8. Multilayer Perceptron Neural Networks

8.1 Introduction

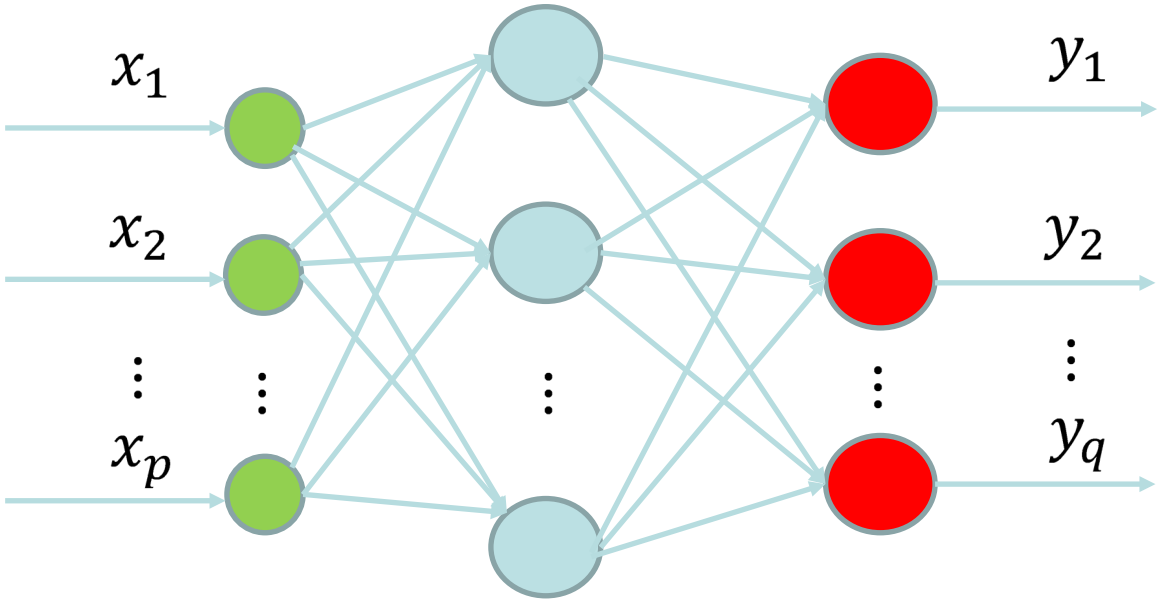
In this part, we are to study the multilayer feed-forward network, which is an important class of neural networks. Typically, the network consists of a set of inputs (source nodes) that constitute the input layer, one or more hidden layers of computational nodes, and an output layer of computational nodes. The input signals propagate through the network in a forward direction, on a layer-by-layer basis. These neural networks are commonly referred to as multilayer perceptron (MLP) neural network.

A MLP neural network has three distinctive characteristics as summarized below:

(1) Smooth activation functions such as sigmoid function are used.

(2) The network contains one or more hidden layers that are not part of the input or output of the network. The hidden neurons enable the network to learn complex tasks.

(3) The network exhibits a high degree of connectivity (fully connected).



8.2 Preliminary: 1-D gradient descent method

Suppose we have a 1-D minimization problem:

$$J(\mathrm{x}) = (\mathrm{a} - \mathrm{bx})^2$$

Where *a* and *b* are parameters. The goal of the optimization problem is to find a value of x so that *J*(x) is minimized.

To minimize the cost function *J*(x), we take the first-order derivative:

$$\frac{\partial J(\mathrm{x})}{\partial \mathrm{x}} = -2\mathrm{b}(\mathrm{a} - \mathrm{bx})$$

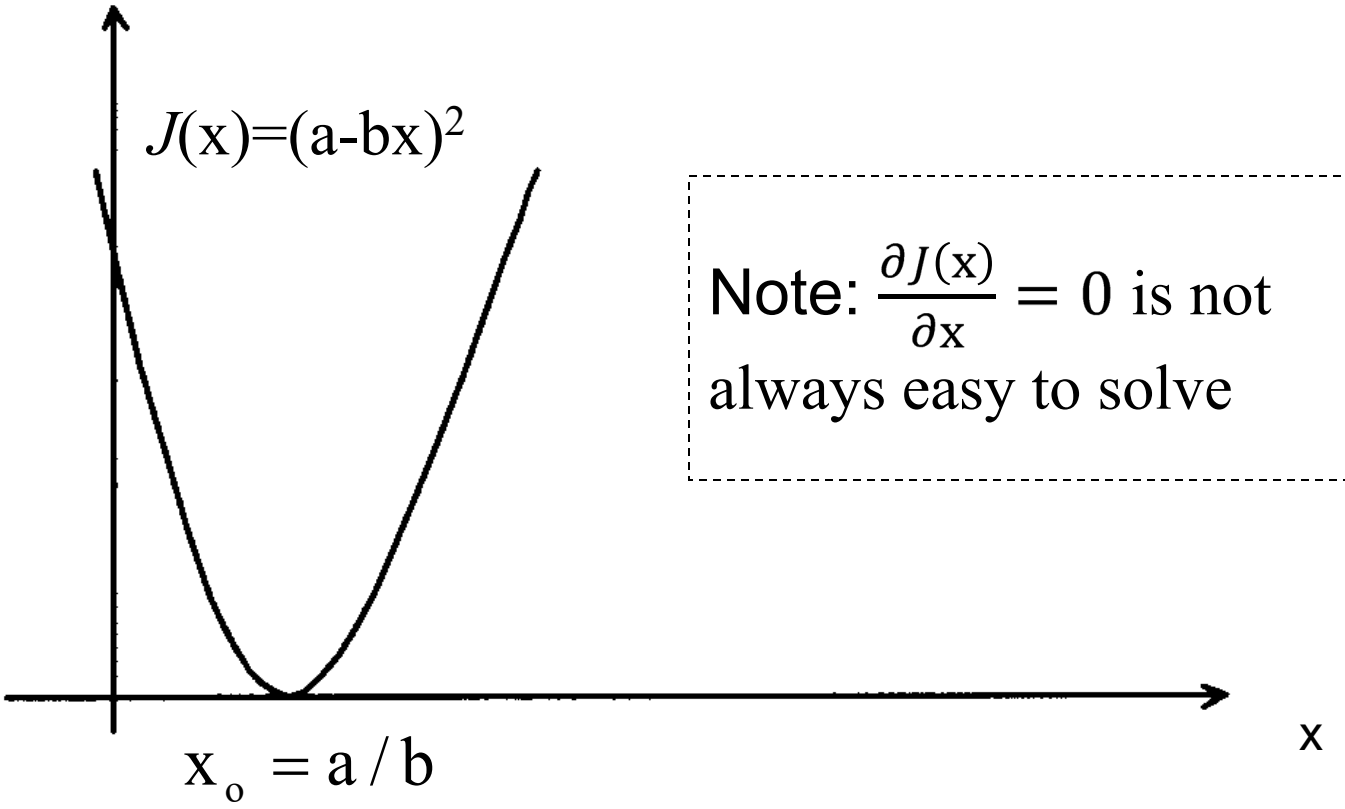
Solve the equation:

$$\frac{\partial J(\mathrm{x})}{\partial \mathrm{x}} = 0$$

We obtain the optimal solution:

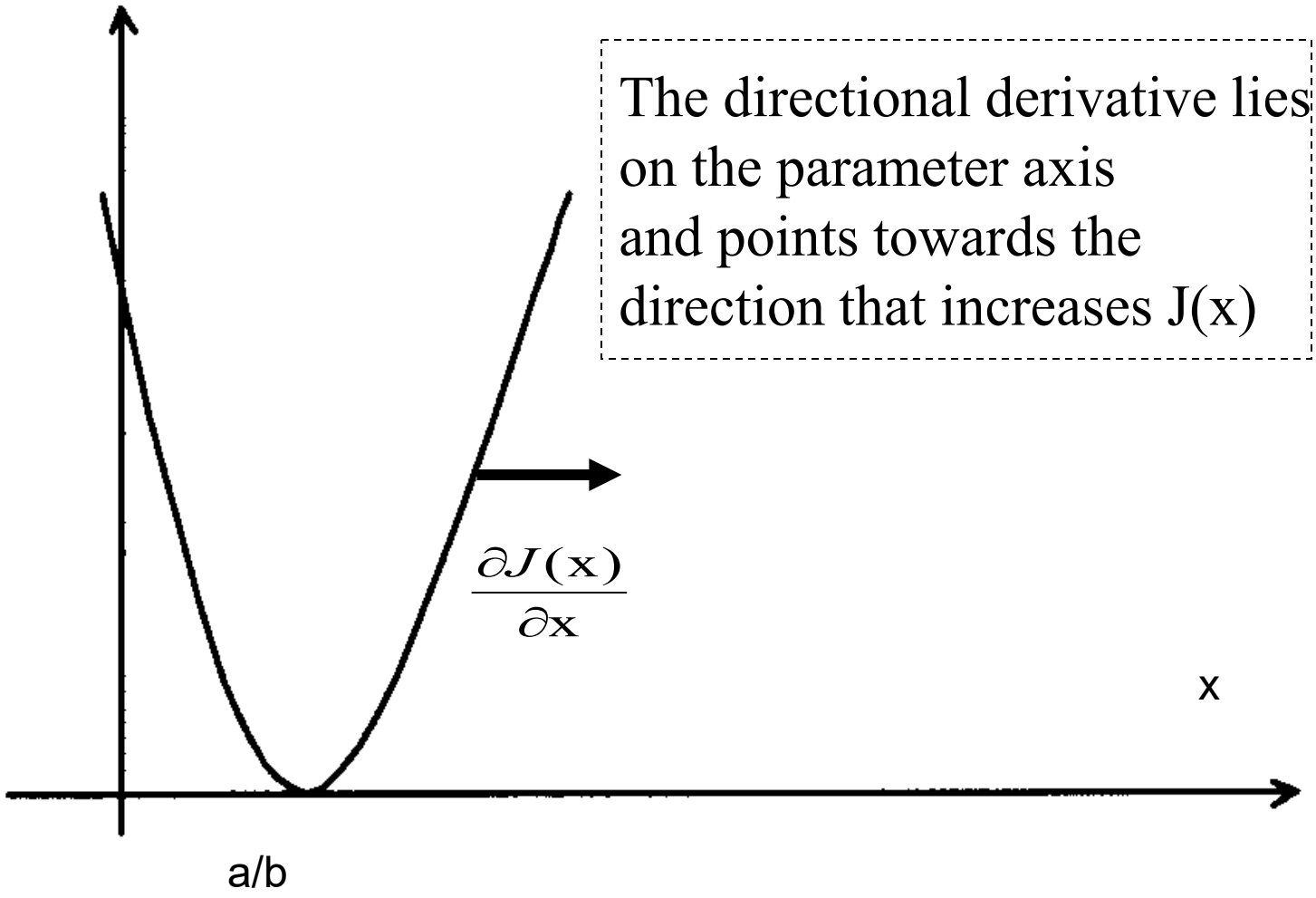
$$x_0 = a / b$$

The following is the graphical illustration:

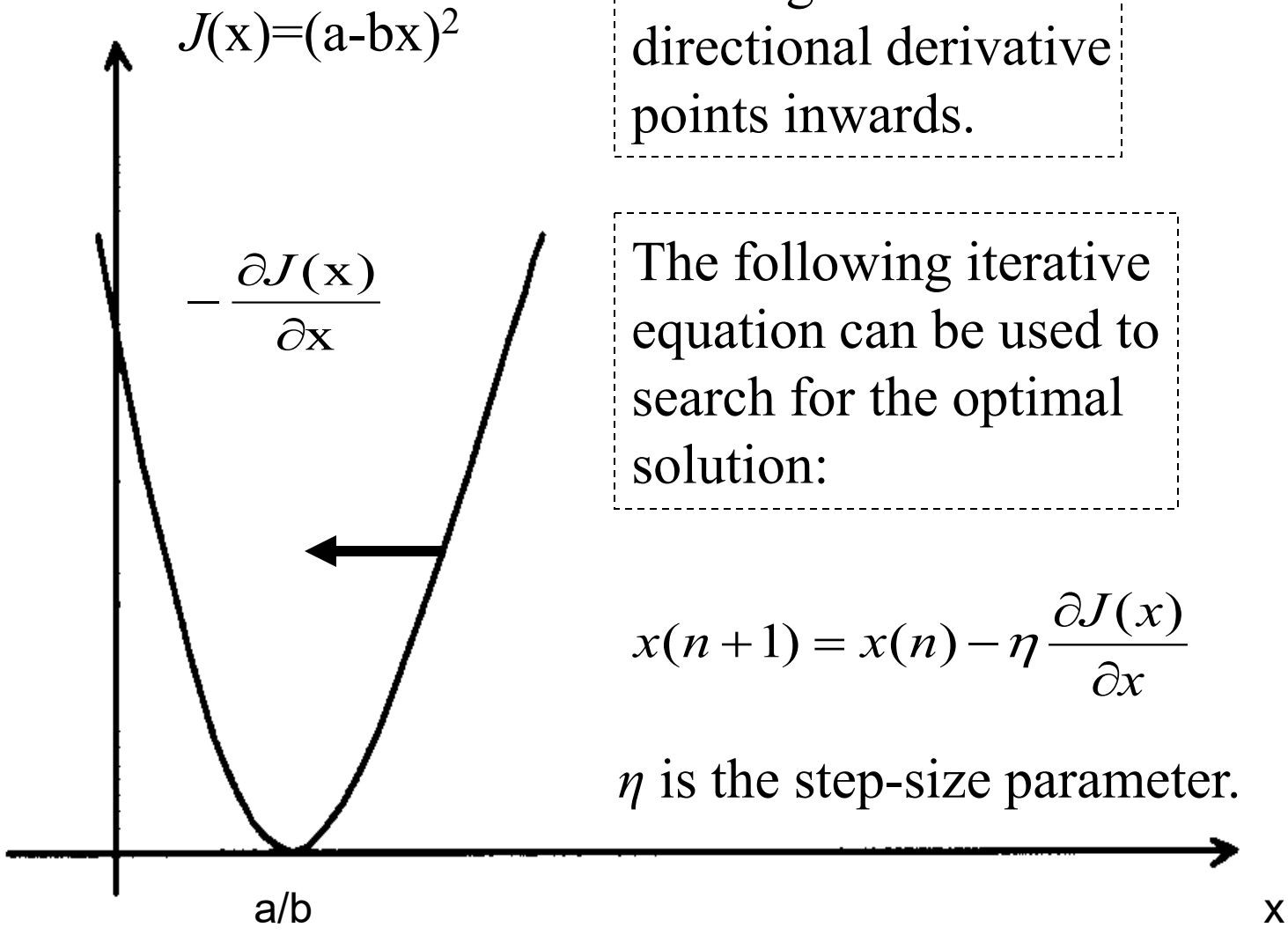


Note: $\frac{\partial J(\mathrm{x})}{\partial \mathrm{x}} = 0$ is not always easy to solve

$J(x)=(a-bx)^2$



The directional derivative lies on the parameter axis and points towards the direction that increases J(x)



The negative of the directional derivative points inwards.

The following iterative equation can be used to search for the optimal solution:

$$x(n+1) = x(n) - \eta \frac{\partial J(x)}{\partial x}$$

η is the step-size parameter.

The optimal x can be found by using the following iterative equation:

$$\Delta x(n) = -\eta \frac{\partial J(x)}{\partial x}$$

$$x(n+1) = x(n) + \Delta x(n) = x(n) - \eta \frac{\partial J(x)}{\partial x}$$

Where η is the step-size parameter.

Next, the 1-D gradient-descent method will be extended to multi-D to address the weight estimation problem of MLP neural networks.

8.3 MLP neural network learning

Given a set of training (learning) samples:

$$\{\mathbf{x}(1), \mathbf{d}(1)\}, \{\mathbf{x}(2), \mathbf{d}(2)\}, \cdots, \{\mathbf{x}(N), \mathbf{d}(N)\}$$

where

$$\mathbf{x}(i) = [x_1(i), x_2(i), \cdots, x_p(i)]^T$$

$$\mathbf{d}(i) = [d_1(i), d_2(i), \cdots, d_q(i)]^T$$

The objective of learning is to build a MLP neural network so that the network response approximates the desired response **d**(i) well:

$$\mathbf{y(i)} \rightarrow \mathbf{d}(i)$$

Construction of a MLP neural network involves the following:

(1)Determination of the MLP neural network architecture

- The number of neurons in the input layer
  - Should be the same as the dimension of the input vector, i.e. p.
- The number of neurons in the output layer
  - Should be the same as the dimension of the target vector, i.e. q.
- The number of hidden layers
  - Usually set to 1
- The number of neurons in the hidden layers
  - This is a hyper-parameter and is often determined by trial and error. The general rule: more neurons for more complicated problems.

(2) **Determination of the weights in the neural network**

- Back-propagation (BP) algorithm

The error signal of the output neuron j when the n-th training sample is presented to the network is defined by:

$$e_j(n) = d_j(n) - y_j(n) \tag{8.1}$$

Where $d_j(n)$and $y_j(n)$ denote the desired response and actual response of neuron j.

The total squared error at iteration (step) n is defined as:

$$E(n) = \frac{1}{2}\sum_{j=1}^{q} e_j^2(n) \tag{8.2}$$

Where q is the number of neurons of the output layer.

The objective of neural network learning is to find such weights that E(n) is minimized.

In back-propagation (BP) algorithm, the weights are learned step by step based on the gradient-descent method, where weight update is proportional to the partial derivative (i.e. gradient) of E(n) with respect to the weight:

$$\Delta w_{ji}(n) = -\eta \frac{\partial E(n)}{\partial w_{ji}(n)}$$

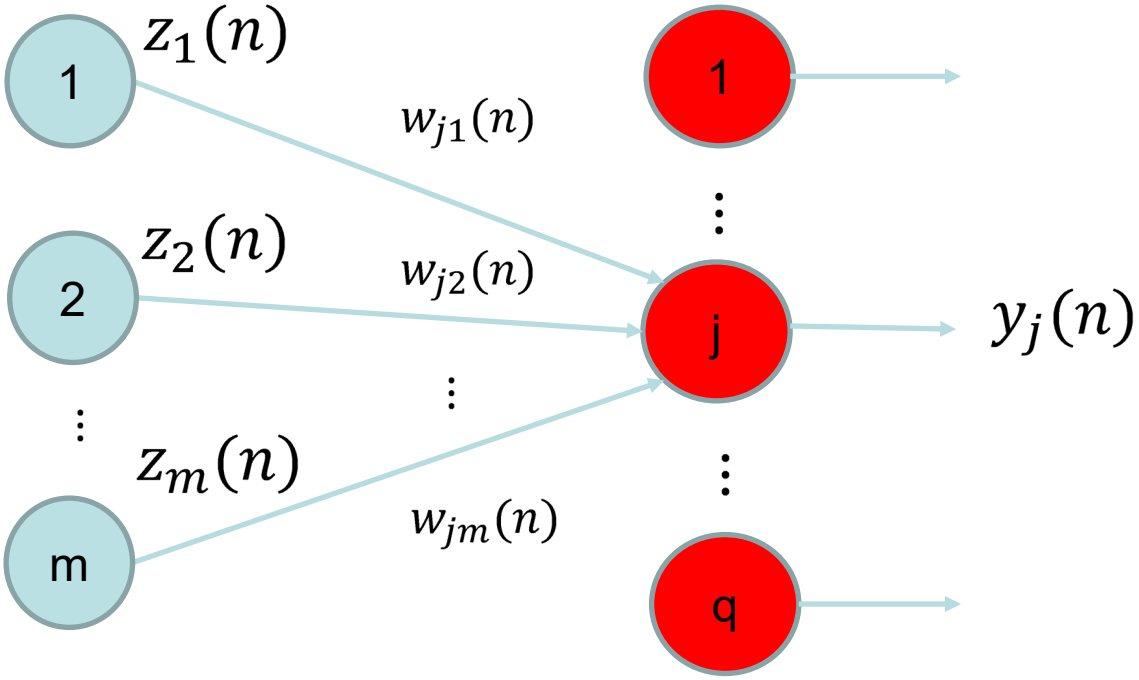
Where η is the learning rate (step size).

To derive $\frac{\partial E(n)}{\partial w_{ji}(n)}$, we consider the following two cases:

Case 1: neurons at the output layer

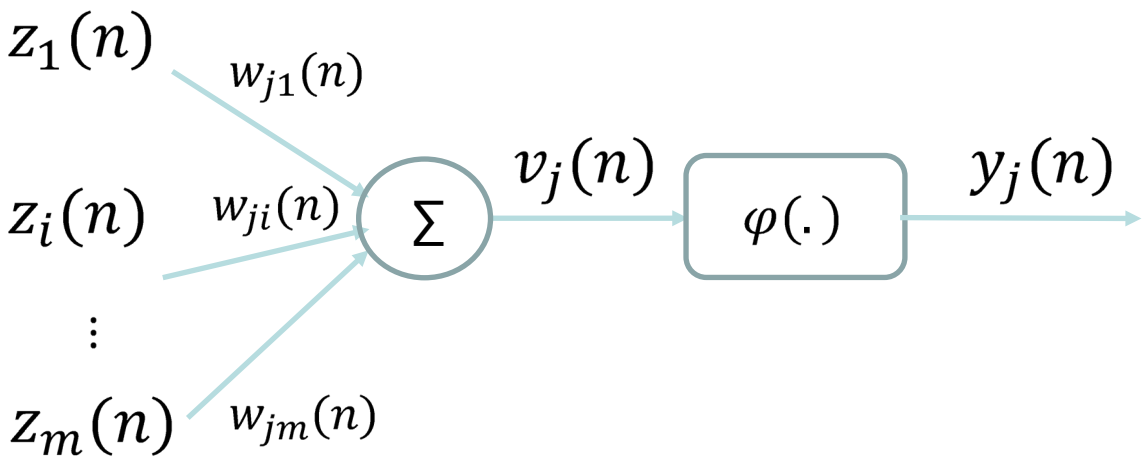
Case 2: neurons at the hidden layer

Case 1: neuron *j* is at the output layer:



Where $z_i(n)$ is the output of hidden layer neuron *i*.

Recall the neuron model:



Activation:

$$v_j(n) = \sum_{i=1}^{m} w_{ji}(n) z_i(n) \tag{8.3}$$

Output:

$$y_j(n) = \varphi\left[v_j(n)\right] \tag{8.4}$$

According to the chain rule of calculus, we may express this derivative (gradient) as:

$$\frac{\partial E(n)}{\partial w_{ji}(n)} = \frac{\partial E(n)}{\partial e_j(n)} \frac{\partial e_j(n)}{\partial y_j(n)} \frac{\partial y_j(n)}{\partial v_j(n)} \frac{\partial v_j(n)}{\partial w_{ji}(n)} \tag{8.5}$$

The partial derivative represents a sensitivity factor that determines the direction of search in weight space for the weight $w_{ji}(n)$.

Differentiating both sides of Eqn. (8.2) with respect to $e_j(n)$, we get:

$$\frac{\partial E(n)}{\partial e_j(n)} = e_j(n) \tag{8.6}$$

Differentiating both sides of Eqn. (8.1) with respect to $y_j(n)$, we get:

$$\frac{\partial e_j(n)}{\partial y_j(n)} = -1 \tag{8.7}$$

Differentiating Eqn. (8.4) with respect to $v_j(n)$, we obtain:

$$\frac{\partial y_j(n)}{\partial v_j(n)} = \varphi'\left[v_j(n)\right] \tag{8.8}$$

Differentiating Eqn. (8.3) with respect to $w_{ji}(n)$, we obtain:

$$\frac{\partial v_j(n)}{\partial w_{ji}(n)} = z_i(n) \tag{8.9}$$

Substituting Eqns. (8.6)-(8.9) into Eqn. (8.5), yields:

$$\frac{\partial E(n)}{\partial w_{ji}(n)} = -e_j(n)\varphi'\big[v_j(n)\big]\, z_i(n) \tag{8.10}$$

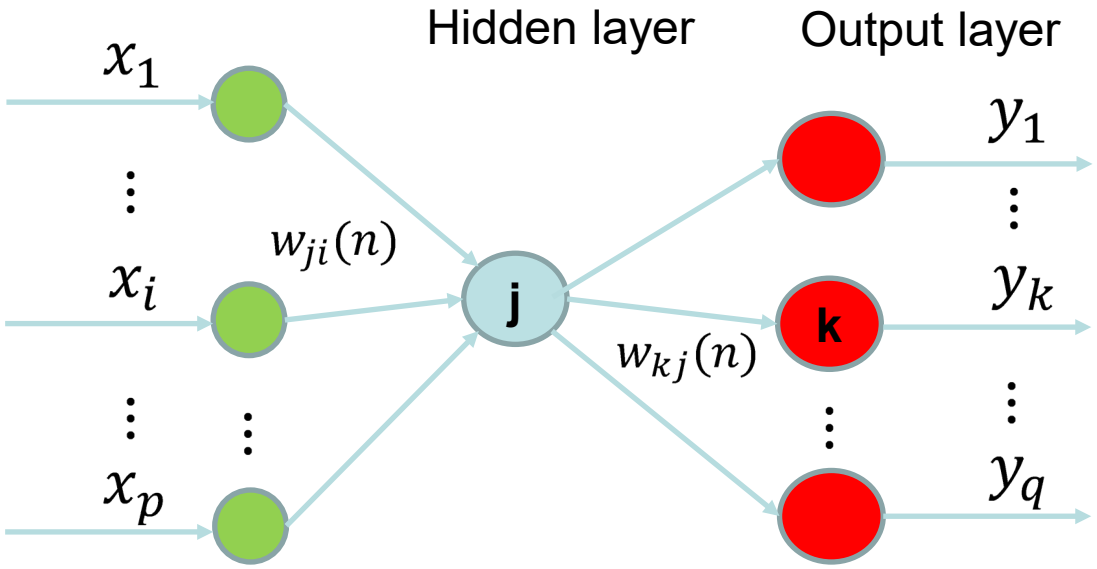
The weight update at step n is then given by:

$$\begin{aligned}\Delta w_{ji}(n) &= -\eta \frac{\partial E(n)}{\partial w_{ji}(n)} \\ &= \eta e_j(n)\, \varphi'\big[v_j(n)\big] z_i(n) \\ &= \eta \delta_j(n) z_i(n)\end{aligned} \tag{8.11}$$

Where η is the learning rate parameter, and $\delta_j(n)$ is the local gradient defined by:

$$\delta_j(n) = e_j(n)\, \varphi'\big[v_j(n)\big] \tag{8.12}$$

Case 2: neuron *j* is at the hidden layer



Obviously, $w_{ji}(n)$ influences all the outputs in the output layer: $y_1(n), y_2(n)$, …, and hence all error signals: $e_1(n), e_2(n)$, ….

By contrast, when neuron j is at the output layer, $w_{ji}(n)$ only influences $y_j(n)$ and $e_j(n)$.

$$
\begin{aligned}
E(n) &= \frac{1}{2}\sum_{k=1}^{q} e_k^2(n) \\
&= \frac{1}{2}\sum_{k=1}^{q} [d_k(n) - y_k(n)]^2 \\
&= \frac{1}{2}\sum_{k=1}^{q} \{d_k(n) - \varphi[v_k(n)]\}^2 \\
&= \frac{1}{2}\sum_{k=1}^{q} \left\{d_k(n) - \varphi\left[\sum_{j=1}^{m} w_{kj}(n) z_j(n)\right]\right\}^2 \\
&= \frac{1}{2}\sum_{k=1}^{q} \left\{d_k(n) - \varphi\left[\sum_{j=1}^{m} w_{kj}(n) \varphi(v_j(n))\right]\right\}^2 \\
&= \frac{1}{2}\sum_{k=1}^{q} \left\{d_k(n) - \varphi\left[\sum_{j=1}^{m} w_{kj}(n) \varphi(\sum_{i=1}^{p} w_{ji}(n) x_i(n))\right]\right\}^2
\end{aligned}
$$

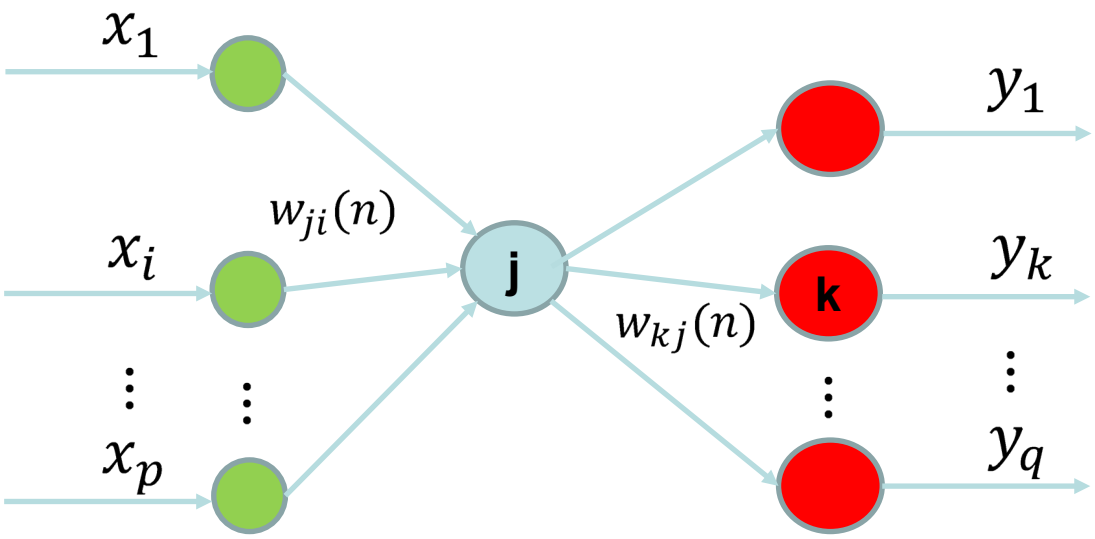
$$\frac{\partial E(n)}{\partial w_{ji}(n)} = \sum_{k=1}^{q} e_k \frac{\partial e_k(n)}{\partial w_{ji}(n)}$$
$$= \sum_{k=1}^{q} e_k \frac{\partial e_k(n)}{\partial y_k(n)} \frac{\partial y_k(n)}{\partial v_k(n)} \frac{\partial v_k(n)}{\partial w_{ji}(n)}$$
$$= \sum_{k=1}^{q} -e_k(n)\varphi'[v_k(n)] \frac{\partial v_k(n)}{\partial w_{ji}(n)} \quad (8.13)$$

Recall the local gradient for the neuron k in the output layer Eqn. (8.12):

$$\delta_k(n) = e_k(n)\, \varphi'[v_k(n)]$$

Hence, we have:

$$\frac{\partial E(n)}{\partial w_{ji}(n)} = -\sum_{k=1}^{q} \delta_k(n) \frac{\partial v_k(n)}{\partial w_{ji}(n)} \quad (8.14)$$



Hence, we have:

$$\begin{aligned}\frac{\partial v_k(n)}{\partial w_{ji}(n)} &= w_{kj}(n)\frac{\partial z_j(n)}{\partial w_{ji}(n)} \\ &= w_{kj}(n)\frac{\partial z_j(n)}{\partial v_j(n)}\frac{\partial v_j(n)}{\partial w_{ji}(n)} \\ &= w_{kj}(n)\varphi'\left[v_j(n)\right]x_i(n)\end{aligned} \tag{8.15}$$

$$\frac{\partial E(n)}{\partial w_{ji}(n)} = -\sum_{k=1}^{q} \delta_k(n) \frac{\partial v_k(n)}{\partial w_{ji}(n)}$$
$$= -\sum_{k=1}^{q} \delta_k(n) w_{kj}(n) \varphi'\big[v_j(n)\big] x_i(n)$$

Define the local gradient for a hidden neuron j as:

$$\delta_j(n) = \varphi'\big[v_j(n)\big] \sum_{k=1}^{q} w_{kj}(n) \delta_k(n) \qquad (8.16)$$

Then the gradient is obtained as:

$$\frac{\partial E(n)}{\partial w_{ji}(n)} = -\delta_j(n) x_i(n) \qquad (8.17)$$

The weight is updated as:

$$\Delta w_{ji}(n) = -\eta \frac{\partial E(n)}{\partial w_{ji}(n)} = \eta \delta_j(n) x_i(n) \qquad (8.18)$$

Activation function

The computation of the local gradient and hence weight update of each neuron of the MLP neural network requires the derivative of the activation function. For this derivative to exist, the activation function needs to be continuous and differentiable. An example of a continuously differentiable nonlinear activation function commonly used in MLP neural network are sigmoid function and hyperbolic tangent function.

(1) Sigmoid function

$$\varphi[v_j(n)] = \frac{1}{1 + \exp[-v_j(n)]} \tag{8.19}$$

Where $v_j(n)$ is the activation signal of neuron j.

According to this nonlinearity, the amplitude of the output lies in the range of:

$$0 \le y_j(n) \le 1$$

Differentiating Eqn. (8.19) with respect to $v_j(n)$, we get:

$$\varphi'[v_j(n)] = \frac{\exp[-v_j(n)]}{[1+\exp[-v_j(n)]]^2} \tag{8.20}$$

Considering $y_j(n) = \varphi\big[v_j(n)\big]$, we may express Eqn. (8-20) into:

$$\begin{aligned}\varphi'\big[v_j(n)\big] &= \frac{1}{1+\exp\big[-v_j(n)\big]} \times \frac{\exp\big[-v_j(n)\big]}{1+\exp\big[-v_j(n)\big]} \\ &= \frac{1}{1+\exp\big[-v_j(n)\big]} \times \left[1 - \frac{1}{1+\exp\big[-v_j(n)\big]}\right] \\ &= y_j(n)[1-y_j(n)]\end{aligned} \tag{8.21}$$

Case 1: For an arbitrary <u>neuron *j* located in the output layer</u>, the local gradient maybe expressed as:

$$\begin{aligned}\delta_j(n) &= e_j(n)\varphi'[v_j(n)] \\ &= [d_j(n) - y_j(n)]y_j(n)[1 - y_j(n)]\end{aligned} \qquad (8.22)$$

Case 2: For an arbitrary <u>hidden neuron *j*</u>, we may express the local gradient as:

$$\begin{aligned}\delta_j(n) &= \varphi'[v_j(n)] \sum_k \delta_k(n) w_{kj}(n) \\ &= y_j(n)[1 - y_j(n)] \sum_k \delta_k(n) w_{kj}(n)\end{aligned} \qquad (8.23)$$

(2) Hyperbolic tangent function

Hyperbolic tangent function is another commonly used activation function defined by:

$$\begin{aligned}\varphi[v_j(n)] &= \tanh[\,v_j(n)] \\ &= \frac{\exp[\,v_j(n)] - \exp[-v_j(n)]}{\exp[\,v_j(n)] + \exp[-v_j(n)]}\end{aligned} \tag{8.24}$$

The derivative of the hyperbolic tangent function with respect to $v_j(n)$ is given by:

$$\begin{aligned}\varphi'[v_j(n)] &= \operatorname{sech}^2[v_j(n)] \\ &= \left(1 - \tanh^2[\,v_j(n)]\right) \\ &= [1 - y_j(n)][1 + y_j(n)]\end{aligned} \tag{8.25}$$

Case 1: For a <u>neuron *j* located in the output layer</u>, the local gradient is:

$$\begin{aligned}\delta_j(n) &= e_j(n)\varphi'[v_j(n)] \\ &= [d_j(n) - y_j(n)][1 - y_j(n)][1 + y_j(n)]\end{aligned} \qquad (8.26)$$

Case 2: For a <u>neuron *j* located in a hidden layer</u>, the local gradient is:

$$\begin{aligned}\delta_j(n) &= \varphi'[v_j(n)] \sum_k \delta_k(n) w_{kj}(n) \\ &= [1 - y_j(n)][1 + y_j(n)] \sum_k \delta_k(n) w_{kj}(n)\end{aligned} \qquad (8.27)$$

Weight update rule:

$$\begin{pmatrix} weight \\ update \\ \Delta w_{ji}(n) \end{pmatrix} = \begin{pmatrix} learning \\ rate \\ \eta \end{pmatrix} \times \begin{pmatrix} local \\ gradient \\ \delta_j(n) \end{pmatrix} \times \begin{pmatrix} input \\ signal \\ x_i \ or \ z_i \end{pmatrix}$$

(i) If neuron *j* is an output layer neuron, the local gradient is computed using Eqn (8.22) or (8.26), depending on the activation function.

(ii) If neuron *j* is a hidden layer neuron, the local gradient is computed using Eqn (8.23) or (8.27), depending on the activation function.

Learn rate parameter η

The smaller we make the learning rate parameter η, the smaller the changes to synaptic weights will be from one iteration to the next. If we make the learning rate parameter too large, the resulting large changes in the synaptic weights might make the network learning become unstable. A simple method of increasing learning speed, yet avoiding the danger of instability, is to modify the delta rule by including a momentum term:

$$\Delta w_{ji}(n) = \alpha \Delta w_{ji}(n-1) + \eta \delta_j(n) y_i(n)$$

Where α is usually a positive number called the momentum constant. For the convergence of the weight estimation, the momentum constant must be restricted to the range:

$$\mathbf{O} \leq \alpha < \mathbf{1}$$

Modes of training

In a practical application of the back-propagation algorithm, learning results from the many presentations of a prescribed set of training samples to the MLP. One complete presentation of the entire training set during the learning process is called an *epoch*. The learning process is maintained on an epoch-by-epoch basis, until the synaptic weights of the network stabilize. For a given set of training samples, back-propagation learning may proceed in one of the following three basic ways.

(i) Sequential mode

The sequential mode of the back-propagation learning is also referred to as on-line learning. In this mode of operation, weight updating is performed after the presentation of each training sample, one sample at a time.

(ii) Batch mode

In the batch model, the weight updating is performed when all the samples in the epoch are presented to the network. Details are not described here, since the sequential mode is the commonly used method in MLP neural network training.

(iii) Mini-batch mode

Mini-batch splits the training samples into small batches that are used to calculate model error and update model coefficients.

Two passes of the computation

In the application of the back-propagation algorithm for MLP neural network training, there are two distinct passes: the *forward pass* and the *backward pass*.

(1) Forward pass

In the forward pass, the synaptic weights remain unchanged throughout the network, and the signals of the network are computed on a neuron-by-neuron, layer-by-layer basis.

(2) Backward pass

The backward pass starts from the output layer, passes error signals leftward through the network, layer-by-layer, and recursively computes the local gradient for each neuron and updates the weights accordingly.

Back-propagation (BP) algorithm (sequential mode)

(1) Initialization.

Pick the synaptic weights from a random (uniform or normal) distribution.

(2) Presentation of training samples

Present the network with an epoch of training samples. For each sample, perform the sequence of forward and backward computations.

(2.1) Forward computation

During forward computation, the weights are fixed. Compute the activation signal and output of each neuron by proceeding through the network, layer by layer.

(2.2) Backward computation

Compute the local gradient of the neurons and adjust the synaptic weights of network. Repeat (2.1)-(2.2) until all samples in the current epoch are presented once.

(3) Iteration

Repeat Step (2) by presenting new epochs of training samples to the network until the following stopping criterion is met.

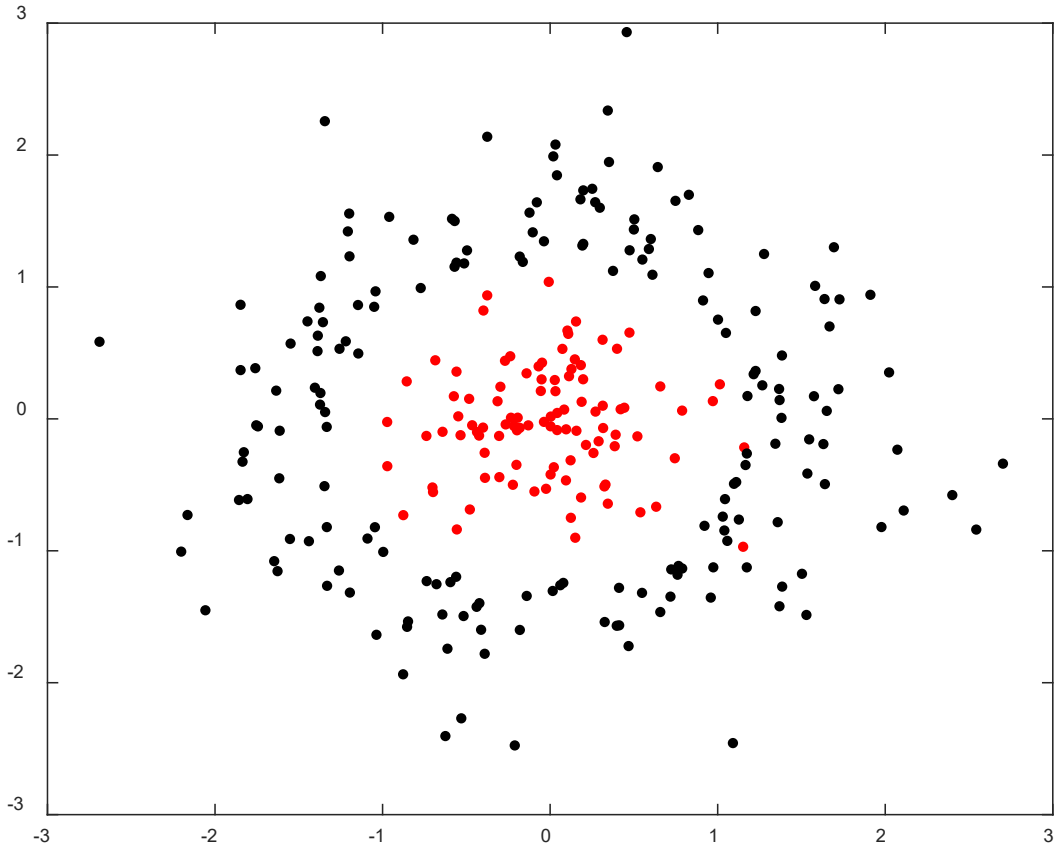
(i) the number of iterations, or

(ii) the change of the average error is small enough.

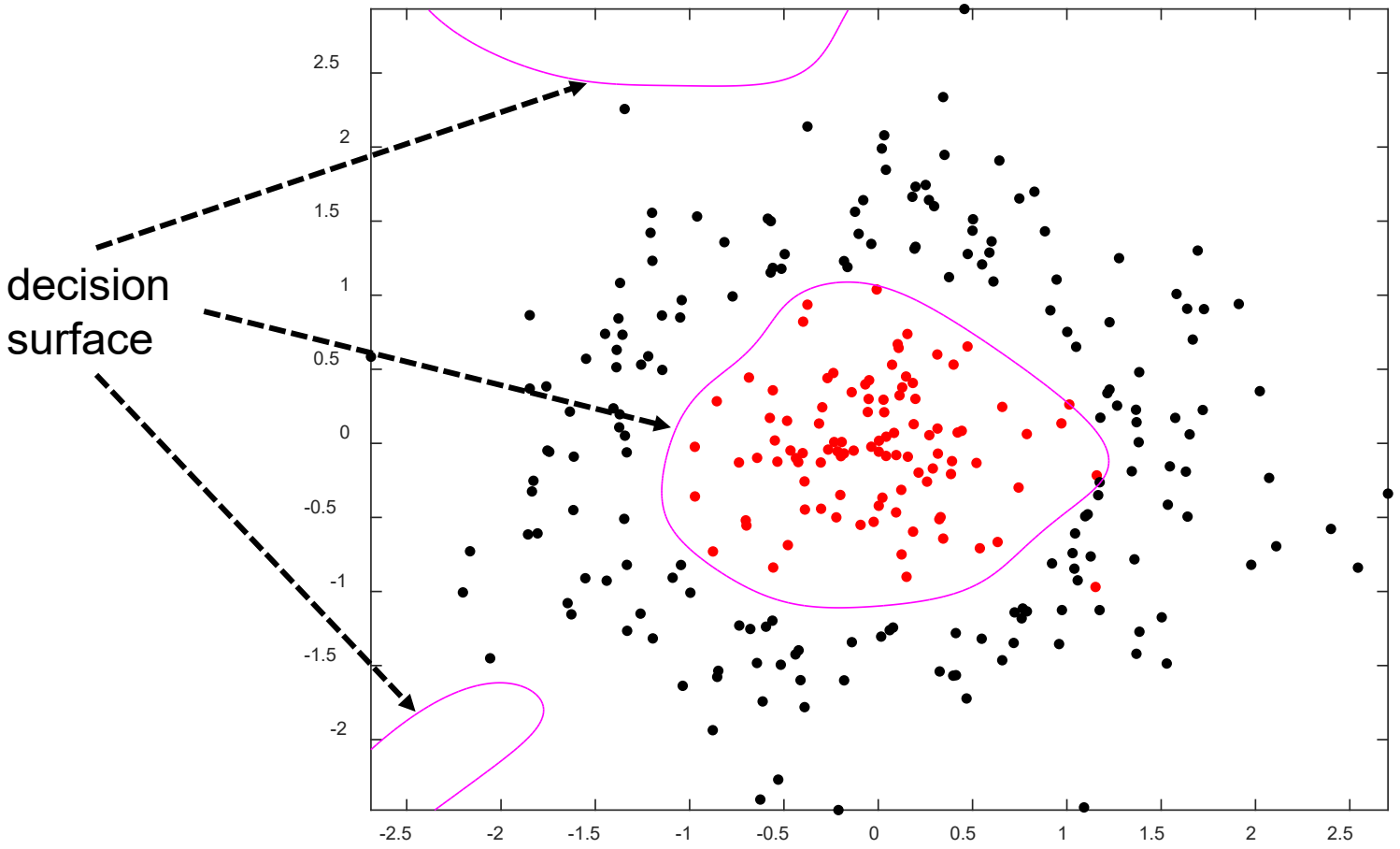
Note, the order of presentation of training samples should be randomized from epoch to epoch.

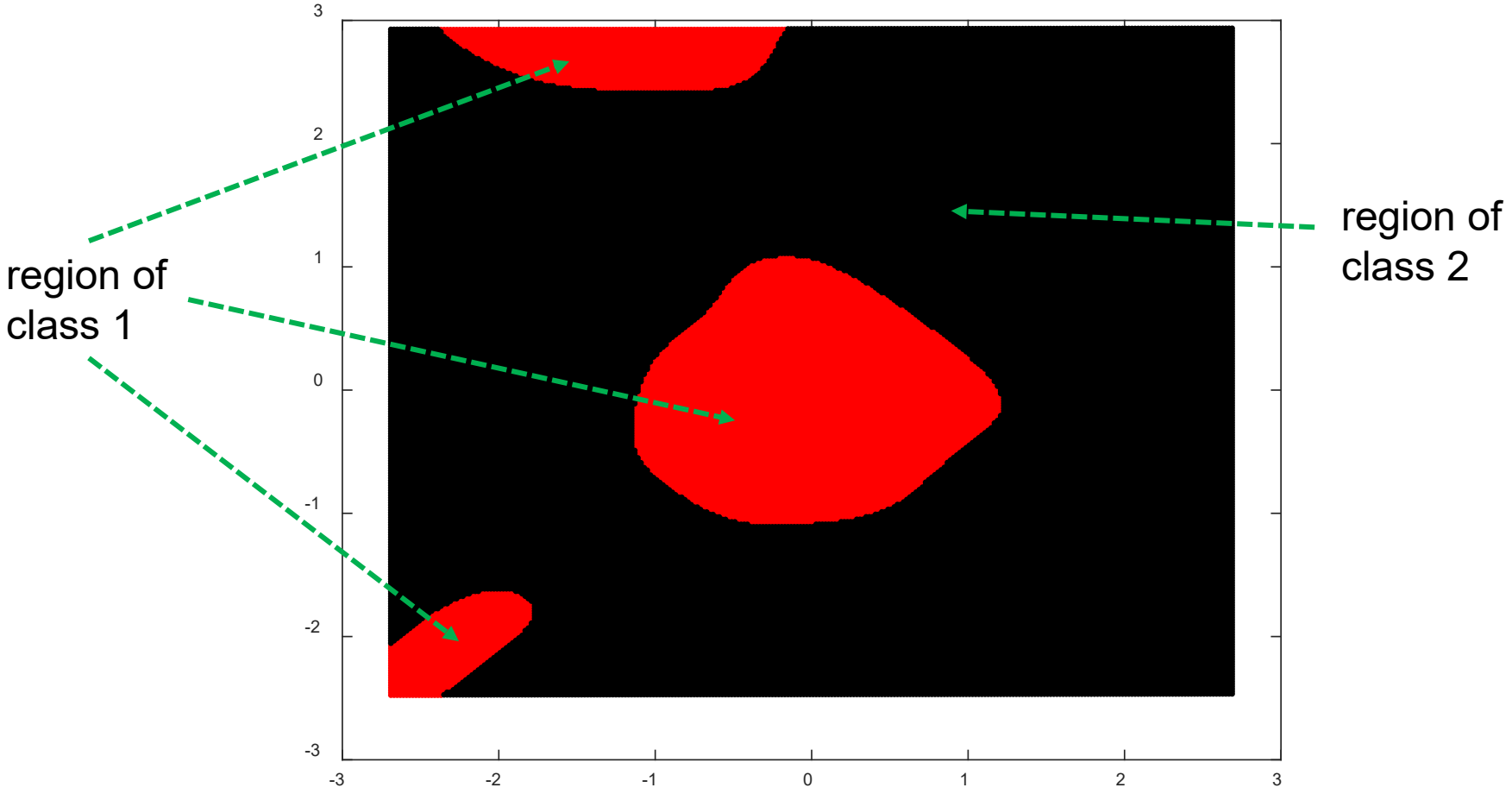
8.4 Discussions on MLP neural networks

Observation 1: The neural network learned is affected by the initial values of the weights

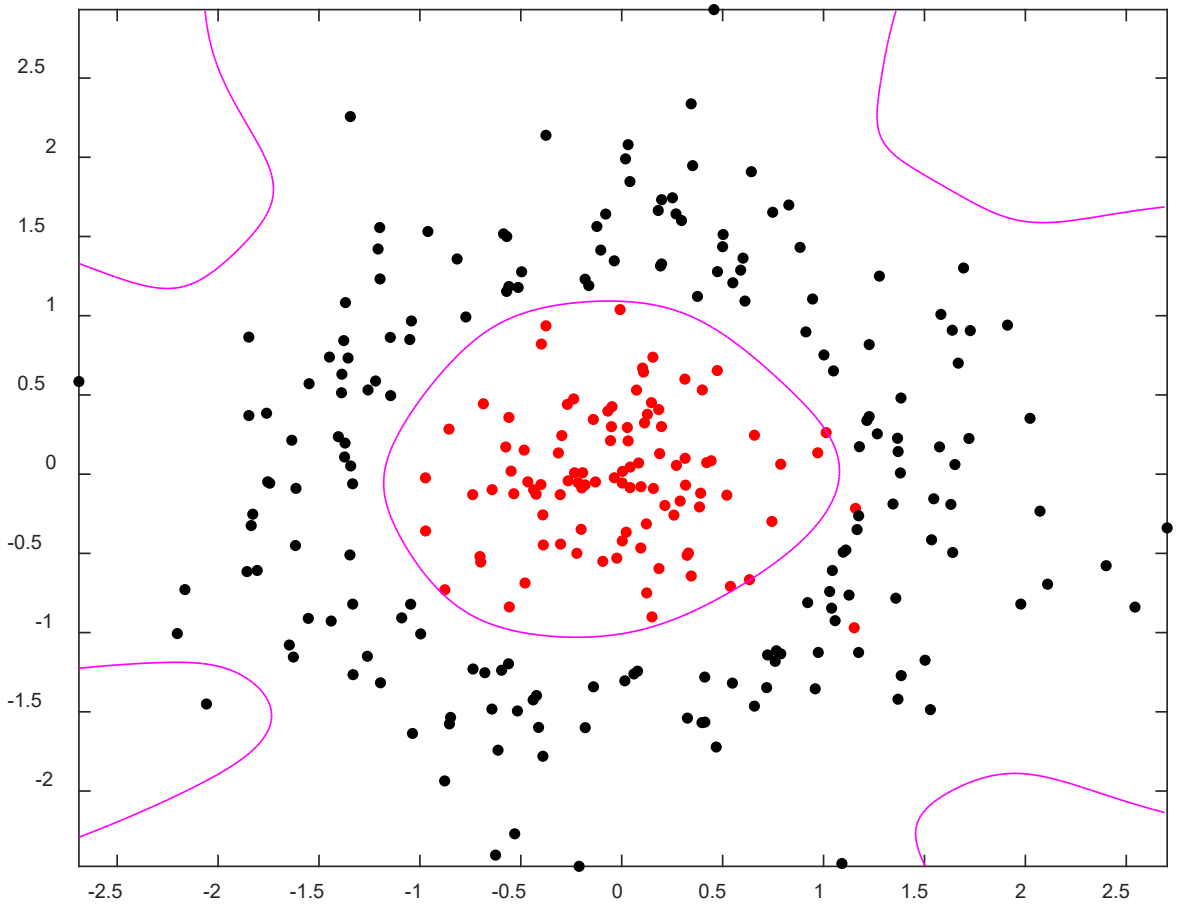


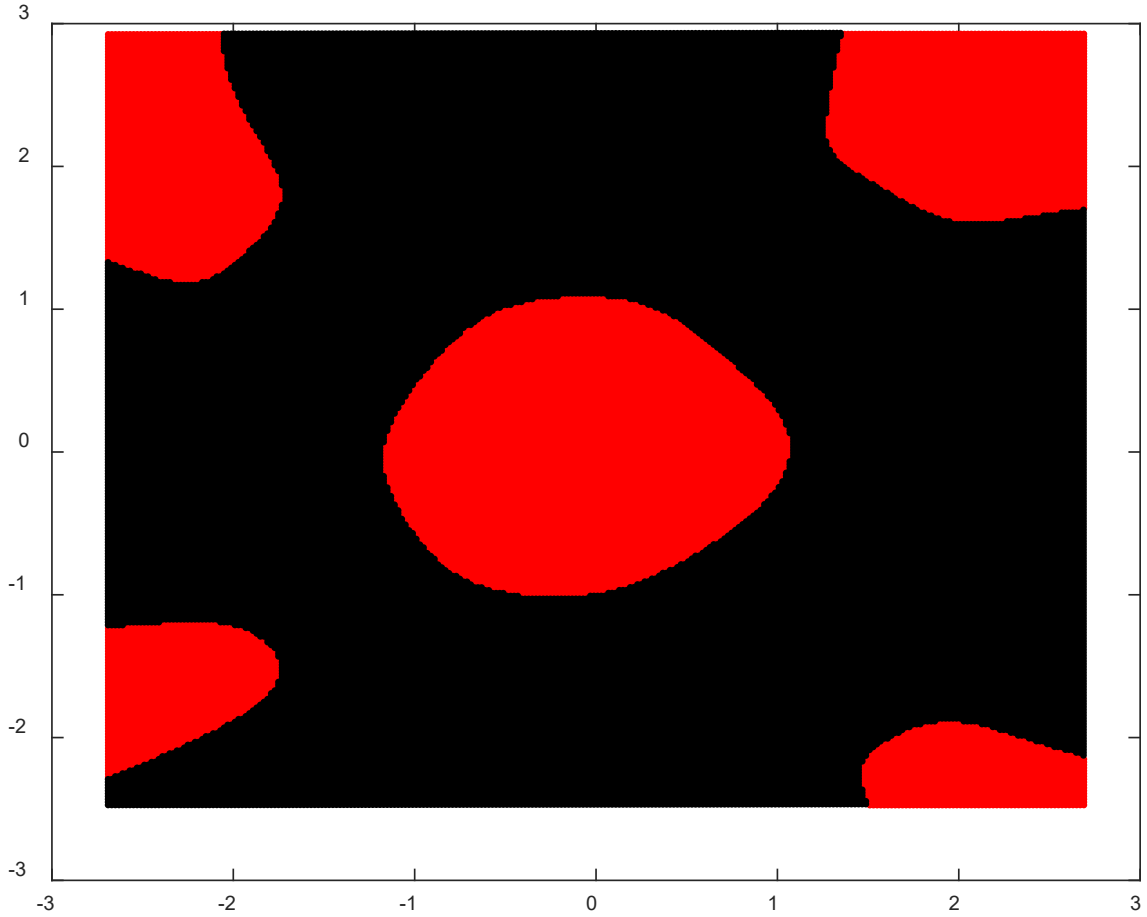
1st run: one hidden layer with 20 neurons



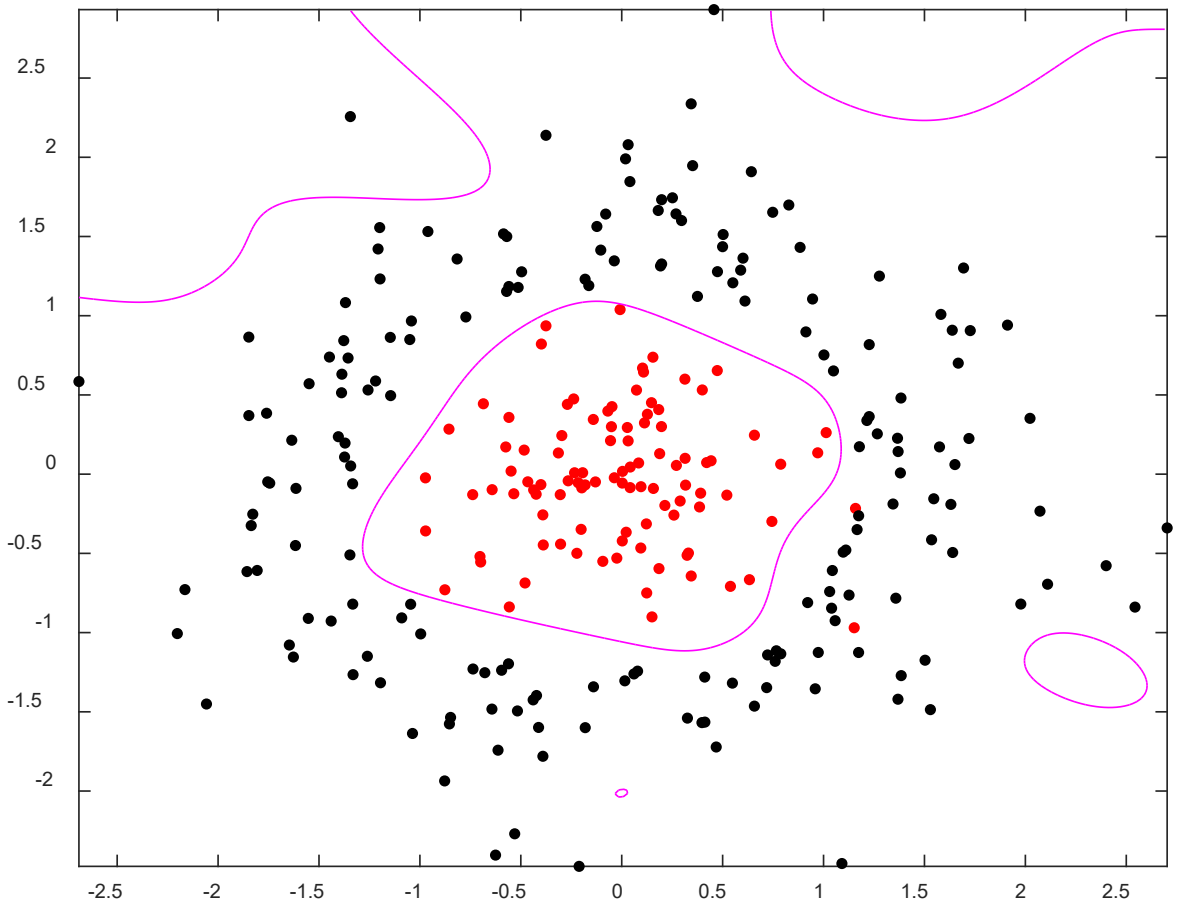

region of
class 1
region of
class 2
3
2
1
0
-1
-2
-3
-3
-2
-1
0
1
2
3

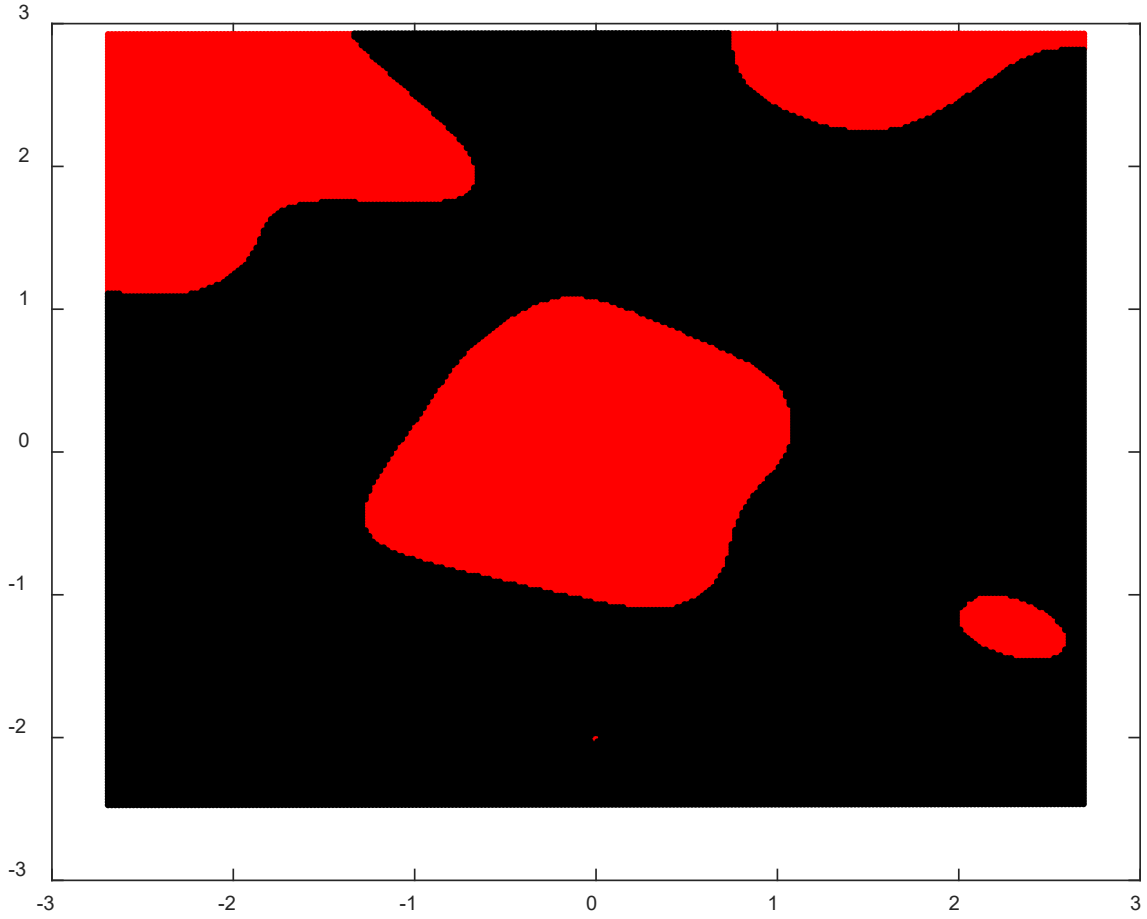
2nd run: one hidden layer with 20 neurons



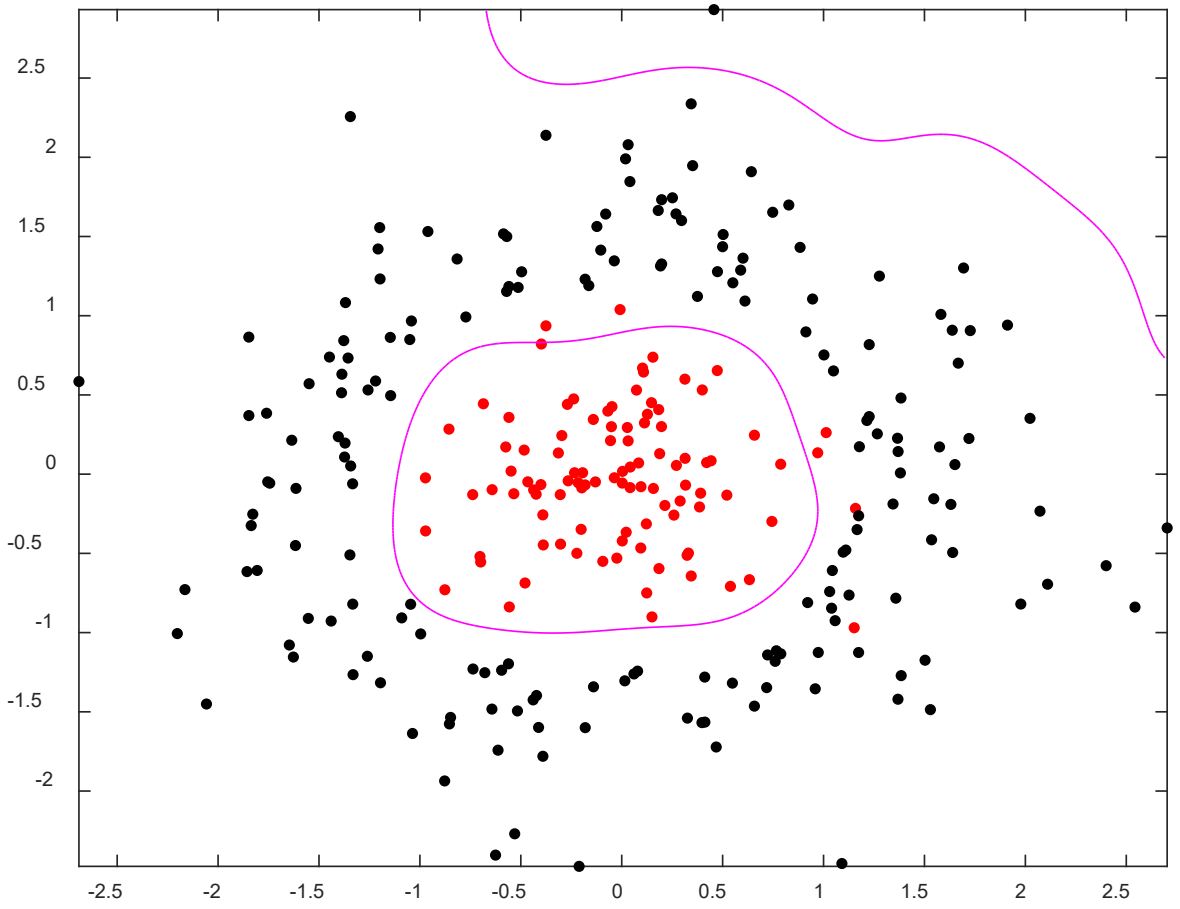

3
2
1
0
-1
-2
-3
-3
-2
-1
0
1
2
3

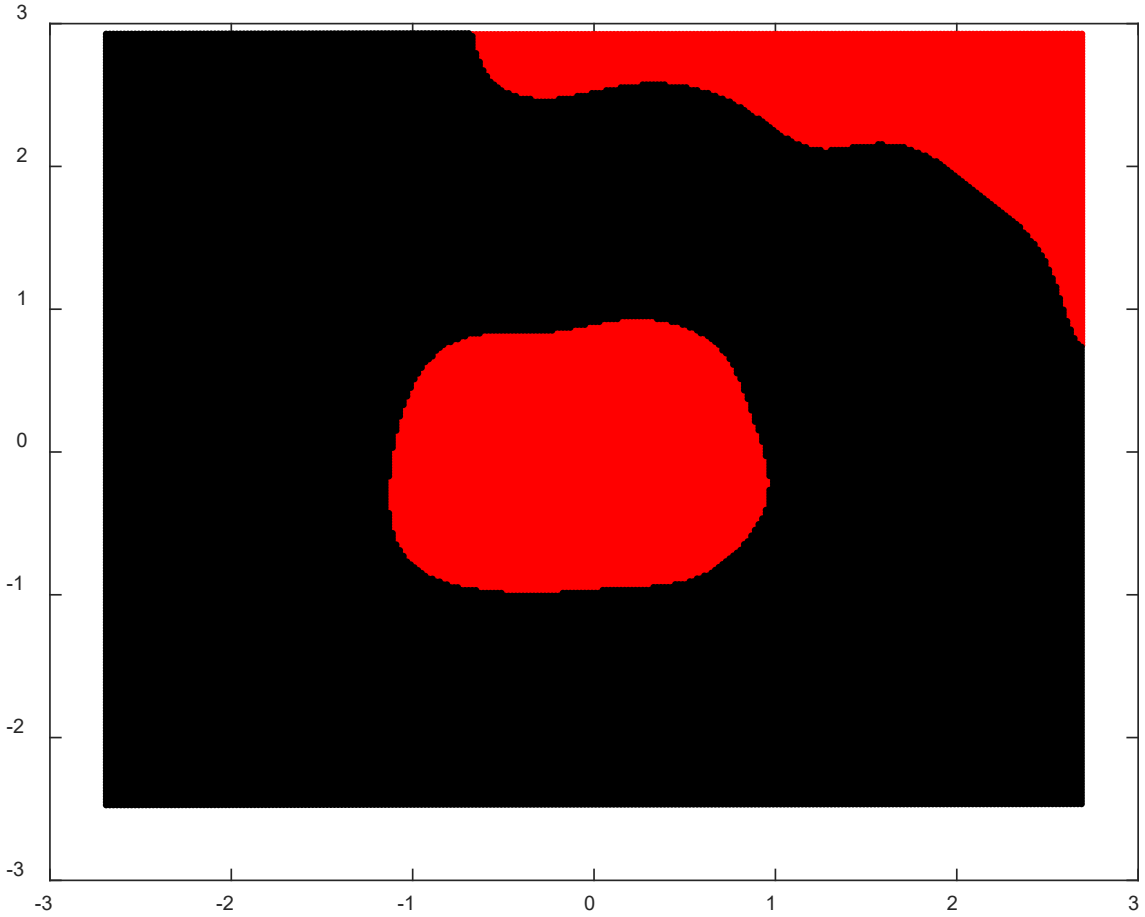
3rd run: one hidden layer with 20 neurons




3
2
1
0
-1
-2
-3
-3
-2
-1
0
1
2
3

4th run: one hidden layer with 20 neurons




3
2
1
0
-1
-2
-3
-3
-2
-1
0
1
2
3

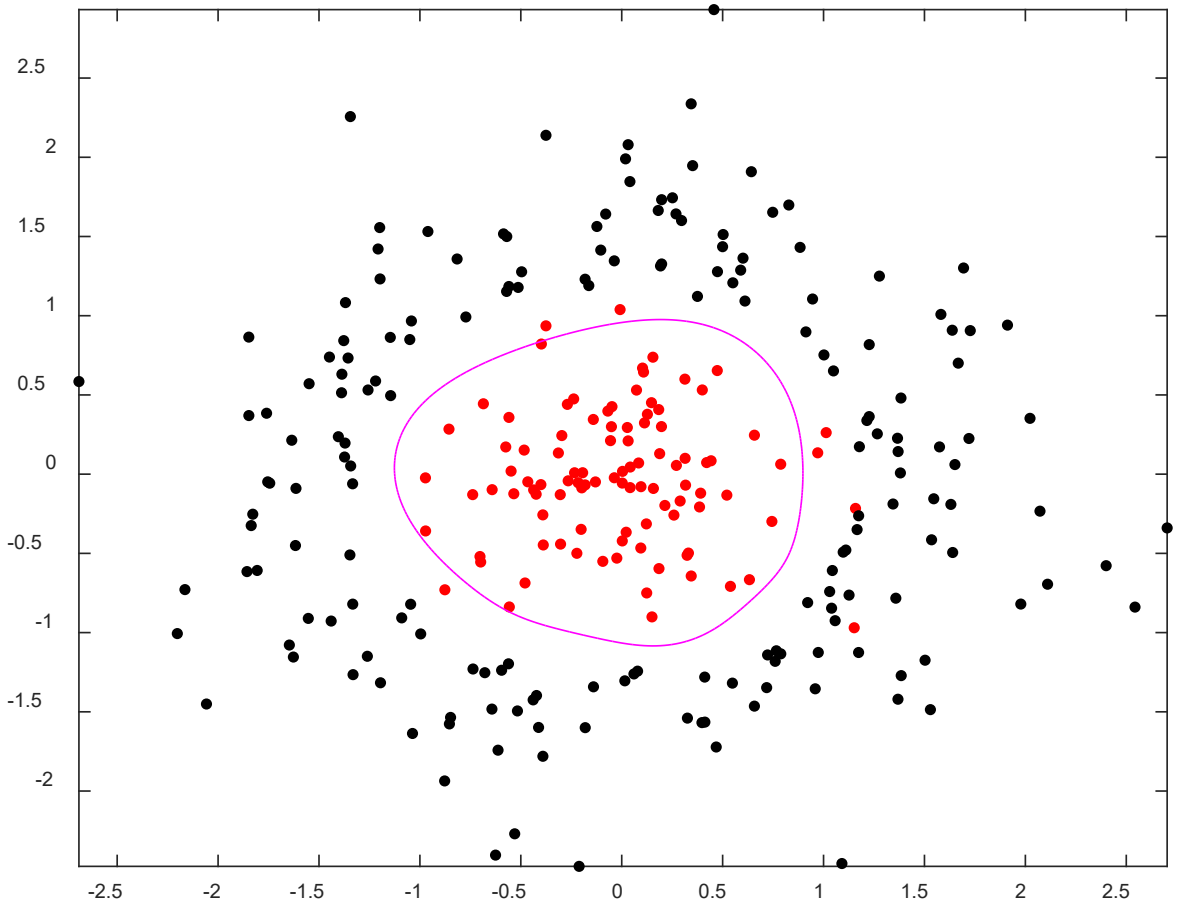
Discussion 1:

Why 4 different runs give different results?

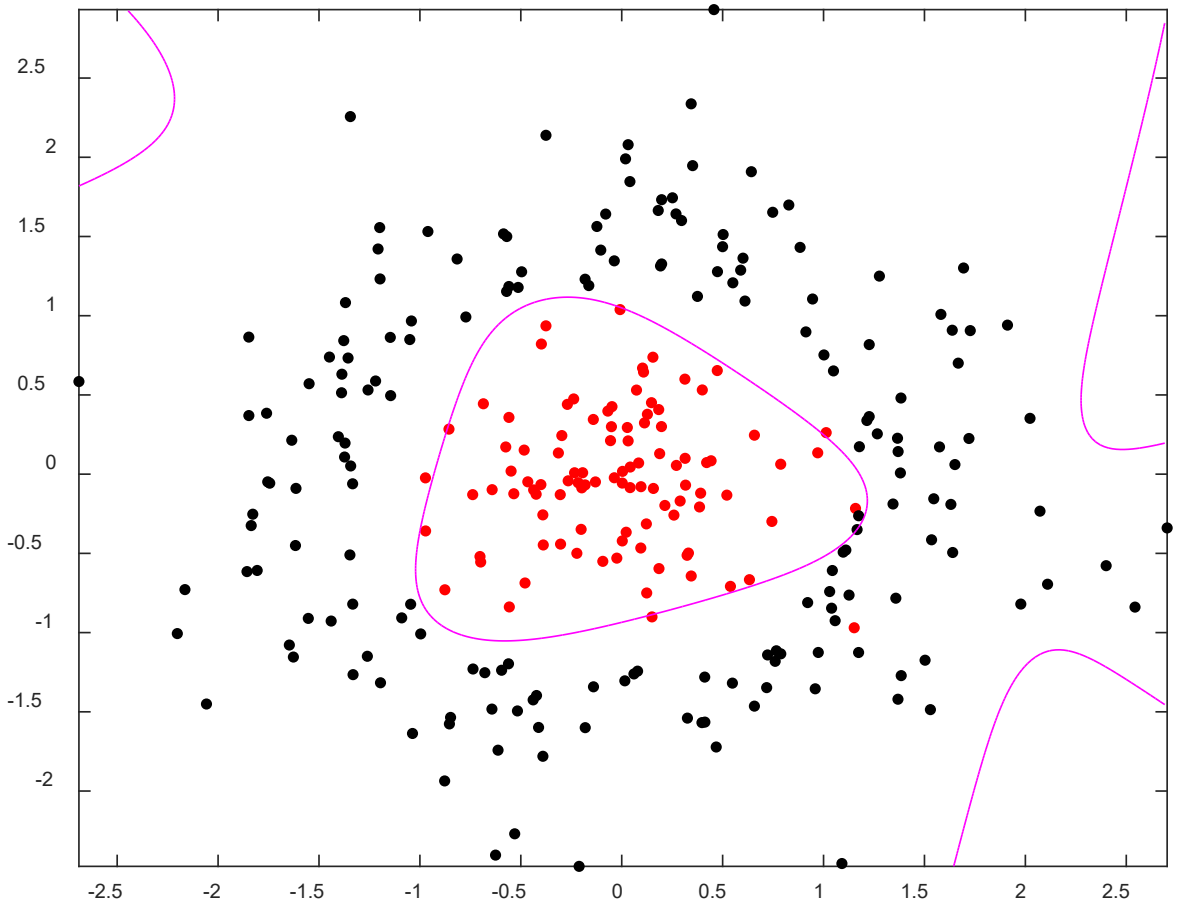
Observation 2: The more hidden layer neurons are used, the more complex the decision surface could be.

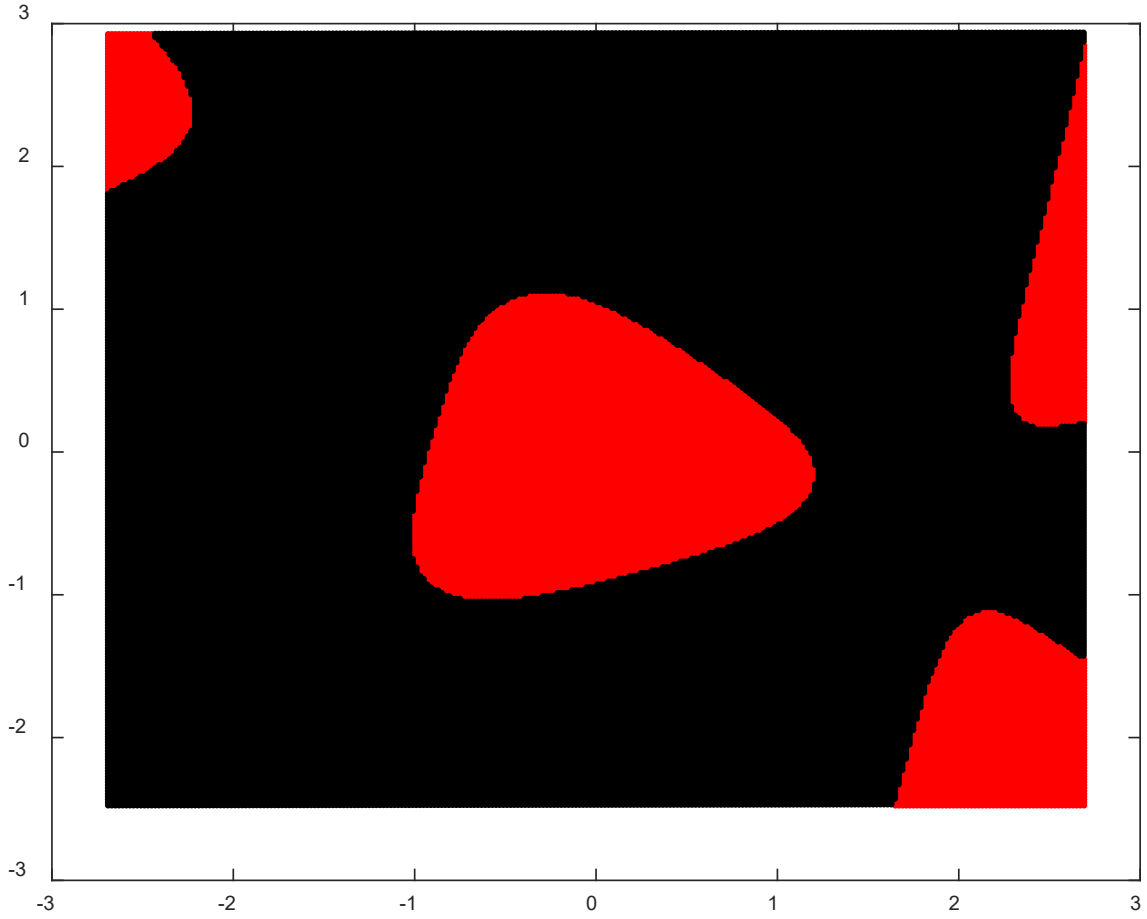
Next, we set the number of neurons to 5 and 200, respectively, and repeat the experiments.

1st run: one hidden layer with 5 neurons

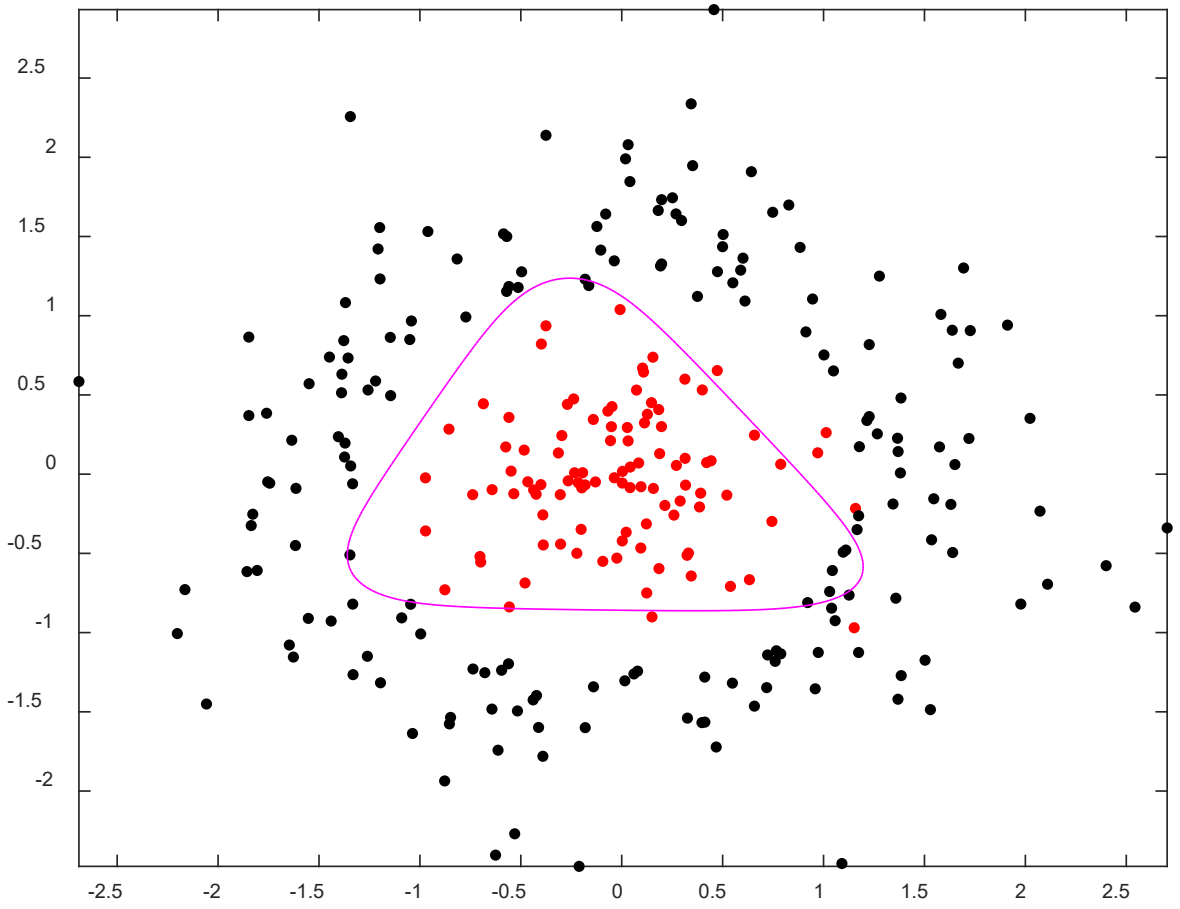


2nd run: one hidden layer with 5 neurons

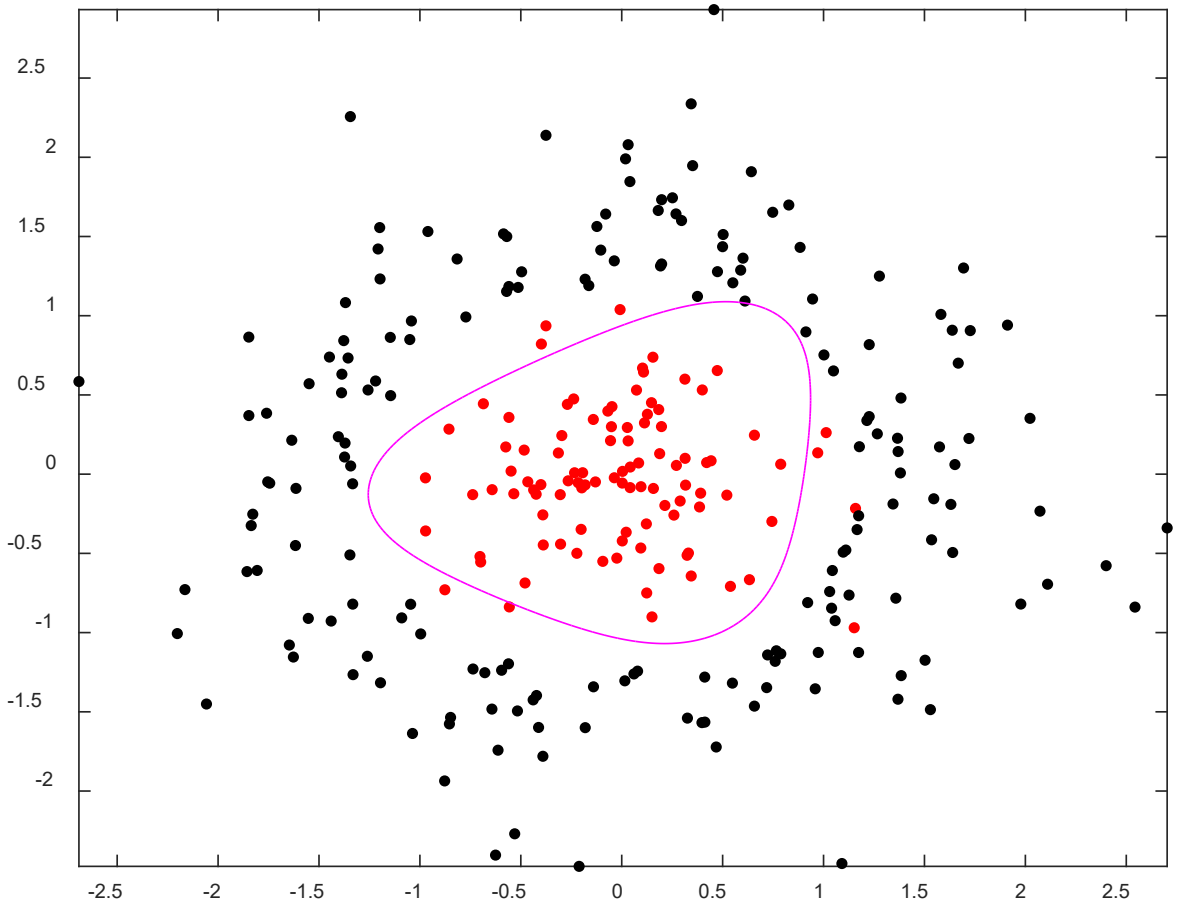



3
2
1
0
-1
-2
-3
-3
-2
-1
0
1
2
3

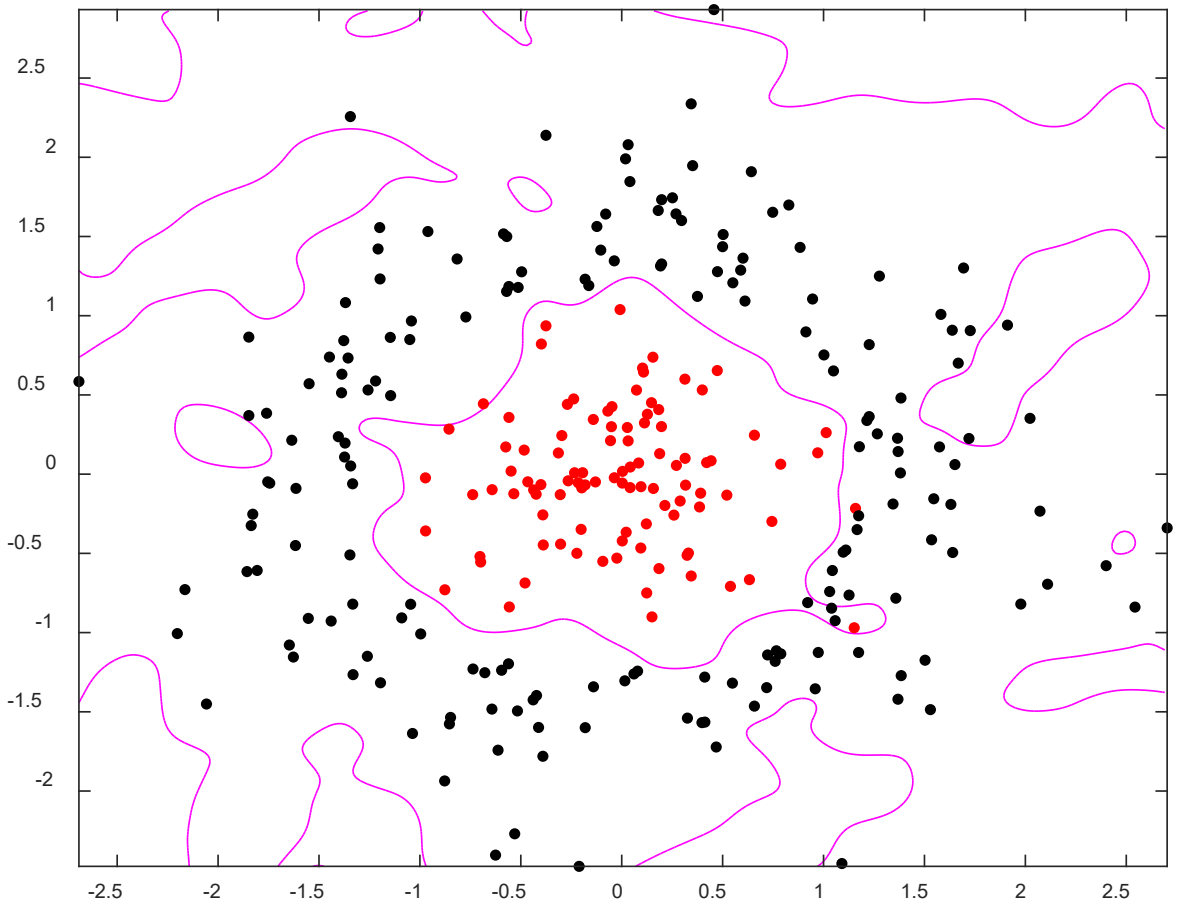
3st run: one hidden layer with 5 neurons

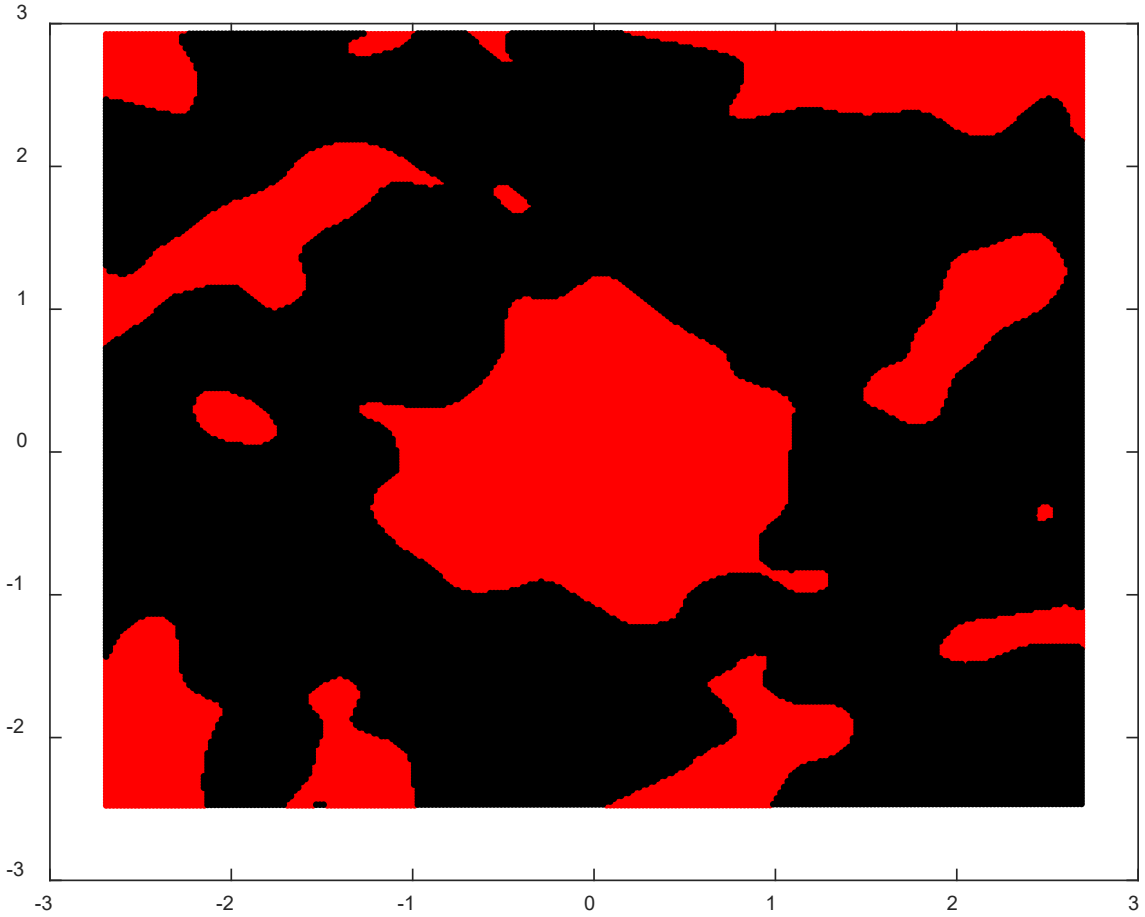


4st run: one hidden layer with 5 neurons

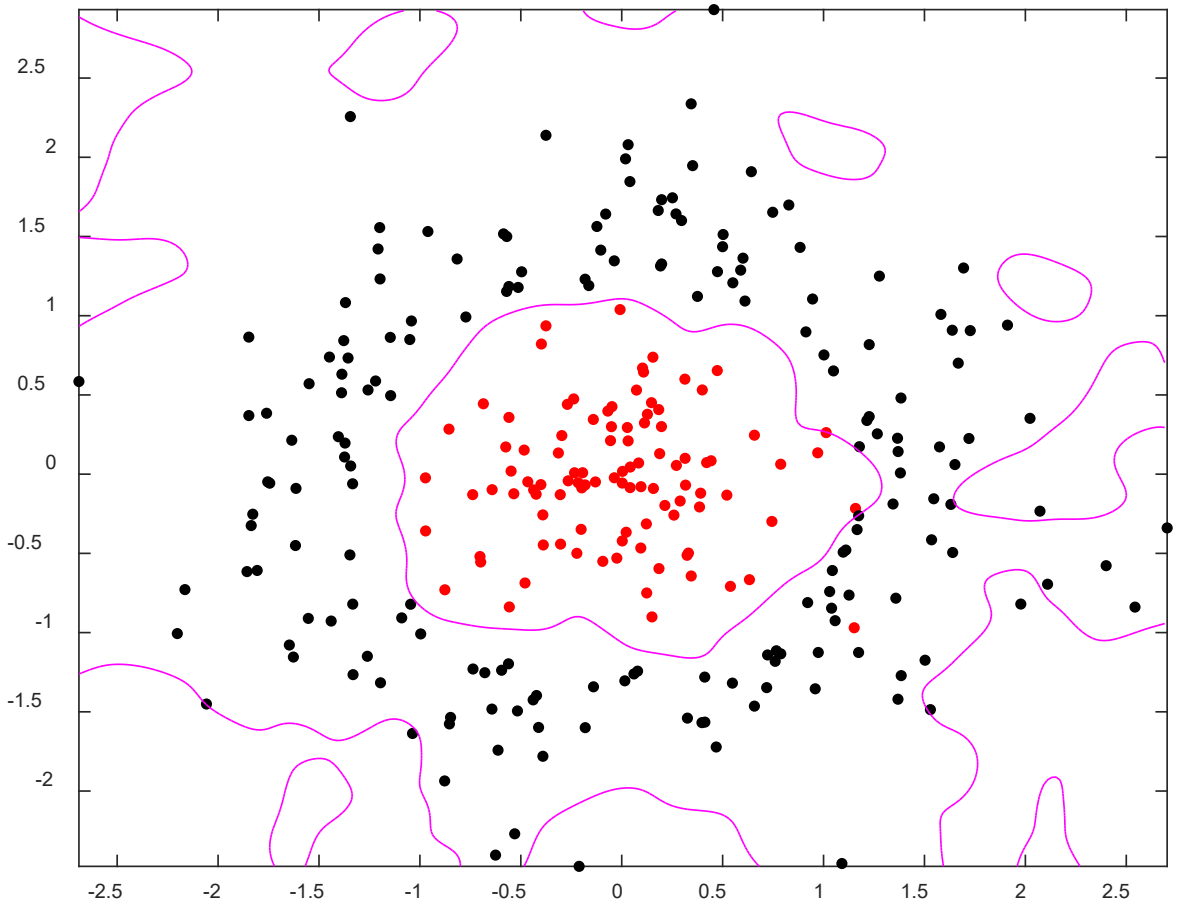


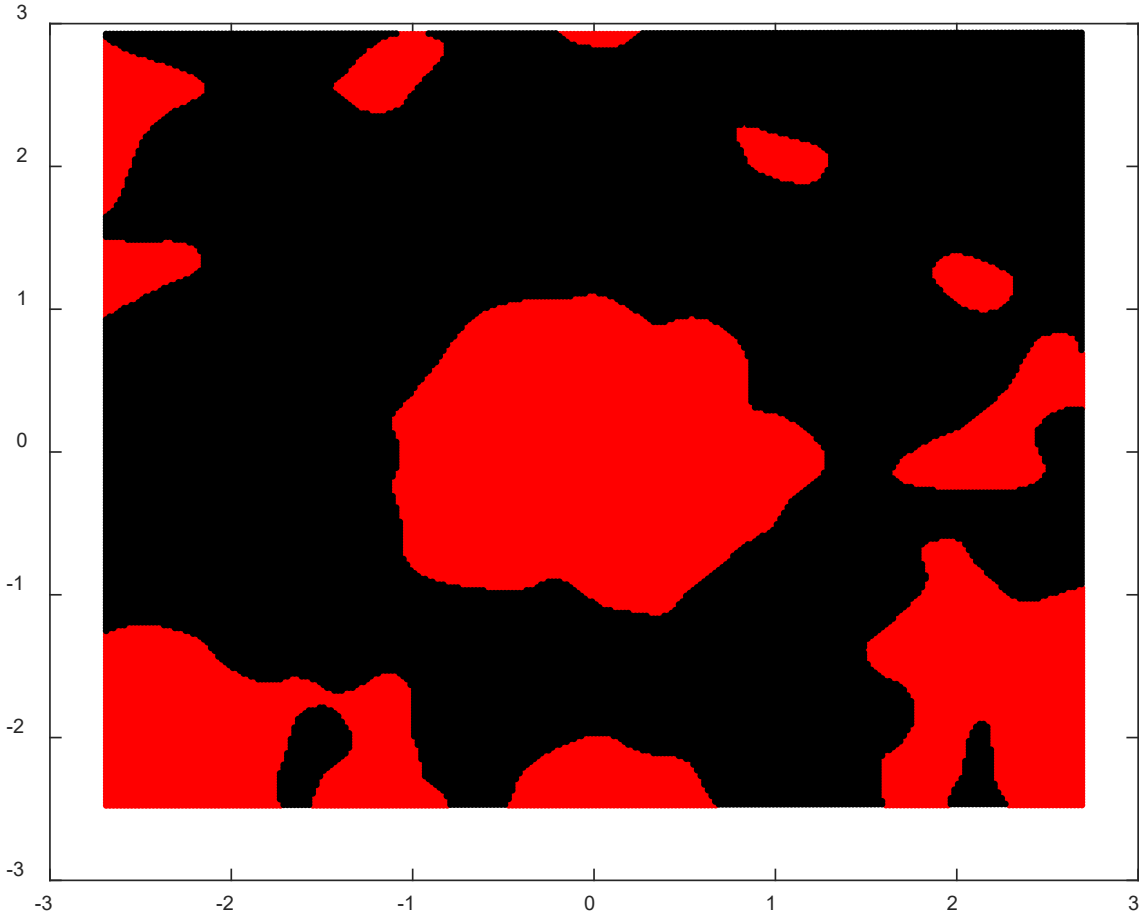
1st run: one hidden layer with 200 neurons




3
2
1
0
-1
-2
-3
-3
-2
-1
0
1
2
3

2nd run: one hidden layer with 200 neurons



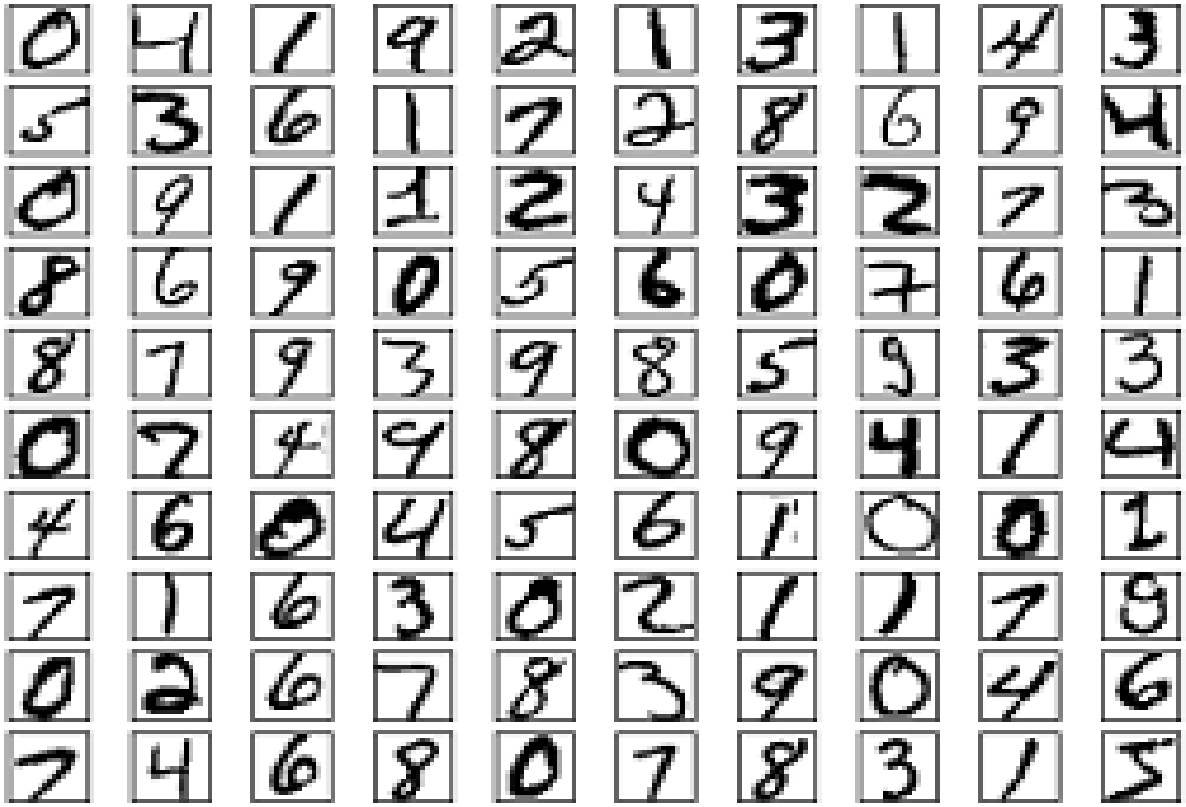

3
2
1
0
-1
-2
-3
-3
-2
-1
0
1
2
3

Discussion 2

What lessons do you learn from these results?

Observation 3: The gradient vanishing problem of MLP if more than one hidden layers are used.

We show this problem using the example of the MLP neural network for recognition of handwritten digits:



Each digit is an image. If the image is with 28*28 pixels, then each image can be represented by a 784-dimensional vector.

Next, we design the MLP neural network architecture, including

(i) The number of neurons in the input layer,

(ii) The number of hidden layers and number of neurons in each layer

(iii) The number of neurons in the output layer

The *number of neurons in the input layer* should be the same as the dimension of input vector. Therefore 784 neurons are used in the input layer.

The *number of neurons in the output layer* is usually set to the number of classes. There are 10 classes corresponding to the 10 digits. Therefore 10 neurons are used in the output layer.

Assuming that 30 neurons are used in the hidden layer. Next, we will train three MLP neural networks with one, two, three and four hidden layers using MNIST data.

MNIST data comes in two parts. The first part contains 50,000 training images. These images are scanned handwriting samples from 250 people including US Census Bureau employees and high school students. The second part of the MNIST data set is 10,000 testing images from a different set of 250 people.

Each training image has an associated label (e.g. 6 for image digit 6). This label can be coded into 10-dimensional vector using one-hot encoding. For example, label 1 and label 6 can be coded as:

$$D = \begin{bmatrix} 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \end{bmatrix}^T$$

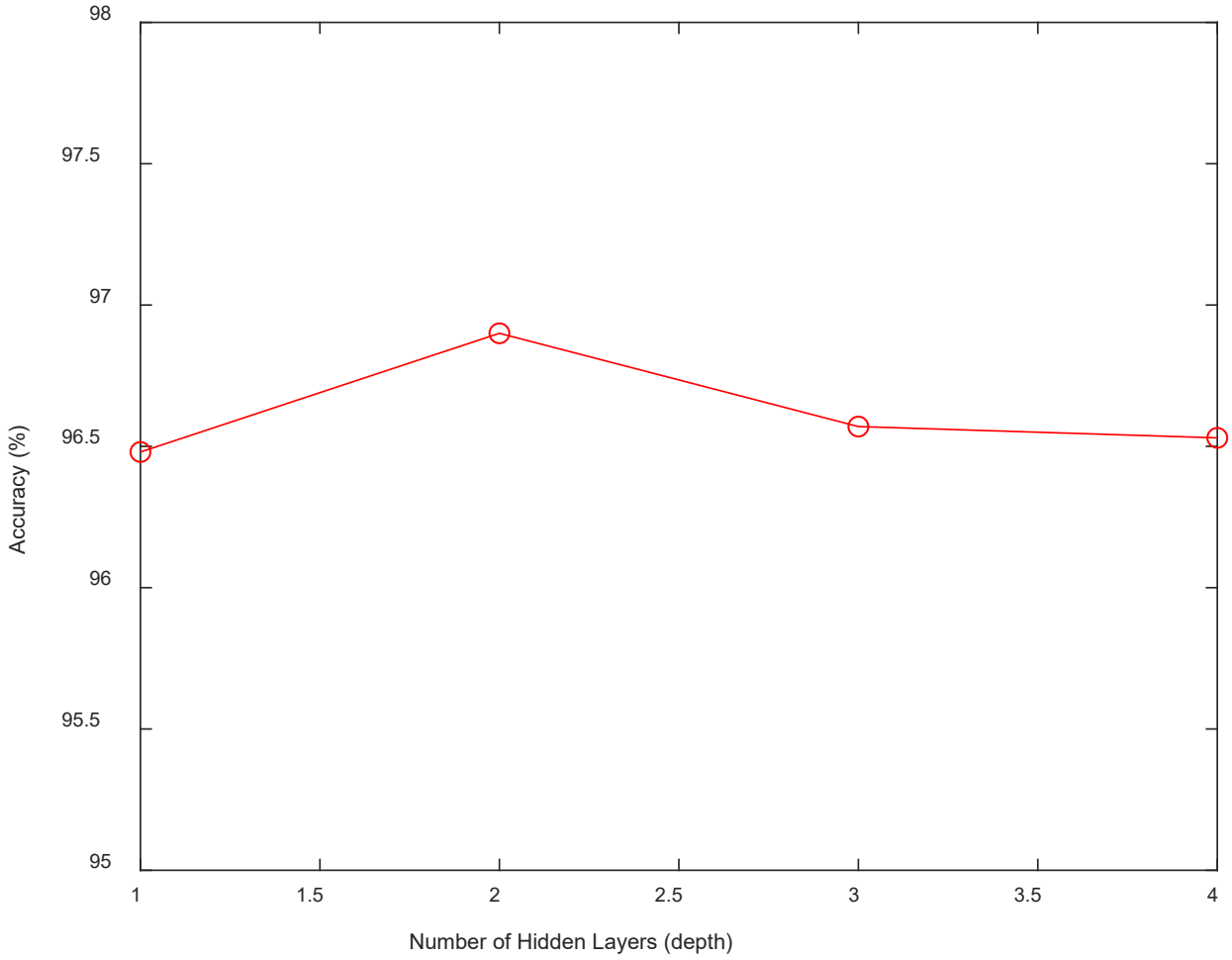
$$D = \begin{bmatrix} 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \end{bmatrix}^T$$

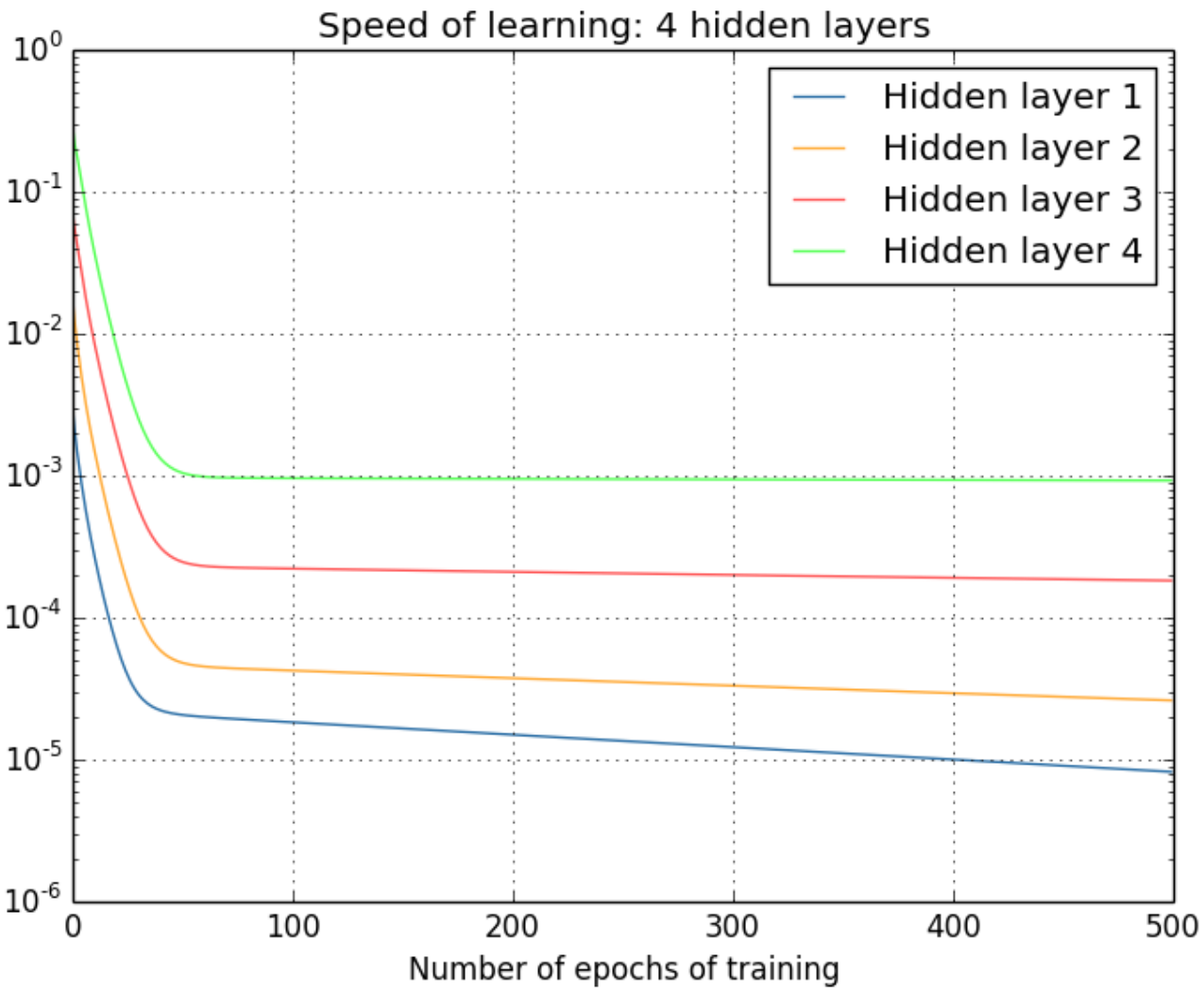
When one hidden layer is used, the performance on the test data is 96.4%.

Now, let's add another hidden layer, also with 30 neurons, and retrain the neural network with a structure of [784 30 30 10], which gives an improved classification accuracy, 96.90%. This result is encouraging: a little more depth is helping.

Let's add another 30-neuron hidden layer. Now the network has 3 hidden layers. But the result drops back down to 96.57 percent, close to our original shallow network.

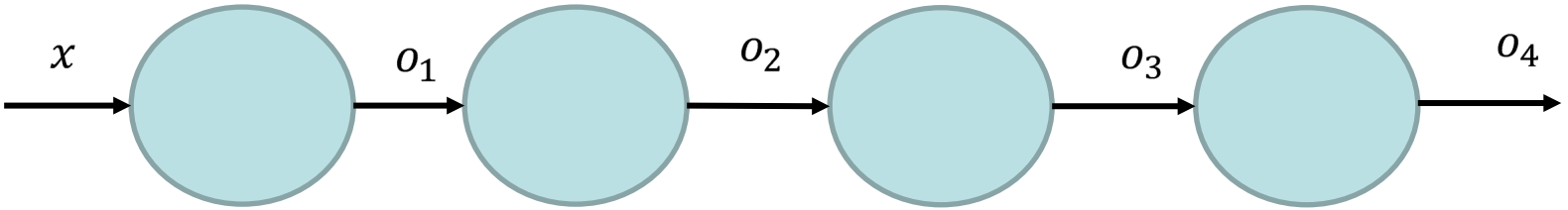
Let's add another 30-neuron hidden layer. Now the network has 4 hidden layers. The result obtained is 96.53 percent. This is not a statistically significant drop, but it is not encouraging, either. The results of MLP neural networks with different number of hidden layer are summarized below:


98
97.5
97
96.5
96
95.5
95
Accuracy (%)
1
1.5
2
2.5
3
3.5
4
Number of Hidden Layers (depth)


Speed of learning: 4 hidden layers
Hidden layer 1
Hidden layer 2
Hidden layer 3
Hidden layer 4
10^{0}
10^{-1}
10^{-2}
10^{-3}
10^{-4}
10^{-5}
10^{-6}
0
100
200
300
400
500
Number of epochs of training

We have here an important observation: the gradient tends to get smaller as we move backward through the hidden layers. This means that neurons in the earlier layers learn much more slowly than neurons in later layers. And while we've seen this in just a single example, there are fundamental reasons why this happens in MLP neural networks. The phenomenon is known as the *vanishing gradient problem.*

To get insight into why the vanishing gradient problem occurs, let's consider the simple neural network with just a single neuron in each layer. Here's a network with three hidden layers:



where $$o_j = \varphi(v_j) = \frac{1}{1+\exp(-v_j)}$$

$$v_j = w_j o_{j-1} + b_j$$

Assuming that error function is given by:

$$J = \frac{1}{2}[o_4 - d]^2$$

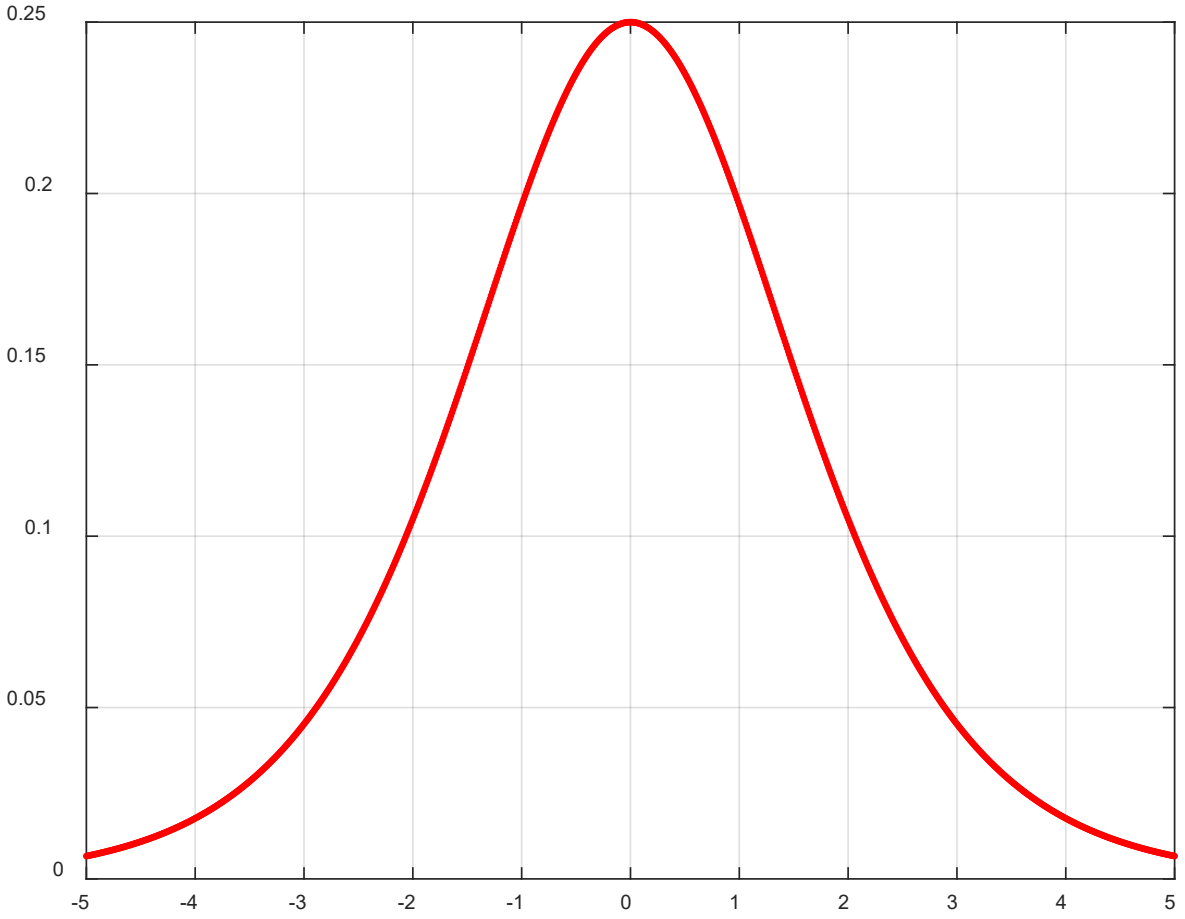
The gradient of J with respect to weight in the first hidden layer is given by:

$$\frac{\partial J}{\partial w_1} = \frac{\partial J}{\partial o_4} \times \frac{\partial o_4}{\partial o_3} \times \frac{\partial o_3}{\partial o_2} \times \frac{\partial o_2}{\partial o_1} \times \frac{\partial o_1}{\partial w_1}$$

$$= \frac{\partial J}{\partial o_4} \times \frac{\partial o_4}{\partial v_4} \times \frac{\partial v_4}{\partial o_3} \times \frac{\partial o_3}{\partial v_3} \times \frac{\partial v_3}{\partial o_2} \times \frac{\partial o_2}{\partial v_2} \times \frac{\partial v_2}{\partial o_1} \times \frac{\partial o_1}{\partial v_1} \times \frac{\partial v_1}{\partial w_1}$$

$$= [d - o_4] \times \varphi'(v_4) \times w_4 \times \varphi'(v_3) \times w_3 \times \varphi'(v_2) \times w_2 \times \varphi'(v_1) \times x$$

$$\varphi'(v_j) = \frac{\exp(-v_j)}{[1+\exp(-v_j)]^2}$$



Obviously,

$$\varphi'(v_j) \leq \frac{1}{4}$$

In addition, if w_j is from a normal distribution with zero mean, and unit standard deviation, most likely we have:

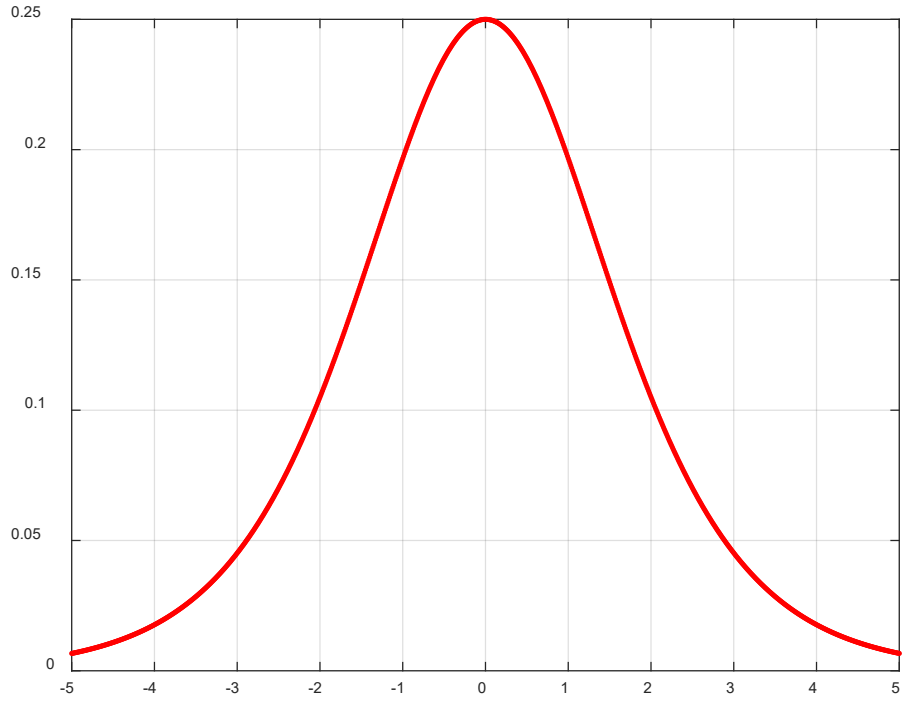
$$|w_j| < 1$$

Hence:

$$\left|\frac{\partial J}{\partial w_1}\right| < |d - o_4| \times \frac{1}{4} \times \frac{1}{4} \times \frac{1}{4} \times \frac{1}{4} |x|$$

As can be seen, the gradient of the cost function J to w_{j-1} will typically be one quarter of the gradient to w_j, which means the learning speed at the earlier layers will be slower than later layers. This is the essential origin of the vanishing gradient problem.

Of course, this is an informal argument, not a rigorous proof. There are several possible escape clauses. In particular, we might wonder whether the weights will grow during training. Indeed, $|w_j|$ can be greater than 1, but this will increase the activation signal v_j, and eventually $|\varphi'(v_j)w_j|$ may not be necessarily increased as shown below.



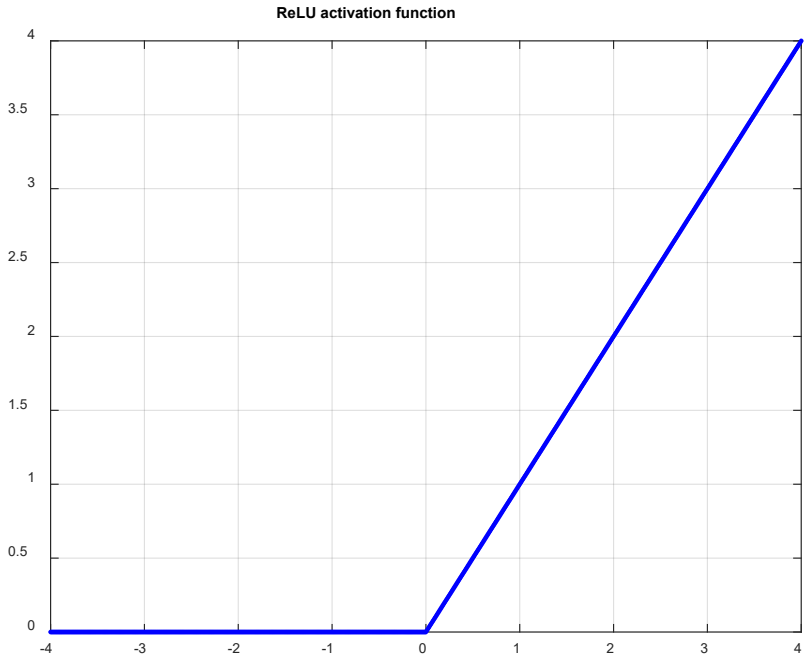
Discussion 3

How to solve the gradient vanishing problem?

To solve the gradient vanishing problem, ReLU activation function and its variants can be used.

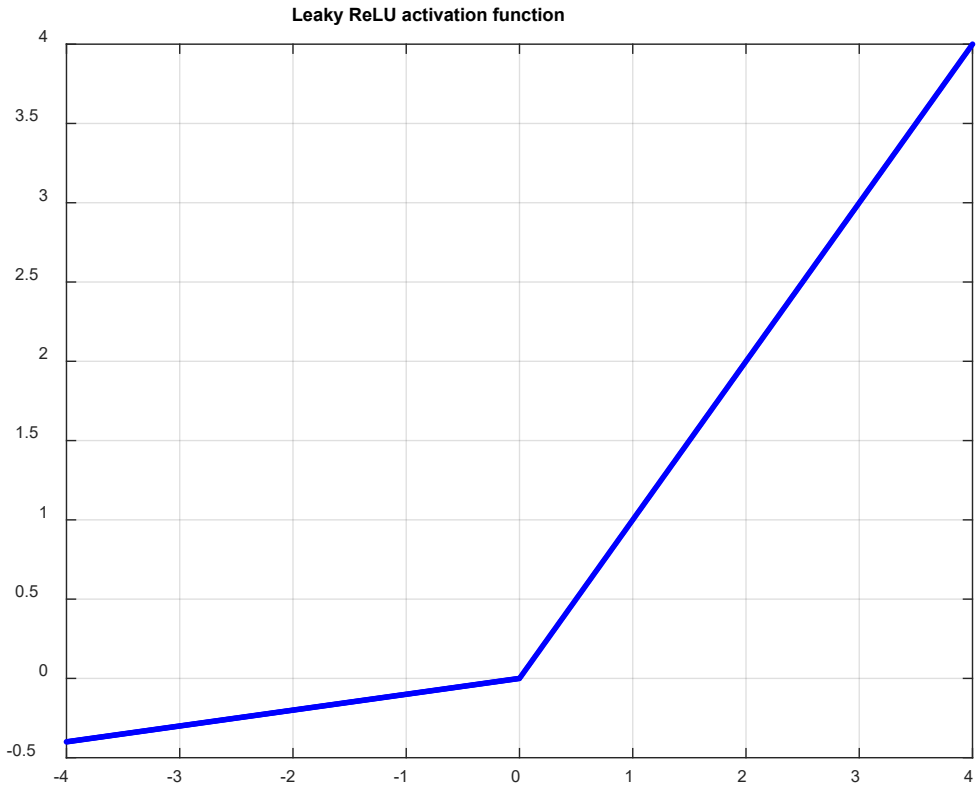
Rectified Linear Unit (ReLU) activation function

$$f(x) = \begin{cases} x & x \geq 0 \\ 0 & x < 0 \end{cases}$$



Leaky/Parametric ReLU

$$f(x) = \begin{cases} x & x \geq 0 \\ ax & x < 0 \end{cases}$$



Exponential Linear Unit (ELU) activation function

$$f(x) = \begin{cases} x & x \geq 0 \\ \alpha(e^x - 1) & x < 0 \end{cases}$$