1=(0.1, 0.1), t_2=(0.9, 0.9). Please use pseudo-inverse to calculate the weight values W=[w_0, w_2, w_3] of the output node [2 pts], and validate your results with respect to the four instances [1 pt]



$$\varphi_1 = exp\left(\frac{-||x-t_1||^2}{\sigma^2}\right) = exp\left(\frac{-||x-t_1||^2}{1^2}\right) = exp(-||x-t_1||^2)$$

$$\varphi_2 = exp\left(\frac{-||x-t_2||^2}{\sigma^2}\right) = exp\left(\frac{-||x-t_2||^2}{1^2}\right) = exp(-||x-t_2||^2)$$

x_1	x_2	**φ_1**	**φ_2**
0	0	exp(- || (0,0)-(0.1,0.1) || ^{2}) = exp(- || (0-0.1)^2 + (0-0.1)^2 || ^{2}) = **0.980**	exp(- || (0,0)-(0.9,0.9) || ^{2}) = exp(- || (0-0.9)^2 + (0-0.9)^2 || ^{2}) = **0.198**
1	0	exp(- || (1,0)-(0.1,0.1) || ^{2}) = exp(- || (1-0.1)^2 + (0-0.1)^2 || ^{2}) = **0.440**	exp(- || (1,0)-(0.9,0.9) || ^{2}) = exp(- || (1-0.9)^2 + (0-0.9)^2 || ^{2}) = **0.440**
0	1	exp(- || (0,1)-(0.1,0.1) || ^{2}) = exp(- || (0-0.1)^2 + (1-0.1)^2 || ^{2}) = **0.440**	exp(- || (0,1)-(0.9,0.9) || ^{2}) = exp(- || (0-0.9)^2 + (1-0.9)^2 || ^{2}) = **0.440**
1	1	exp(- || (1,1)-(0.1,0.1) || ^{2}) = exp(- || (1-0.1)^2 + (1-0.1)^2 || ^{2}) = **0.198**	exp(- || (1,1)-(0.9,0.9) || ^{2}) = exp(- || (1-0.9)^2 + (1-0.9)^2 || ^{2}) = **0.980**

$$\begin{bmatrix} 1 & \varphi_1(||X_1-t_1||) & \varphi_2(||X_1-t_2||) \\ 1 & \varphi_1(||X_2-t_1||) & \varphi_2(||X_2-t_2||) \\ 1 & \varphi_1(||X_3-t_1||) & \varphi_2(||X_3-t_2||) \\ 1 & \varphi_1(||X_4-t_1||) & \varphi_2(||X_4-t_2||) \end{bmatrix} \times \begin{bmatrix} w_0 & w_1 & w_2 \end{bmatrix}^T = \begin{bmatrix} Y_1 \\ Y_2 \\ Y_3 \\ Y_4 \end{bmatrix}^T$$

$$\begin{bmatrix} 1 & 0.980 & 0.198 \\ 1 & 0.440 & 0.440 \\ 1 & 0.440 & 0.440 \\ 1 & 0.165 & 0.980 \end{bmatrix} \times \begin{bmatrix} w_0 & w_1 & w_2 \end{bmatrix}^T = \begin{bmatrix} -1 \\ 1 \\ 1 \\ -1 \end{bmatrix}^T$$

$$\begin{bmatrix} w_0 & w_1 & w_2 \end{bmatrix}^T = \Phi^+ \times \begin{bmatrix} -1 \\ 1 \\ 1 \\ -1 \end{bmatrix}^T$$

$$\begin{bmatrix} w_0 & w_1 & w_2 \end{bmatrix}^T = \begin{bmatrix} -1.477 & 1.977 & 1.977 & -1.477 \\ 2.317 & -1.678 & -1.678 & 1.038 \\ 1.038 & -1.678 & -1.678 & 2.317 \end{bmatrix} \times \begin{bmatrix} -1 \\ 1 \\ 1 \\ -1 \end{bmatrix}^T$$

$$\begin{bmatrix} w_0 & w_1 & w_2 \end{bmatrix}^T = \begin{bmatrix} 6.908 \\ -6.711 \\ -6.711 \end{bmatrix}$$

Validating the instances:

X_1	X_2	$\sum w_i \varphi_i$	Predicted Y
0	0	1 x (6.908) + 0.980 x (-6.711) + 0.198 x (-6.711) = -0.998 < 0	**-1**
1	0	1 x (6.908) + 0.440 x (-6.711) + 0.440 x (-6.711) = 0.996 > 0	**1**
0	1	1 x (6.908) + 0.440 x (-6.711) + 0.440 x (-6.711) = 0.996 > 0	**1**
1	1	1 x (6.908) + 0.198 x (-6.711) + 0.980 x (-6.711) = -0.998 < 0	**-1**

Question 7 [4 pts]

The Face.Recognition.R file in the canvas includes R code which trains multi-layer neural networks from pgm files to classify faces into different categories. The course lecture “CAP5615.R.Multi.Layer.NN.pdf” explains detailed procedures about how to load pgm files to train Neural Network classifiers. In addition, the CMU Face Recognition website (http://www.cs.cmu.edu/~tom/faces.html) includes faces with different directions, facial expression etc. (http://www.cs.cmu.edu/afs/cs.cmu.edu/project/theo-8/faceimages/faces/). Please design and implement following face recognition task using Neural Networks.

Please down at least 100 face images from CMU website (you can also download images from other sites) to build a two class classification tasks (50 images for each class). You must specify faces in the positive and negative class (e.g., faces turning left are positive, and faces turning right are negative). You must include all faces in your final submission [1 pt]

Please see attached file cap5615-homework4-question7.R.

The selected classes are:

- Right - class = 1
- Left - class = 0

Please show one example of the face in each class (using R) [1 pt].

Example of right and left classes, respectively:

Class 1 - right

Class 0 - left

R code used to show them:

```
class1_sample <- read.pnm(file = "an2i_right_sad_open_4.pgm", cellres=1)
plot(class1_sample)
class0_sample <- read.pnm(file = "saavik_left_happy_open_4.pgm", cellres=1)
plot(class0_sample)
```

Please randomly select 80% of images as training samples to train neural networks with one hidden layer but different number of hidden nodes (3, 5, 7, 9, 11 hidden nodes, respectively). Please show the classification accuracies of the neural networks on the remaining 20% of images (which are not selected as training samples). Please report R code [2 pt], and also use a table to summarize the classification accuracy of the neural networks with respect to different number of hidden nodes [1 pt].

The accuracy vs. number of hidden nodes:

Number of hidden nodes

	3	5	7	9	11
Accuracy	95%	100%	100%	100%	100%

Confusion tables are shown below. Note: the tables below use 0.5 as the cutoff point to assign classes (i.e. when predicted value is ≥ 0.5 → class = 1, otherwise class = 0).

```
> classification3 <- classify_faces(training_set, test_set, 3)
> show_confusion_matrix("3 hidden nodes", test_set, classification3)
[1] "3 hidden nodes"
```

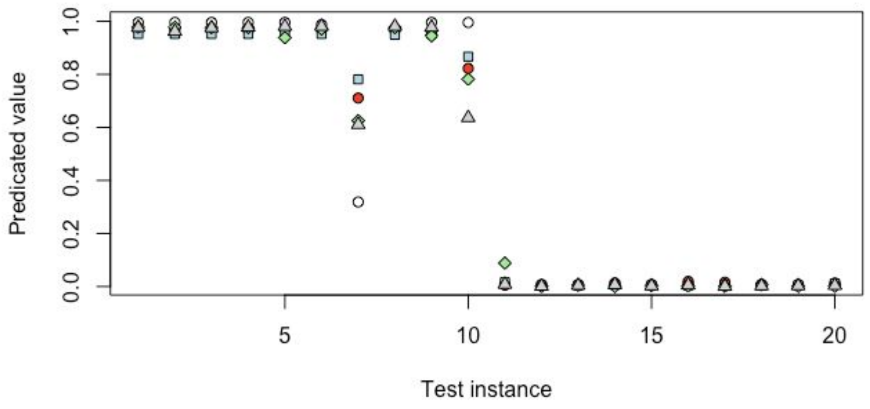
```
         actual
predicted  0  1
        0 10  1
        1  0  9
[1] 0.95

> classification5 <- classify_faces(training_set, test_set, 5)
> show_confusion_matrix("5 hidden nodes", test_set, classification5)
[1] "5 hidden nodes"
         actual
predicted  0  1
        0 10  0
        1  0 10
[1] 1

> classification7 <- classify_faces(training_set, test_set, 7)
> show_confusion_matrix("7 hidden nodes", test_set, classification7)
[1] "7 hidden nodes"
         actual
predicted  0  1
        0 10  0
        1  0 10
[1] 1

> classification9 <- classify_faces(training_set, test_set, 9)
> show_confusion_matrix("9 hidden nodes", test_set, classification9)
[1] "9 hidden nodes"
         actual
predicted  0  1
        0 10  0
        1  0 10
[1] 1

> classification11 <- classify_faces(training_set, test_set, 11)
> show_confusion_matrix("11 hidden nodes", test_set, classification11)
[1] "11 hidden nodes"
         actual
predicted  0  1
        0 10  0
        1  0 10
[1] 1
```

Since the accuracy of most models are the same, the tables below show the continuous predicted values, to have a better idea of how more neurons in the hidden layer affect accuracy. The red item is the misclassified test instance when we use three hidden nodes.

The bottom row shows the MSE for each number of hidden nodes. Beyond seven nodes the MSE starts to increase, which indicates that for this training set we should use seven hidden nodes, since beyond that we don't seem to gain much more accuracy (for the given training set).

Class	3	5	7	9	11
1	0.995802056	0.969442852	0.952144524	0.977541178	0.976356638
1	0.995802056	0.969356266	0.952036559	0.977629723	0.961886796
1	0.995802042	0.969254163	0.952088541	0.976083995	0.975024169
1	0.995801804	0.966117810	0.952134390	0.976754600	0.977663675
1	0.995790631	0.969269505	0.952031265	0.938304256	0.980590799
1	0.987602734	0.968180457	0.952136353	0.971006914	0.980595653
1	0.318746059	0.710520230	0.781054466	0.625270892	0.610488014
1	0.965919526	0.969339695	0.948921899	0.976407092	0.980551862
1	0.995802056	0.959351015	0.952091489	0.945004790	0.977180951
1	0.995247853	0.822325158	0.866741092	0.781795798	0.636443408
0	0.007404953	0.005031534	0.015976254	0.088223743	0.007932649
0	0.007404280	0.003078835	0.004038268	0.000395817	0.000973482
0	0.007404289	0.003487380	0.004028929	0.004956037	0.003748423
0	0.007411650	0.013214583	0.005622960	0.000776028	0.006516064
0	0.007415446	0.003445592	0.004253934	0.003654172	0.002106737
0	0.007412347	0.019373291	0.004101691	0.001716864	0.005121257
0	0.007404280	0.016032525	0.004083969	0.000592768	0.000570548
0	0.007404302	0.006927695	0.004222345	0.006478630	0.002018368
0	0.007404281	0.006126220	0.004613304	0.000707586	0.002558230
0	0.007404392	0.012866356	0.010671759	0.001132540	0.003522339
MSE=	**0.00722**	**0.00312**	**0.002122**	**0.00516**	**0.00722**

The scatterplot below shows how the different number of hidden nodes classified the instances. The larger errors are in the class 1 instances. Those instance are the largest contributors to the MSE. In this case it seems to be an indication of a small training set (a small number of instances have a large effect on the network), or perhaps poorly selected instance for that category (e.g. image is not properly segmented).


Predicated value
Test instance