Week 4 – Multilayer Perceptrons and Backpropagation Algorithm

ELEC0144 Machine Learning for Robotics

Dr. Chow Yin Lai

Email: uceecyl@ucl.ac.uk

Schedule

Week	Lecture	Workshop	Assignment Deadlines
1	Introduction; Image Processing	Image Processing	
2	Camera and Robot Calibration	Camera and Robot Calibration	
3	Introduction to Neural Networks	Camera and Robot Calibration	Friday: Camera and Robot Calibration
4	MLP and Backpropagation	MLP and Backpropagation	
5	CNN and Image Classification	MLP and Backpropagation	
6	Object Detection	MLP and Backpropagation	Friday: MLP and Backpropagation
7	Path Planning	Path Planning	
8	Kalman Filter SLAM	Path Planning	
9	Extended Kalman Filter SLAM	Path Planning	
10	Particle Filter SLAM	Path Planning	Friday: Path Planning

Content

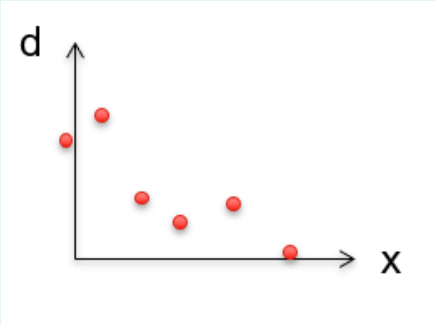
- Revision: Optimization
- Backpropagation
- Generalization
- Discussions
- MATLAB NN Toolbox Example

Content

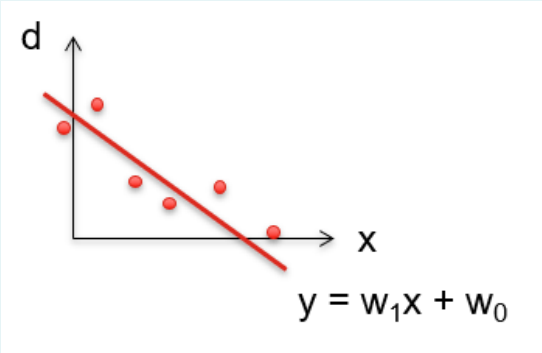
- Revision: Optimization
- Backpropagation
- Generalization
- Discussions
- MATLAB NN Toolbox Example

Revision: Steepest Descent - Batch (1)

- Consider the following data:



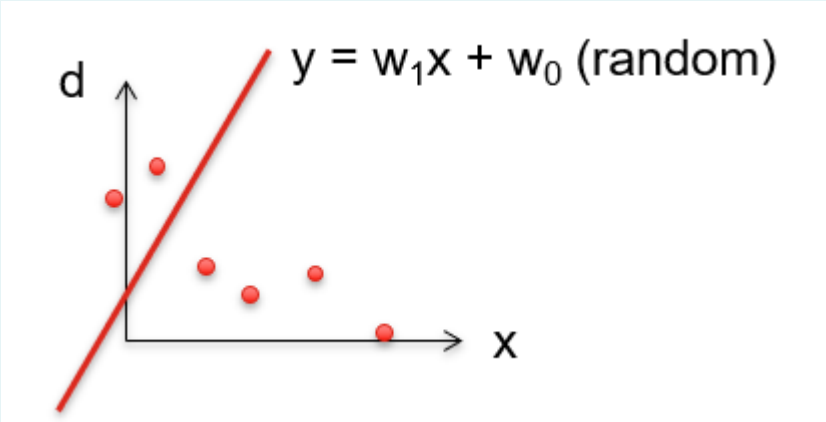
- We want to find a best-fit straight line so that the total error is the least.



- How do we do that?

Revision: Steepest Descent - Batch (2)

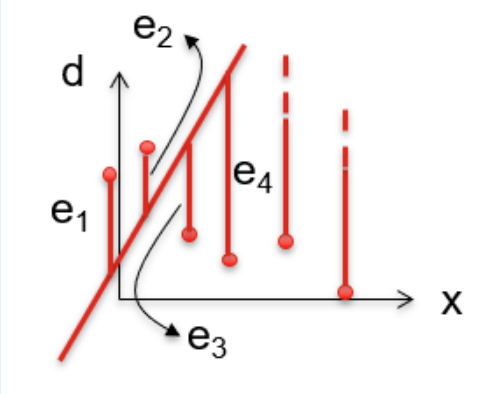
- Firstly, we start with any random w_1 and w_0.
- This will obviously give a very poor fit.



Revision: Steepest Descent - Batch (3)

- We get the modelling error at each data point, then calculate the sum of squared errors (SSE).

$$\text{SSE} = e_1^2 + e_2^2 + \cdots + e_n^2$$



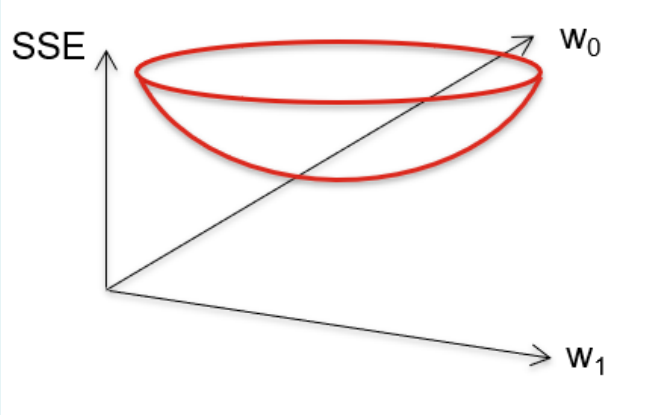
Revision: Steepest Descent - Batch (4)

- The sum of squared errors, in terms of the weights, is:

$$
\begin{aligned}
\text{SSE} &= e_1^2 + e_2^2 + e_3^2 + \cdots + e_n^2 \\
&= (d_1 - y_1)^2 + \cdots + (d_n - y_n)^2 \\
&= (d_1 - w_1 x_1 - w_0)^2 + \cdots + (d_n - w_1 x_n - w_0)^2
\end{aligned}
$$

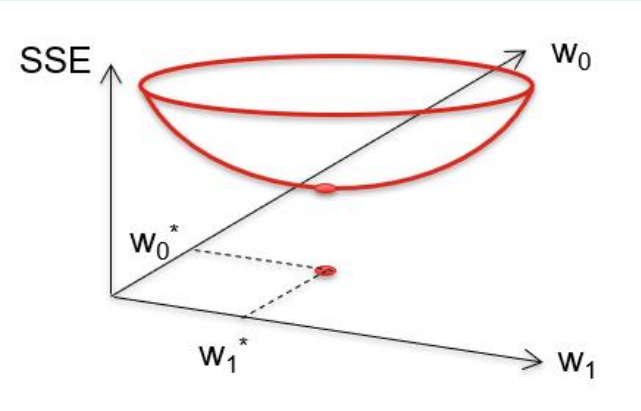
Revision: Steepest Descent - Batch (5)

- If we try out all possible combinations of w_1 and w_0, each time getting the value of SSE, and plot a graph of SSE vs all possible combinations of w_1 and w_0, we may get something like this:



Revision: Steepest Descent - Batch (6)

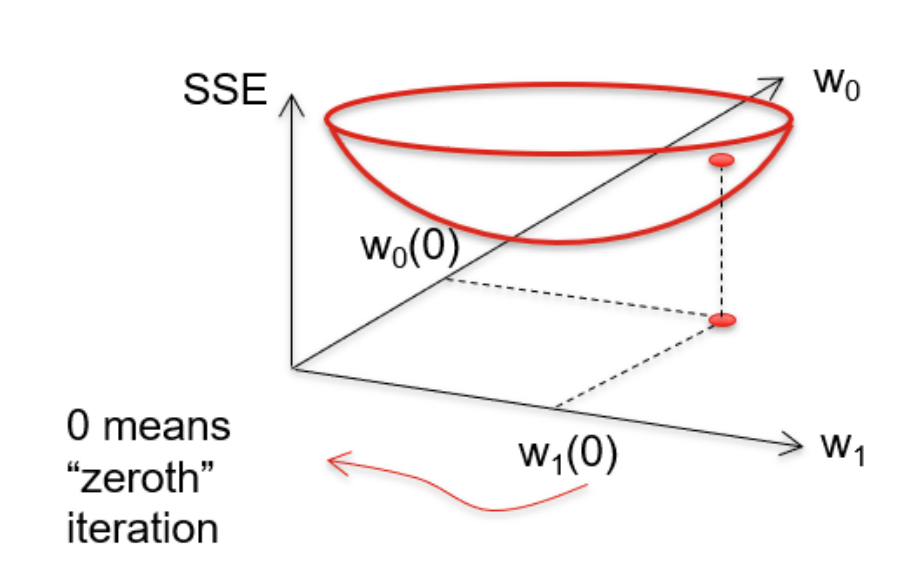
- The best w_1 and w_0 is when the SSE is at its lowest point.



- In practice, however, we might not have the luxury of testing all possible combinations of w_1 and w_0.
- We need something computationally less intensive.

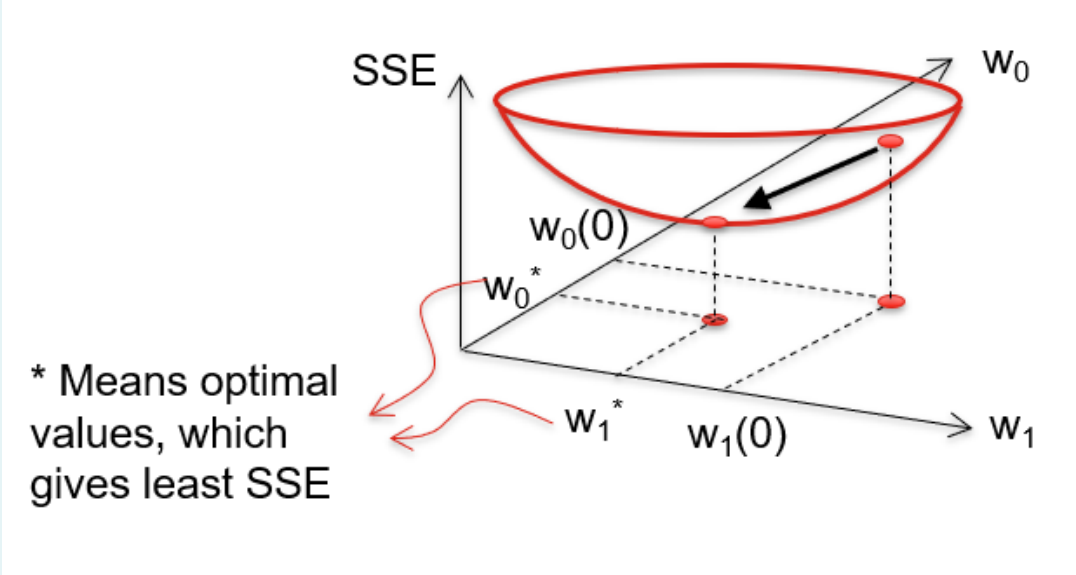
Revision: Steepest Descent - Batch (7)

- Coming back to our randomly generated weights, the SSE may be as shown:



Revision: Steepest Descent - Batch (8)

- We want to use an algorithm such that the weights move to the optimal values.



Revision: Steepest Descent - Batch (9)

- The idea is to adjust the weight in the direction of steepest descent.
  - Where the direction of steepest descent is opposite to the gradient.

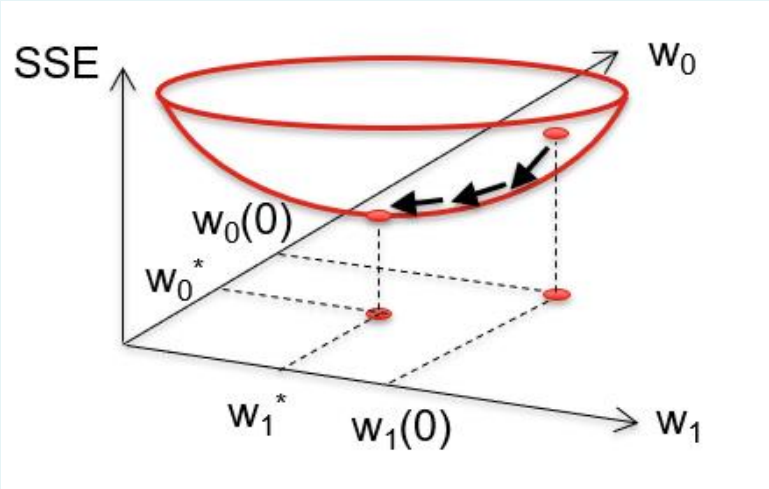
$$w_0(t+1) = w_0(t) - \eta \frac{\partial(\text{SSE})}{\partial(w_0)} \text{ and } w_1(t+1) = w_1(t) - \eta \frac{\partial(\text{SSE})}{\partial(w_1)}$$

  - η is the learning rate parameter.
- Because $\text{SSE} = (d_1 - w_1x_1 - w_0)^2 + \cdots + (d_n - w_1x_n - w_0)^2$, we have:

$$w_0(t+1) = w_0(t) - \eta[-2(d_1 - w_1x_1 - w_0) - \cdots - 2(d_n - w_1x_n - w_0)]$$
$$w_1(t+1) = w_1(t) - \eta[-2(d_1 - w_1x_1 - w_0)x_1 - \cdots - 2(d_n - w_1x_n - w_0)x_n]$$

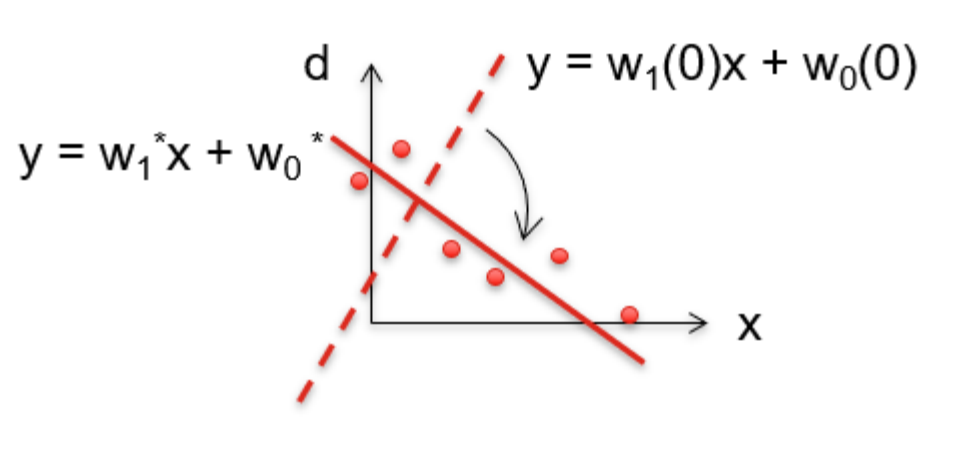
Revision: Steepest Descent - Batch (10)

- Depending on η, it is very likely that it will take several iterations to reach the optimum point.
- Therefore, this is a iterative process.



Revision: Steepest Descent - Batch (11)

- At the end, the optimal values would give the best-fit line:

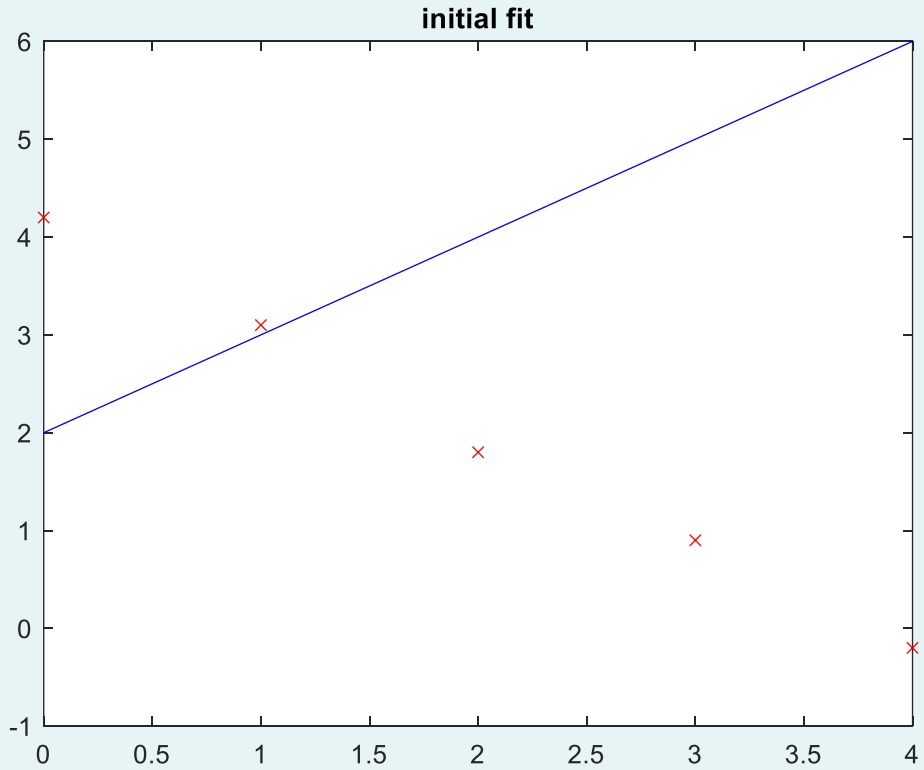


MATLAB Example – Batch (1)

```
%% Data

x = [0 1 2 3 4];
y = [4.2 3.1 1.8 0.9 -0.2];

figure, plot(x,y,'rx')

len = length(x);
```

```
%% Settings

w0 = 2; % initial w0
w1 = 1; % initial w1
eta = 0.01; % learning rate
iteration = 1000; % number of iterations
```

```
%% plot initial fit

ytestx0 = w1*0+w0;
ytestx4 = w1*4+w0;
hold on, plot([0,4],[ytestx0,ytestx4],'b')
title('initial fit')
```

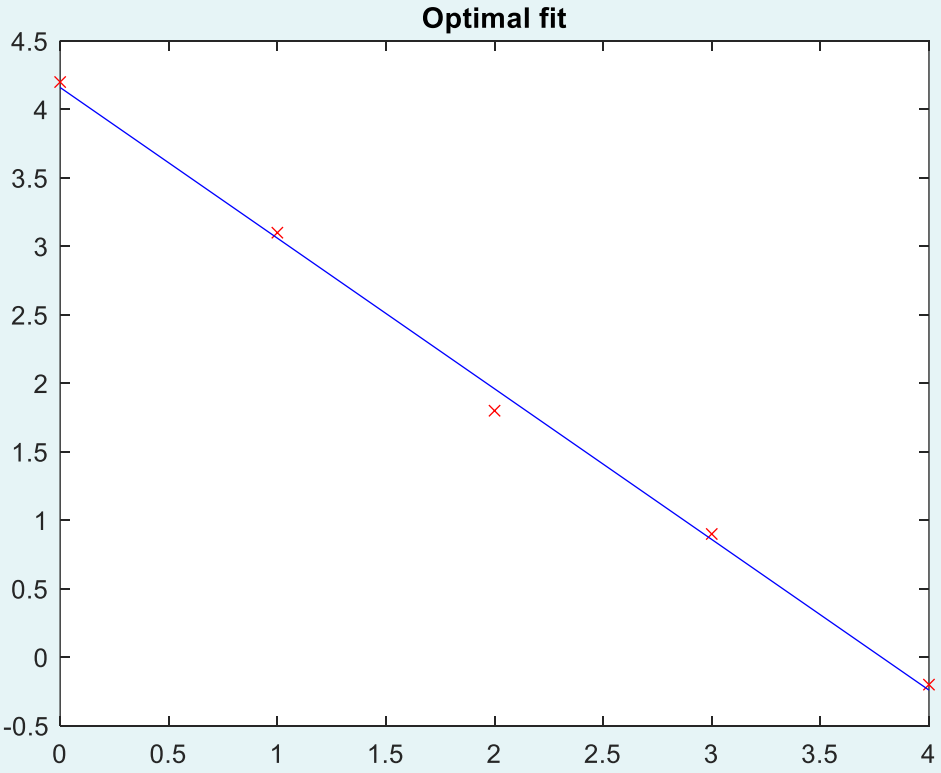


MATLAB Example – Batch (2)

```
%% Algorithm

for i = 1:iteration
    grad_w0 = 0;
    grad_w1 = 0;
    cost = 0;
    for j = 1:len
        grad_w0 = grad_w0 - 2*(y(j)-w1*x(j)-w0); % summing grad_w0 from data_1 to data_n
        grad_w1 = grad_w1 -2*(y(j)-w1*x(j)-w0)*x(j); % summing grad_w1 from data_1 to data_n
        cost = cost + (y(j)-w1*x(j)-w0)^2; % summing cost from data_1 to data_n
    end
    w0 = w0 - eta*grad_w0;
    w1 = w1 - eta*grad_w1;
    w0record(i) = w0;
    w1record(i) = w1;
    costrecord(i) = cost;
end
```

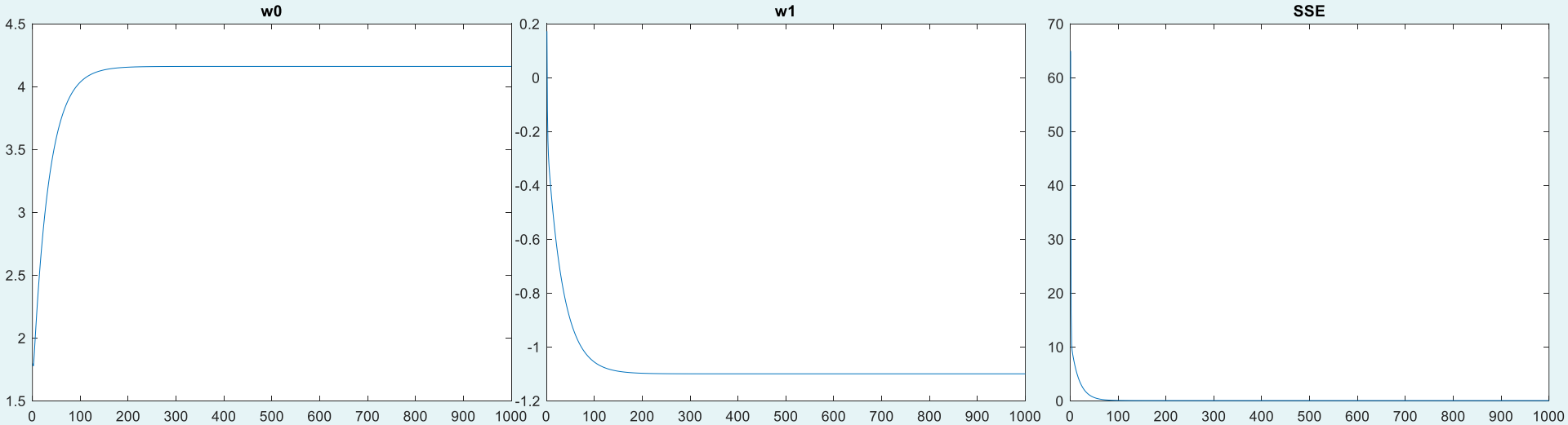
MATLAB Example – Batch (3)

```
%% plot result

ytestx0 = w1*0+w0;
ytestx4 = w1*4+w0;
figure, plot(x,y,'rx')
hold on, plot([0,4],[ytestx0,ytestx4],'b')
title('Optimal fit')

figure,plot(w0record), title('w0')
figure,plot(w1record), title('w1');
figure,plot(costrecord), title('SSE');
```



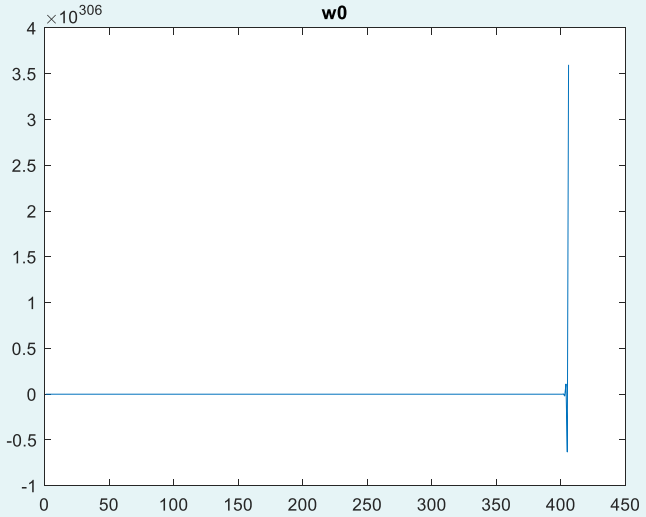
MATLAB Example (4)

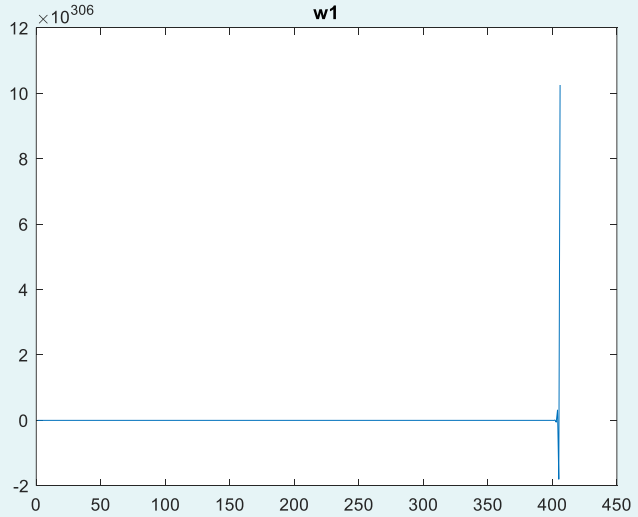
- The values of w_0, w_1 and SSE during the iterations are shown below:



MATLAB Example (5)

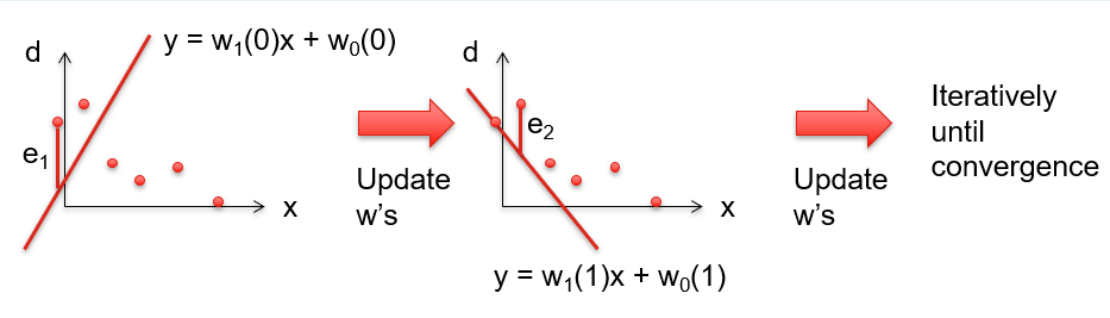
- Note: The setting of learning rate (η) is important.
- If η is too large, the algorithm may not converge!
- E.g. same Matlab code, but $\eta = 0.1$





Steepest Descent – Sequential (1)

- What you have seen so far is "batch" mode, i.e. all the data are presented at once before weight updates.
  - Slow if a lot of data, as need to wait until all are added.
- Another possibility is the "sequential" mode, where data is presented one at a time, and the weights are updated after presentation of each data.



Steepest Descent – Sequential (2)

- For sequential mode, since only one data is presented at a particular instance, we do not have *sum* of squared errors. Rather, we have the instantaneous squared error.

$$\boxed{E(w,i) = \frac{1}{2}e(i)^2}$$

where i refers to time i.

Steepest Descent – Sequential (3)

- For the same example,

$$E(w,i) = \frac{1}{2}e(i)^2 = \frac{1}{2}(d_i - w_1 x_i - w_0)^2$$

- The weights are updated as follows:

$$\begin{aligned} w_0(t+1) &= w_0(t) - \eta \frac{\partial(E)}{\partial(w_0)} \\ &= w_0(t) + \eta(d_i - w_1 x_i - w_0) \\ &= w_0(t) + \eta e(i) \end{aligned}$$

$$\begin{aligned} w_1(t+1) &= w_1(t) - \eta \frac{\partial(E)}{\partial(w_1)} \\ &= w_1(t) + \eta(d_i - w_1 x_i - w_0)x_i \\ &= w_1(t) + \eta e(i)x_i \end{aligned}$$

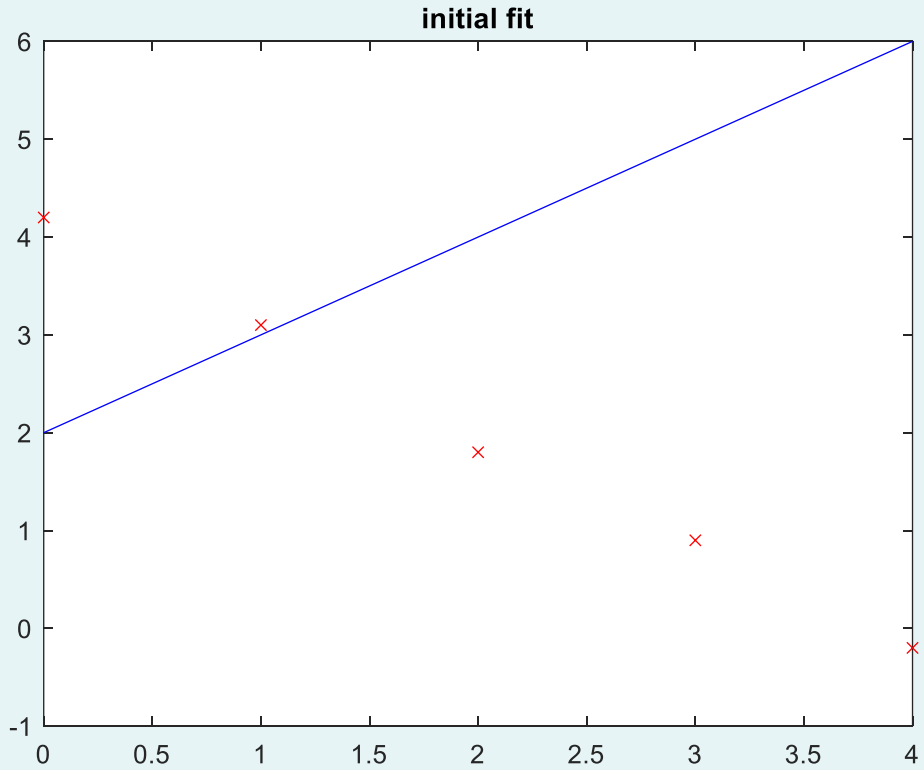
- This is also called "Stochastic Gradient Descent (SGD)"

MATLAB Example – Sequential (1)

```
%% Data

x = [0 1 2 3 4];
y = [4.2 3.1 1.8 0.9 -0.2];

figure, plot(x,y,'rx')

len = length(x);
```

```
%% Settings

w0 = 2; % initial w0
w1 = 1; % initial w1
eta = 0.01; % learning rate
iteration = 1000; % number of iterations
```

```
%% plot initial fit

ytestx0 = w1*0+w0;
ytestx4 = w1*4+w0;
hold on, plot([0,4],[ytestx0,ytestx4],'b')
title('initial fit')
```

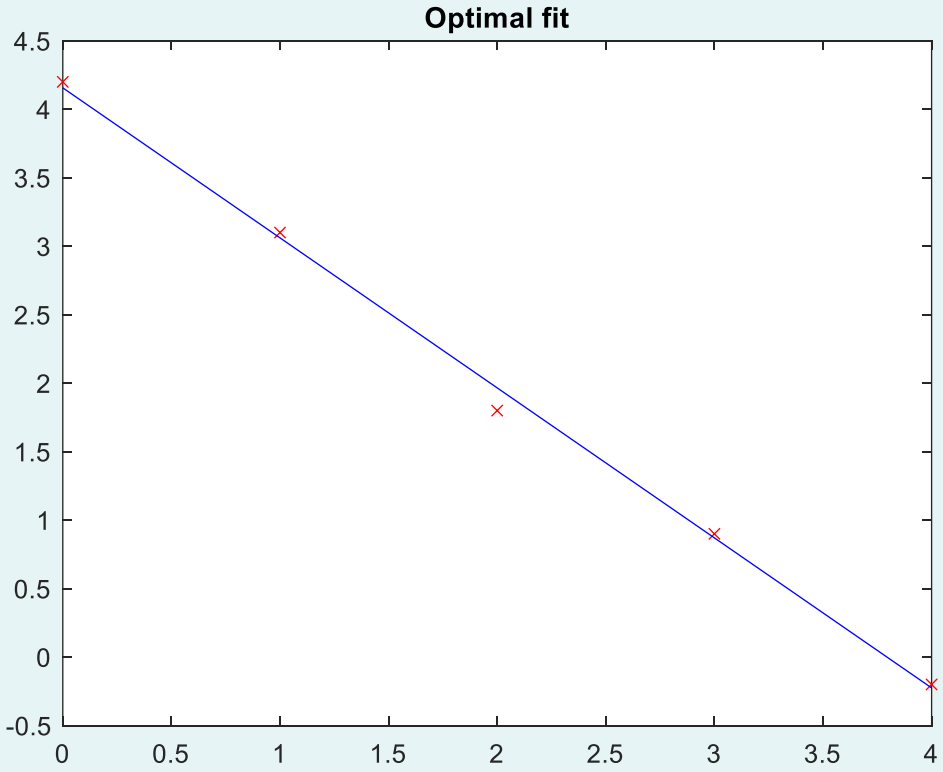


MATLAB Example – Sequential (2)

```
%% Algorithm

w0record = [];
w1record = [];
for i = 1:iteration
    cost = 0;
    for j = 1:len
        grad_w0 = -(y(j)-w1*x(j)-w0);
        grad_w1 = -(y(j)-w1*x(j)-w0)*x(j);
        w0 = w0 - eta*grad_w0;
        w1 = w1 - eta*grad_w1;
        w0record = [w0record;w0];
        w1record = [w1record;w1];
    end
end
```

MATLAB Example – Sequential (3)

```
%% plot result

ytestx0 = w1*0+w0;
ytestx4 = w1*4+w0;
figure, plot(x,y,'rx')
hold on, plot([0,4],[ytestx0,ytestx4],'b')
title('Optimal fit')

figure,plot(w0record), title('w0')
figure,plot(w1record), title('w1');
```



Steepest Descent – Minibatch (1)

- There is another version which sits in between purely batch mode and purely sequential mode – the "Minibatch" mode.
  - Rather than waiting for all the data to be presented before weight is updated (batch mode), or
  - Rather than updating with presentation of every data (sequential),
  - We update the weights after a fixed portion of data is presented.
  - E.g. Number of data = 1000; Split into 50 minibatches; Thus each minibatch has 20 data.
  - Update data after presentation of one minibatch of data, and continue to the next minibatch.

Loss Function & Cost Function (1)

- You will come across the terms “loss function” and “cost function” in machine learning.
- Loss function L: Defined with respect to one single training example.
- Cost function J: Average of loss function of the entire training set.
  - Examples:
    - Loss function: Square of errors: $\boxed{L(i) = e(i)^2 = \big(d(i) - y(i)\big)^2}$
    - Cost Function: SSE with average: $\boxed{J = \frac{1}{n}\sum_{i=1}^{n} L(i) = \frac{1}{n}\sum_{i=1}^{n}\big(d(i) - y(i)\big)^2}$

Loss Function & Cost Function (2)

- For binary classification (class 0, class 1), another commonly used loss function is the binary cross-entropy:

$$\boxed{L(i) = -\left[d(i)log\left(y(i)\right) + \left(1 - d(i)\right)log\left(1 - y(i)\right)\right]}$$

- Its cost function is therefore:

$$\boxed{J = \frac{1}{n}\sum_{i=1}^{n} L\,(i) = -\frac{1}{n}\sum_{i=1}^{n}\left[d(i)log\left(y(i)\right) + \left(1 - d(i)\right)log\left(1 - y(i)\right)\right]}$$

Loss Function & Cost Function (3)

- How does binary cross-entropy work?

$$\boxed{L(i) = -\left[d(i)log\left(y(i)\right) + \left(1 - d(i)\right)log\left(1 - y(i)\right)\right]}$$

- If $d(i) = 1$, then $L(i) = -log\left(y(i)\right)$.
  - If we minimize $L(i)$, we will maximize $y(i)$, which is what we want!
- If $d(i) = 0$, then $L(i) = -log\left(1 - y(i)\right)$.
  - If we minimize $L(i)$, we will maximize $\left(1 - y(i)\right)$, which in turn minimizes $y(i)$, which is what we want!

Loss Function & Cost Function (4)

- If there are more than 2 classes, we can calculate separate cross-entropy loss for each class and sum the results:

$$L(i) = -\sum_{\text{class}=1}^{\#\text{classes}} \left[d_{\text{class}}(i) log\left(y_{\text{class}}(i) \right) \right]$$

Loss Function & Cost Function (5)

- Example 1:
  - $d_1 = 1, y_1 = 0.1 \rightarrow -d_1 log(y_1) = -1 \times (-1) = 1$
  - $d_2 = 0, y_2 = 0.9 \rightarrow -d_2 log(y_2) = 0$
  - $d_3 = 0, y_3 = 0.2 \rightarrow -d_3 log(y_3) = 0$
  - Sum = 1 → Wrong classification, big loss

$$L(i) = -\sum \left[d_{\text{class}}(i) log \left(y_{\text{class}}(i) \right) \right]$$

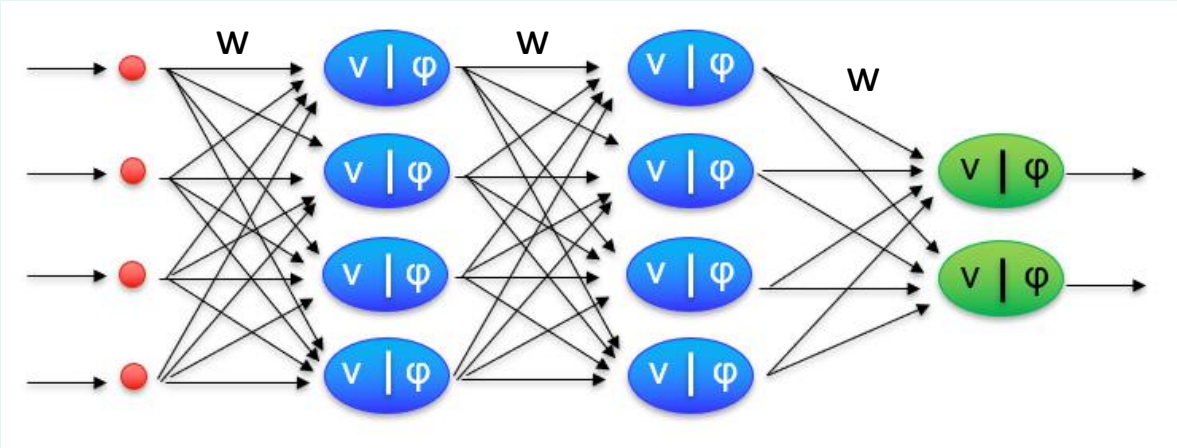
- Example 2:
  - $d_1 = 0, y_1 = 0.1 \rightarrow -d_1 log(y_1) = 0$
  - $d_2 = 1, y_2 = 0.9 \rightarrow -d_2 log(y_2) = -1 \times (-0.046) = 0.046$
  - $d_3 = 0, y_3 = 0.2 \rightarrow -d_3 log(y_3) = 0$
  - Sum = 0.046 → Correct classification, small loss

Content

- Revision: Optimization
- Backpropagation
- Generalization
- Discussions
- MATLAB NN Toolbox Example

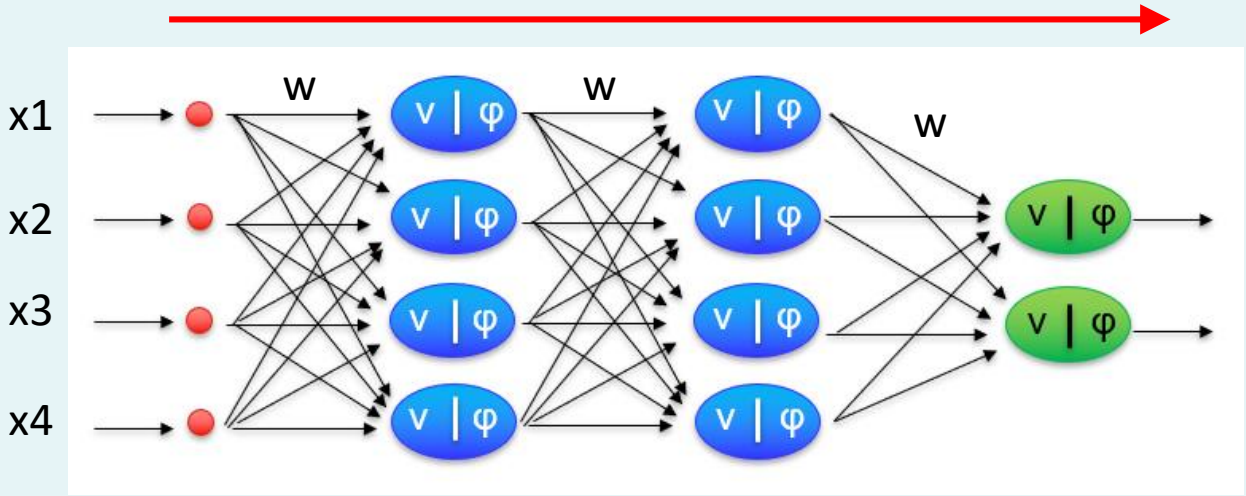
Intro to Backpropagation for MLP (1)

- The principle behind training a neural network is similar to finding the parameters w_0 and w_1 in the previous example.
  - First, the weights of the neural network are generated randomly.



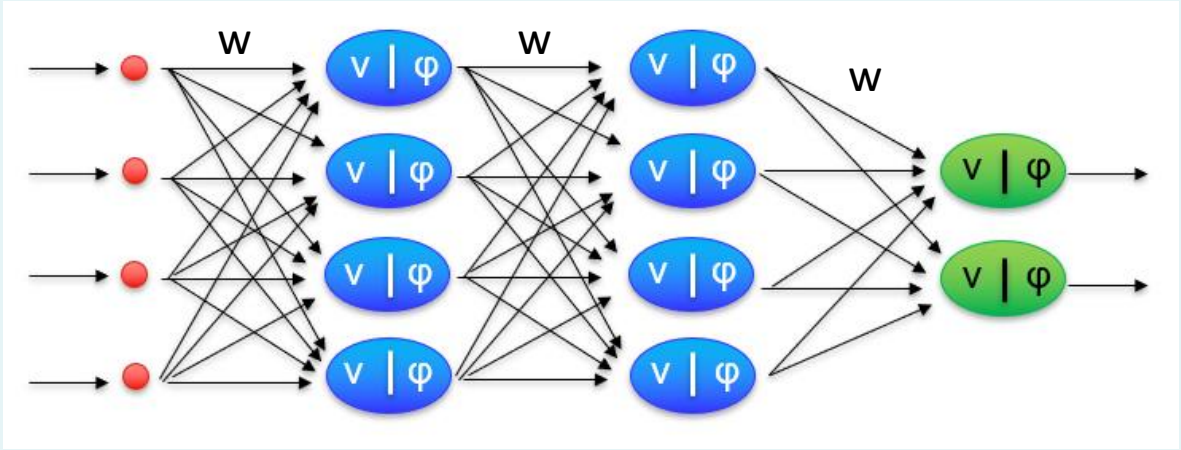
Intro to Backpropagation for MLP (2)

- Using the weights, a “forward pass” is run.
- The input data will flow through the network from left to right, each time multiplied by the weights and modified by the nonlinear activation function.



Intro to Backpropagation for MLP (3)

- The output of the network (y) will then be compared with the desired output (d), and the loss/cost function can be calculated.
- We then iteratively update the weights until the loss/cost is minimized.

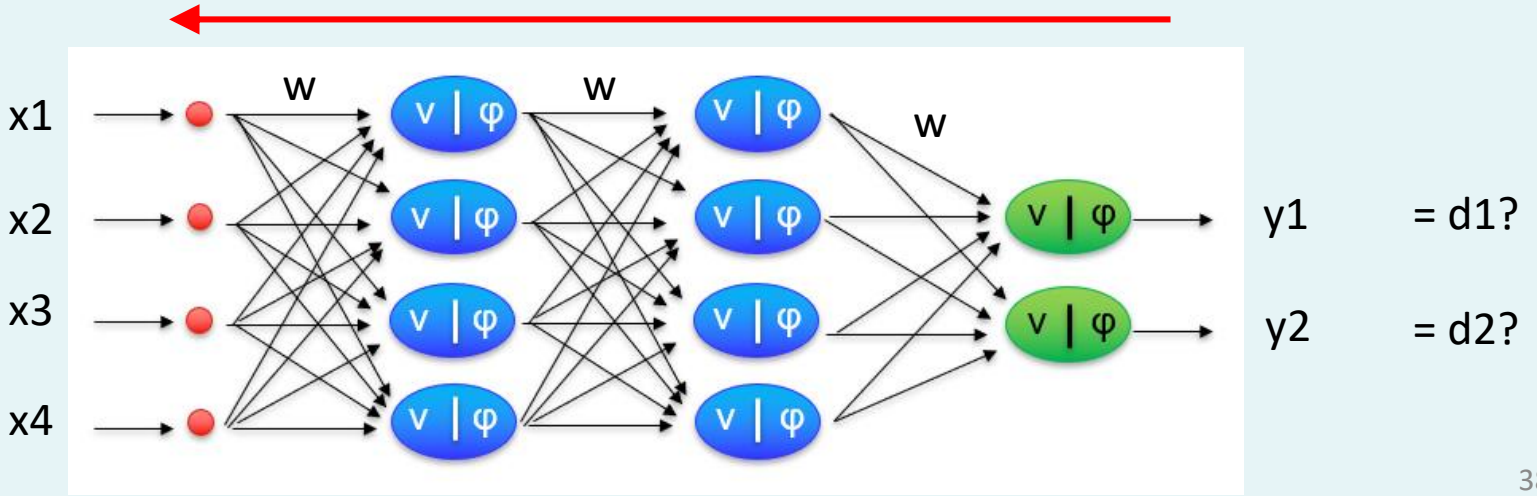


Intro to Backpropagation for MLP (4)

- Now that you have understood the method of steepest descent, it would be quite easy to understand the algorithm to update the weights in MLP.
- The derivation in this notes will be based on sequential mode (one training data at a time), and by using the instantaneous squared error as loss function.
  - You can try deriving your algorithm using other loss function or batch mode.

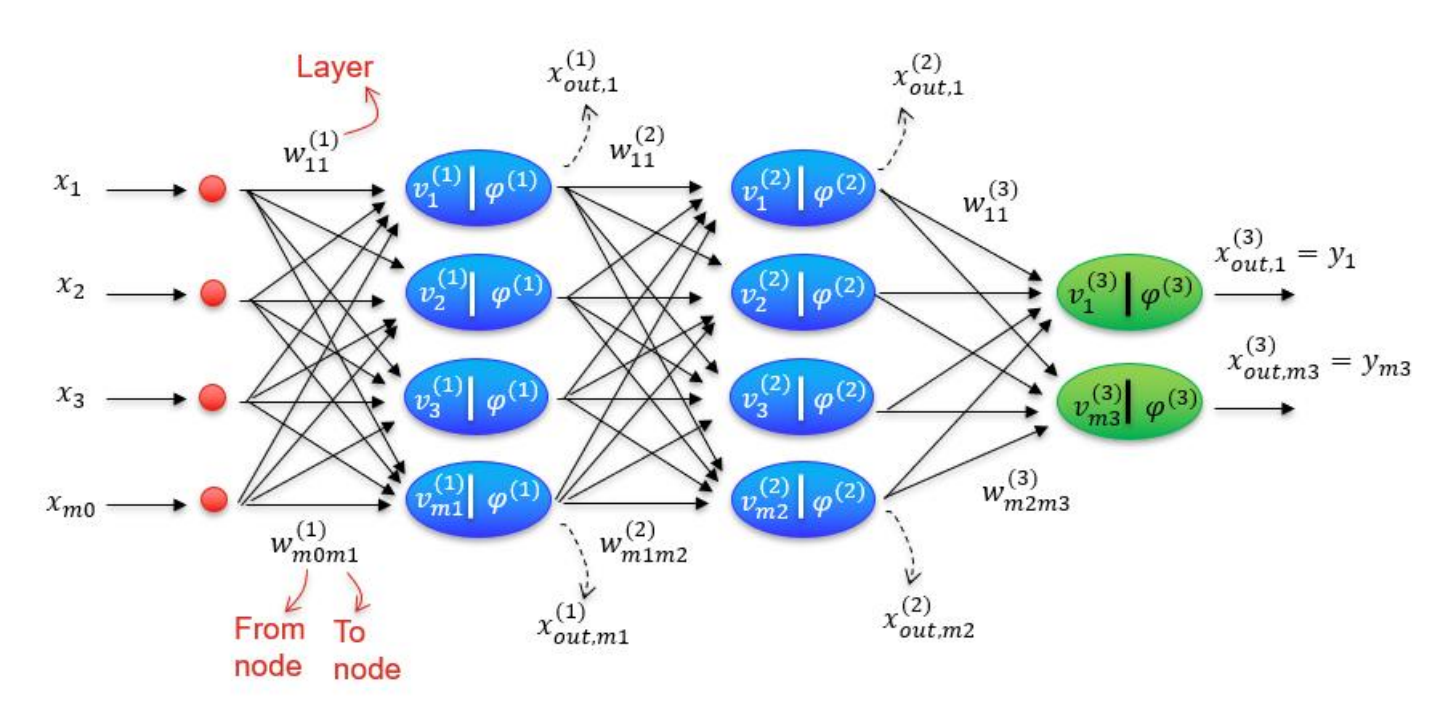
Intro to Backpropagation for MLP (5)

- The algorithm is called “Backpropagation (BP)”, because we update the weights from the last layer to the first layer.
  - i.e. the output layer first, then the last hidden layer, the second last hidden layer, up until the first hidden layer.)



Derivation of Backpropagation (1)

- Now, let's label the signals and weights appropriately.



Derivation of Backpropagation (2)

- The instantaneous squared errors of all the outputs for one instance of input data is:

$$L = \frac{1}{2}(d_1 - y_1)^2 + \frac{1}{2}(d_2 - y_2)^2 + \cdots + \frac{1}{2}(d_{m3} - y_{m3})^2$$

Derivation of BP Output Layer (1)

- To update the output layer weights:
  - Write the loss function in terms of output layer weights:

$$L = \frac{1}{2}\left(d_1 - \varphi^{(3)}\left(v_1^{(3)}\right)\right)^2 + \frac{1}{2}\left(d_2 - \varphi^{(3)}\left(v_2^{(3)}\right)\right)^2 + \cdots + \frac{1}{2}\left(d_{m3} - \varphi^{(3)}\left(v_{m3}^{(3)}\right)\right)^2$$

  - The v's are:

$$v_1^{(3)} = w_{11}^{(3)} x_{out,1}^{(2)} + w_{21}^{(3)} x_{out,2}^{(2)} + \cdots + w_{m21}^{(3)} x_{out,m2}^{(2)}$$

$$\vdots$$

$$v_{m3}^{(3)} = w_{1m3}^{(3)} x_{out,1}^{(2)} + w_{2m3}^{(3)} x_{out,2}^{(2)} + \cdots + w_{m2m3}^{(3)} x_{out,m2}^{(2)}$$

Derivation of BP Output Layer (2)

- Let's use $w_{21}^{(3)}$ as example. The method of steepest descent gives:

$$w_{21,new}^{(3)} = w_{21,old}^{(3)} - \eta^{(3)} \frac{\partial L}{\partial \left(w_{21}^{(3)} \right)}$$

$$= w_{21,old}^{(3)} - \eta^{(3)} \cdot \frac{\partial L}{\partial \left(v_1^{(3)} \right)} \cdot \frac{\partial \left(v_1^{(3)} \right)}{\partial \left(w_{21}^{(3)} \right)} \text{ using chain rule}$$

- $\frac{\partial \left(v_1^{(3)} \right)}{\partial \left(w_{21}^{(3)} \right)}$ is straightforward. Since $v_1^{(3)} = w_{11}^{(3)} x_{out,1}^{(2)} + w_{21}^{(3)} x_{out,2}^{(2)} + \cdots$,

$$\frac{\partial \left(v_1^{(3)} \right)}{\partial \left(w_{21}^{(3)} \right)} = x_{out,2}^{(2)}$$

Derivation of BP Output Layer (3)

- What about $\frac{\partial L}{\partial\left(v_1^{(3)}\right)}$?
- Since $L=\frac{1}{2}\left(d_1-\varphi^{(3)}\left(v_1^{(3)}\right)\right)^2+\frac{1}{2}\left(d_2-\varphi^{(3)}\left(v_2^{(3)}\right)\right)^2+\cdots,$

$$\boxed{\frac{\partial L}{\partial\left(v_1^{(3)}\right)}=-\left(\underbrace{d_1-\varphi^{(3)}\left(v_1^{(3)}\right)}_{\substack{\text{Error of}\\ \text{1st output node}}}\right)\frac{\partial\varphi^{(3)}}{\partial v_1^{(3)}}\triangleq-\delta_1^{(3)}}$$

- Therefore:

$$\boxed{w_{21,new}^{(3)}=w_{21,old}^{(3)}+\eta^{(3)}\cdot\delta_1^{(3)}\cdot x_{out,2}^{(2)}}$$

Derivation of BP Output Layer (4)

- Similarly, for all other weights in the output layer, we have:

$$w_{ij,new}^{(3)} = w_{ij,old}^{(3)} + \eta^{(3)} \cdot \delta_j^{(3)} \cdot x_{out,i}^{(2)}$$

  - Where:

$$\delta_j^{(3)} = \left(\underbrace{d_j - \varphi^{(3)}\left(v_j^{(3)}\right)}_{\substack{\text{Error of} \\ \text{jth output node}}} \right) \frac{\partial \varphi^{(3)}}{\partial v_j^{(3)}}$$

  - is called the local gradient (or local error) at the j-th neuron.

Derivation of BP Hidden Layer (1)

- The derivation of BP algorithm for hidden layer is not much different from that of the output layer.
- General idea:
  - Express the loss function in terms of output layer weights.
  - Then express the inputs of the output layer in terms of hidden layer weights.
  - The derivatives can be then obtained by chain rule.
- Of course, the expression will be slightly different (and longer!)

Derivation of BP Hidden Layer (2)

- Again, start with the loss function:

$$L = \frac{1}{2}(d_1 - y_1)^2 + \frac{1}{2}(d_2 - y_2)^2 + \cdots + \frac{1}{2}(d_{m3} - y_{m3})^2$$

- Still write in terms of output layer weights:

$$L = \frac{1}{2}\left(d_1 - \varphi^{(3)}\left(v_1^{(3)}\right)\right)^2 + \frac{1}{2}\left(d_2 - \varphi^{(3)}\left(v_2^{(3)}\right)\right)^2 + \cdots + \frac{1}{2}\left(d_{m3} - \varphi^{(3)}\left(v_{m3}^{(3)}\right)\right)^2$$

  - The v's are:

$$v_1^{(3)} = w_{11}^{(3)} x_{out,1}^{(2)} + w_{21}^{(3)} x_{out,2}^{(2)} + \cdots + w_{m21}^{(3)} x_{out,m2}^{(2)}$$

$$v_{m3}^{(3)} = w_{1m3}^{(3)} x_{out,1}^{(2)} + w_{2m3}^{(3)} x_{out,2}^{(2)} + \cdots + w_{m2m3}^{(3)} x_{out,m2}^{(2)}$$

Derivation of BP Hidden Layer (3)

- We continue to express the terms using signals/weights of 2nd hidden layers:

$$x_{out,1}^{(2)} = \varphi^{(2)}\left(v_1^{(2)}\right) \text{ with } v_1^{(2)} = w_{11}^{(2)} x_{out,1}^{(1)} + w_{21}^{(2)} x_{out,2}^{(1)} + \cdots + w_{m11}^{(2)} x_{out,m1}^{(1)}$$

$$x_{out,2}^{(2)} = \varphi^{(2)}\left(v_2^{(2)}\right) \text{ with } v_2^{(2)} = w_{12}^{(2)} x_{out,1}^{(1)} + w_{22}^{(2)} x_{out,2}^{(1)} + \cdots + w_{m12}^{(2)} x_{out,m1}^{(1)}$$

$$\vdots$$

$$x_{out,m2}^{(2)} = \varphi^{(2)}\left(v_{m2}^{(2)}\right) \text{ with } v_{m2}^{(2)} = w_{1m2}^{(2)} x_{out,1}^{(1)} + w_{2m2}^{(2)} x_{out,2}^{(1)} + \cdots + w_{m1m2}^{(2)} x_{out,m1}^{(1)}$$

Derivation of BP Hidden Layer (4)

- Let's use $w^{(2)}{}_{21}$ as example:

$$w_{21,new}^{(2)} = w_{21,old}^{(2)} - \eta^{(2)} \frac{\partial L}{\partial \left(w_{21}^{(2)} \right)}$$

- If we look at the equations in the previous two slides carefully, we see that:
  - $w_{21}^{(2)}$ appears in the $v_1^{(2)}$ equation.
  - $v_1^{(2)}$ appears in the $x_{out,1}^{(2)}$ equation.
  - $x_{out,1}^{(2)}$ appears several times in $v_1^{(3)}, v_2^{(3)}, \ldots, v_{m3}^{(3)}$ equations.
  - $v_1^{(3)}, v_2^{(3)}, \ldots, v_{m3}^{(3)}$ appear in L equation.

Derivation of BP Hidden Layer (5)

- Therefore:

$$\frac{\partial L}{\partial\left(w_{21}^{(2)}\right)}=\frac{\partial L}{\partial\left(v_1^{(3)}\right)}\cdot\frac{\partial\left(v_1^{(3)}\right)}{\partial\left(x_{out,1}^{(2)}\right)}\cdot\frac{\partial\left(x_{out,1}^{(2)}\right)}{\partial\left(v_1^{(2)}\right)}\cdot\frac{\partial\left(v_1^{(2)}\right)}{\partial\left(w_{21}^{(2)}\right)}+\frac{\partial L}{\partial\left(v_2^{(3)}\right)}\cdot\frac{\partial\left(v_2^{(3)}\right)}{\partial\left(x_{out,1}^{(2)}\right)}\cdot\frac{\partial\left(x_{out,1}^{(2)}\right)}{\partial\left(v_1^{(2)}\right)}\cdot\frac{\partial\left(v_1^{(2)}\right)}{\partial\left(w_{21}^{(2)}\right)}$$
$$+\cdots+\frac{\partial L}{\partial\left(v_{m3}^{(3)}\right)}\cdot\frac{\partial\left(v_{m3}^{(3)}\right)}{\partial\left(x_{out,1}^{(2)}\right)}\cdot\frac{\partial\left(x_{out,1}^{(2)}\right)}{\partial\left(v_1^{(2)}\right)}\cdot\frac{\partial\left(v_1^{(2)}\right)}{\partial\left(w_{21}^{(2)}\right)}$$

$$=\left(\begin{array}{c}\frac{\partial L}{\partial\left(v_1^{(3)}\right)}\cdot\frac{\partial\left(v_1^{(3)}\right)}{\partial\left(x_{out,1}^{(2)}\right)}+\frac{\partial L}{\partial\left(v_2^{(3)}\right)}\cdot\frac{\partial\left(v_2^{(3)}\right)}{\partial\left(x_{out,1}^{(2)}\right)}\\+\cdots\\+\frac{\partial L}{\partial\left(v_{m3}^{(3)}\right)}\cdot\frac{\partial\left(v_{m3}^{(3)}\right)}{\partial\left(x_{out,1}^{(2)}\right)}\end{array}\right)\cdot\frac{\partial\left(x_{out,1}^{(2)}\right)}{\partial\left(v_1^{(2)}\right)}\cdot\frac{\partial\left(v_1^{(2)}\right)}{\partial\left(w_{21}^{(2)}\right)}\qquad\text{-- (A)}$$

Derivation of BP Hidden Layer (6)

- Good news! $\frac{\partial L}{\partial\left(v_1^{(3)}\right)},\frac{\partial L}{\partial\left(v_2^{(3)}\right)},\ldots,\frac{\partial L}{\partial\left(v_{m3}^{(3)}\right)}$ have already been calculated when we updated output layer just now:

$$\boxed{\frac{\partial L}{\partial\left(v_1^{(3)}\right)}=-\delta_1^{(3)}}\ \boxed{\frac{\partial L}{\partial\left(v_2^{(3)}\right)}=-\delta_2^{(3)}}\ \cdots\ \boxed{\frac{\partial L}{\partial\left(v_{m3}^{(3)}\right)}=-\delta_{m3}^{(3)}}$$

  - So there is no need to re-calculate these terms.

- Next:

$$\boxed{\frac{\partial\left(v_1^{(3)}\right)}{\partial\left(x_{out,1}^{(2)}\right)}=w_{11}^{(3)}}\ \boxed{\frac{\partial\left(v_2^{(3)}\right)}{\partial\left(x_{out,1}^{(2)}\right)}=w_{12}^{(3)}}\ \cdots\ \boxed{\frac{\partial\left(v_{m3}^{(3)}\right)}{\partial\left(x_{out,1}^{(2)}\right)}=w_{1m3}^{(3)}}$$

Derivation of BP Hidden Layer (7)

- The final two terms in equation (A) are:

$$\frac{\partial\left(x_{out,1}^{(2)}\right)}{\partial\left(v_1^{(2)}\right)} = \frac{\partial\left(\varphi^{(2)}\left(v_1^{(2)}\right)\right)}{\partial\left(v_1^{(2)}\right)}$$

$$\frac{\partial\left(v_1^{(2)}\right)}{\partial\left(w_{21}^{(2)}\right)} = x_{out,2}^{(1)}$$

Derivation of BP Hidden Layer (8)

- Putting all the individual terms together into (A):

$$\frac{\partial L}{\partial\left(w_{21}^{(2)}\right)}=\left(\begin{array}{c}\frac{\partial L}{\partial\left(v_1^{(3)}\right)}\cdot\frac{\partial\left(v_1^{(3)}\right)}{\partial\left(x_{out,1}^{(2)}\right)}+\frac{\partial L}{\partial\left(v_2^{(3)}\right)}\cdot\frac{\partial\left(v_2^{(3)}\right)}{\partial\left(x_{out,1}^{(2)}\right)}\\ +\cdots\\ +\frac{\partial L}{\partial\left(v_{m3}^{(3)}\right)}\cdot\frac{\partial\left(v_{m3}^{(3)}\right)}{\partial\left(x_{out,1}^{(2)}\right)}\end{array}\right)\cdot\frac{\partial\left(x_{out,1}^{(2)}\right)}{\partial\left(v_1^{(2)}\right)}\cdot\frac{\partial\left(v_1^{(2)}\right)}{\partial\left(w_{21}^{(2)}\right)}$$

$$=\left(\begin{array}{c}-\delta_1^{(3)}\cdot w_{11}^{(3)}-\delta_2^{(3)}\cdot w_{12}^{(3)}\\ -\cdots\\ -\delta_{m3}^{(3)}\cdot w_{1m3}^{(3)}\end{array}\right)\cdot\frac{\partial\left(\varphi^{(2)}\left(v_1^{(2)}\right)\right)}{\partial\left(v_1^{(2)}\right)}\cdot x_{out,2}^{(1)}$$

Derivation of BP Hidden Layer (9)

- The weight update is therefore:

$$w_{21,new}^{(2)} = w_{21,old}^{(2)} + \eta^{(2)} \begin{pmatrix} \delta_1^{(3)} \cdot w_{11}^{(3)} + \delta_2^{(3)} \cdot w_{12}^{(3)} \\ + \cdots \\ + \delta_{m3}^{(3)} \cdot w_{1m3}^{(3)} \end{pmatrix} \cdot \frac{\partial \left(\varphi^{(2)} \left(v_1^{(2)} \right) \right)}{\partial \left(v_1^{(2)} \right)} \cdot x_{out,2}^{(1)}$$

- Similarly, for all other weights:

$$w_{ij,new}^{(2)} = w_{ij,old}^{(2)} + \eta^{(2)} \begin{pmatrix} \delta_1^{(3)} \cdot w_{j1}^{(3)} + \delta_2^{(3)} \cdot w_{j2}^{(3)} \\ + \cdots \\ + \delta_{m3}^{(3)} \cdot w_{jm3}^{(3)} \end{pmatrix} \cdot \frac{\partial \left(\varphi^{(2)} \left(v_j^{(2)} \right) \right)}{\partial \left(v_j^{(2)} \right)} \cdot x_{out,i}^{(1)}$$

Derivation of BP Hidden Layer (10)

- The final equation can be generalized for all other hidden layers:

$$w_{ij,new}^{(s)} = w_{ij,old}^{(s)} + \eta^{(s)} \underbrace{\begin{pmatrix} \delta_1^{(s+1)} \cdot w_{j1}^{(s+1)} + \delta_2^{(s+1)} \cdot w_{j2}^{(s+1)} \\ + \cdots \\ + \delta_{m3}^{(s+1)} \cdot w_{jm3}^{(s+1)} \end{pmatrix} \cdot \frac{\partial\left(\varphi^{(s)}\left(v_j^{(s)}\right)\right)}{\partial\left(v_j^{(s)}\right)}}_{\delta_j^{(s)}} \cdot x_{out,i}^{(s-1)}$$

$$w_{ij,new}^{(s)} = w_{ij,old}^{(s)} + \eta^{(s)} \cdot \delta_j^{(s)} \cdot x_{out,i}^{(s-1)}$$

BP Algorithm – Summary (1)

- In summary, for both output and hidden layers, the weight updates are: $\boxed{w_{ij,new}^{(s)} = w_{ij,old}^{(s)} + \eta^{(s)} \cdot \delta_j^{(s)} \cdot x_{out,i}^{(s-1)}}$
  - The difference lies in how the local gradients are calculated.
  - For output layer:

$$\boxed{\delta_j^{(s)} = \left(d_j - \varphi^{(s)}\left(v_j^{(s)}\right)\right)\frac{\partial \varphi^{(s)}}{\partial v_j^{(s)}} = \left(d_j - y_j\right)\frac{\partial \varphi^{(s)}}{\partial v_j^{(s)}}}$$

  - For hidden layer:

$$\boxed{\delta_j^{(s)} = \begin{pmatrix} \delta_1^{(s+1)} \cdot w_{j1}^{(s+1)} + \delta_2^{(s+1)} \cdot w_{j2}^{(s+1)} \\ + \cdots \\ + \delta_{m3}^{(s+1)} \cdot w_{jm3}^{(s+1)} \end{pmatrix} \cdot \frac{\partial\left(\varphi^{(s)}\left(v_j^{(s)}\right)\right)}{\partial\left(v_j^{(s)}\right)}}$$

BP Algorithm – Summary (2)

- For the first hidden layer, i.e. when $s = 1$:

$$\boxed{w_{ij,new}^{(s)} = w_{ij,old}^{(s)} + \eta^{(s)} \cdot \delta_j^{(s)} \cdot x_{out,i}^{(s-1)}} \rightarrow \boxed{w_{ij,new}^{(1)} = w_{ij,old}^{(1)} + \eta^{(1)} \cdot \delta_j^{(1)} \cdot x_{out,i}^{(0)}}$$

- But what is $x_{out,i}^{(0)}$?
  - It's simply the input data, x_i.

- Note: Do not forget about the bias terms. Set $i = 0$ and $x_{out,0}^{(s-1)} = 1$.

$$\boxed{w_{0j,new}^{(s)} = w_{0j,old}^{(s)} + \eta^{(s)} \cdot \delta_j^{(s)}}$$

BP Algorithm – Summary (3)

- At this point, it is good to remind ourselves about the derivatives of activation functions:
  - ReLU:

$$\boxed{\varphi = max\{0, v\} = \begin{cases} 0, v < 0 \\ v, v \geq 0 \end{cases}} \rightarrow \boxed{\frac{d\varphi}{dv} = \begin{cases} 0, v < 0 \\ 1, v \geq 0 \end{cases}}$$

  - Leaky ReLU:

$$\boxed{\varphi(v) = max\{0.01v, v\} = \begin{cases} 0.01v, v < 0 \\ v, v \geq 0 \end{cases}} \rightarrow \boxed{\frac{d\varphi}{dv} = \begin{cases} 0.01, v < 0 \\ 1, v \geq 0 \end{cases}}$$

BP Algorithm – Summary (4)

- Log Sigmoid:

$$\boxed{\varphi(v) = \frac{1}{1+e^{-av}} = (1+e^{-av})^{-1}} \rightarrow \boxed{\frac{d\varphi}{dv} = a \cdot \big(1-\varphi(v)\big) \cdot \varphi(v)}$$

- Tanh:

$$\boxed{\varphi(v) = \tanh(v) = \frac{e^{v}-e^{-v}}{e^{v}+e^{-v}}} \rightarrow \boxed{\frac{d\varphi}{dv} = 1-\varphi^{2}(v)}$$

- Linear:

$$\boxed{\varphi(v) = v} \rightarrow \boxed{\frac{d\varphi}{dv} = 1}$$

BP Algorithm – Summary (5)

- Present first set of data.
- Forward pass:
  - Use previously calculated weights. (For first iteration, randomly initialized).
  - Compute signals for each neuron
- Backward pass:
  - Starting from the output layer, the local gradient is calculated towards the first layer (hence "backpropagation").
  - At each layer, the weights are updated based on the formula on previous slides.
- Present next set of data.
- Continue until convergence (i.e. loss function does not decrease much), or until the pre-set number of epochs are achieved.

Content

- Revision: Optimization
- Backpropagation
- Generalization
- Discussions
- MATLAB NN Toolbox Example

Generalization (1)

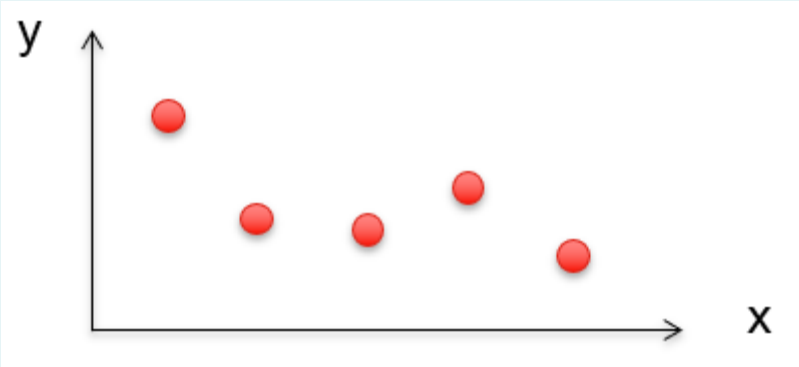
- When updating the weights, we present the network with a set of data, for which we know what the desired outputs are.
  - This is called “supervised training”, and the data set is called “training set”.
- After training, we would like to use the network for any unseen new data coming in.
  - This “new” data set is called “test set”.

Generalization (2)

- For e.g. a network has been trained to predict the wind speed of an area, based on historical data of temperature, pressure, previous day wind speed etc.
- Then the network is used to predict *tomorrow's* wind speed based on today's temperature, pressure and yesterday's wind speed.
  - Because today's weather pattern is different from any other historical data, it will test the network's ability to generalize.

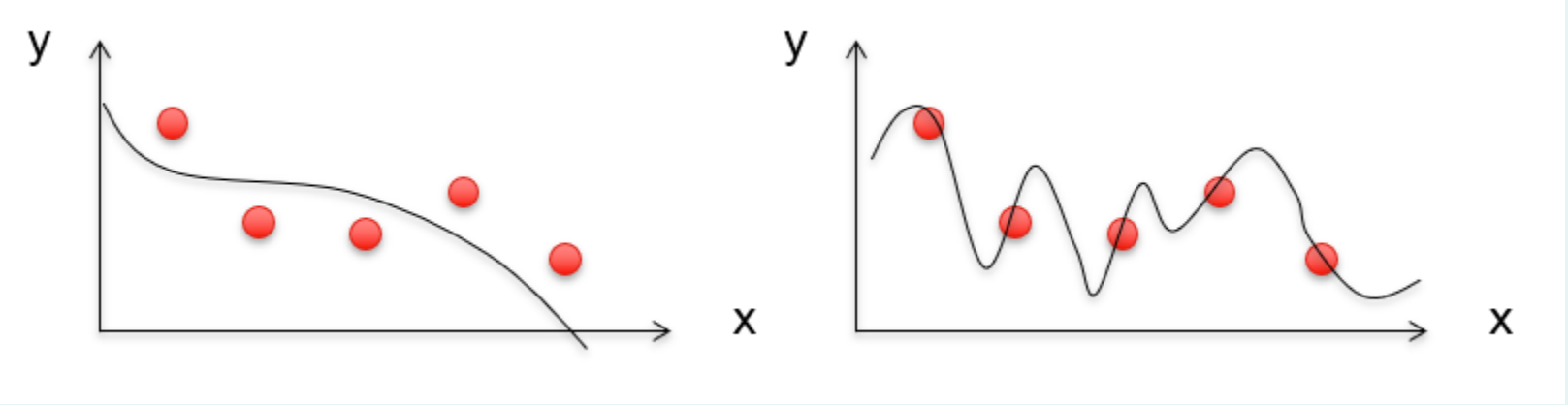
Generalization (3)

- Another example: Assume we have some training data, which are represented by the red dots.



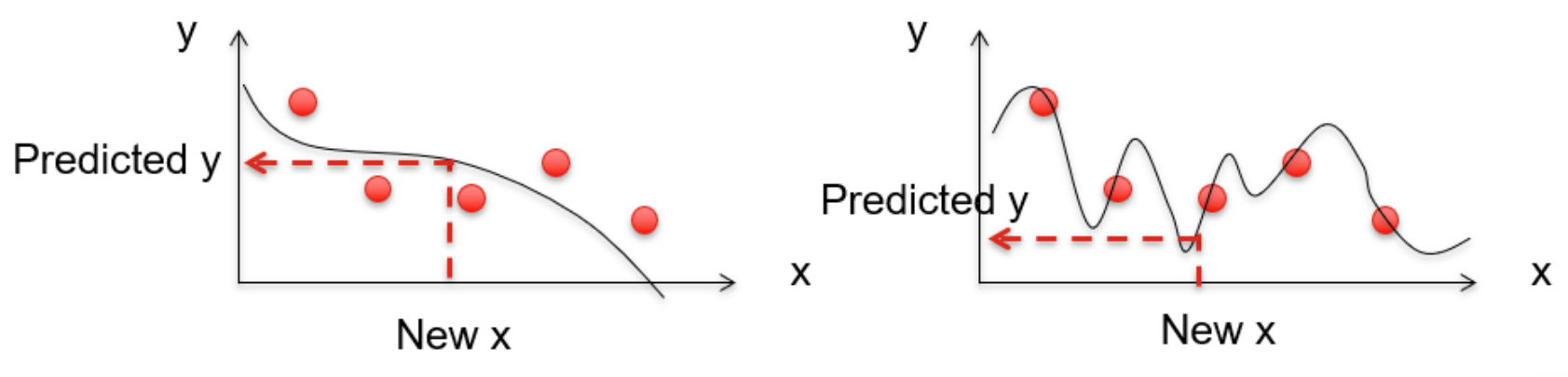
Generalization (4)

- The network could “fit” the data in the following ways:



Generalization (5)

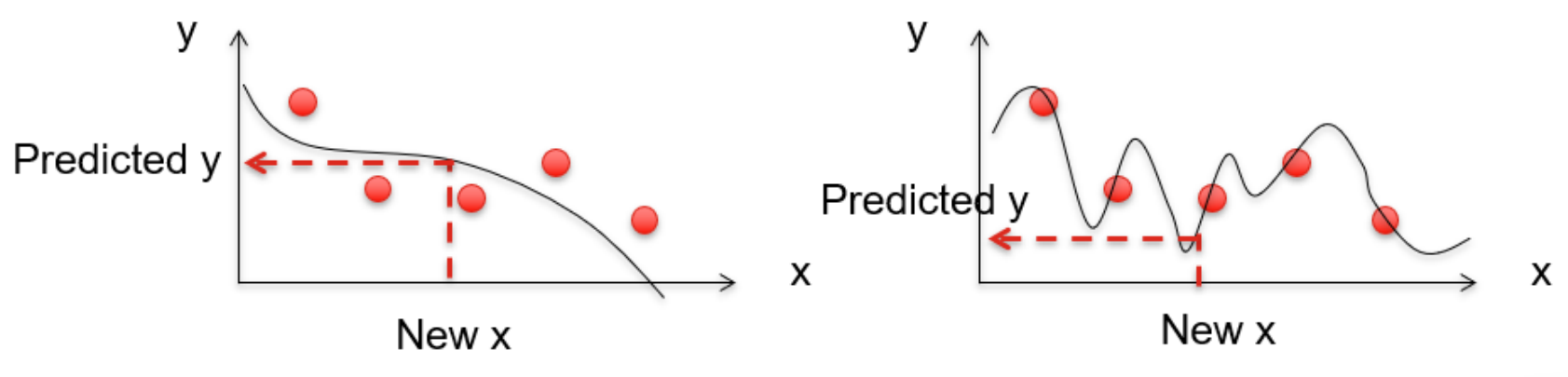
- Now we test the network with some new (unseen before) input data (x).



- It can be seen that the left network gives a better prediction, in face of new data.
  - The left network can generalize better than the right network.

Generalization (6)

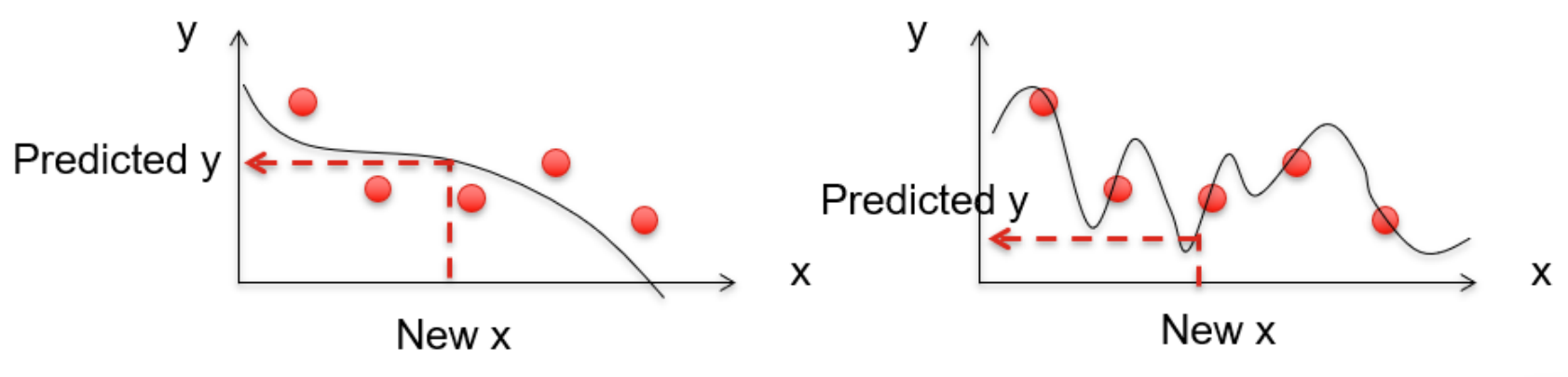
- How do we improve the generalization of a network?
- Firstly, observe the two curve fitting again:



- On the left, the curve do not really pass through the training points. So the training error is not really that small, or in other words, the training accuracy is high but not too high.

Generalization (7)

- How do we improve the generalization of a network?
- Firstly, observe the two curve fitting again:



- On the right, the curve passes through almost all the training points. So the training error is very small, or in other words, the training accuracy is extremely high.

Generalization (8)

- It is interesting to observe that:
  - Training accuracy high but not too high → Test accuracy good
  - Training accuracy extremely high → Test accuracy deteriorates!
    - This is called “overfitting”: The network learns too well such that it even learns noise or random fluctuations.

Generalization (9)

- So the way to improve generalization is to make sure that the network doesn't learn *too* well.
  - Limit the number of hidden neurons.
  - Stop the learning before it has time to overfit.
  - Limit the size of the weights.

Train, Validation, Test Sets (1)

- The first strategy on the previous slide (limit the number of hidden neurons), is done based on trial-and-error.
- That is, we train a network using training data, and check its generalization capability.
  - If the generalization is not good, then we retrain another model (perhaps with reduced number of layers or neurons) until we achieve good generalization.

Train, Validation, Test Sets (2)

- Now, we must make sure that the "test set" is truly unseen, and is NOT used at all to inform the choice of our model.
  - So we shouldn't use the test set during the trial-and-error process.
- However, we still need some data to check if the model has overfitted. What should we do?
- The idea is to leave out a small portion of the training set as "validation set". And we will use this validation set to evaluate the generalization.

Train, Validation, Test Sets (3)

- The process is therefore:
  - Train the network using the training set until good accuracy on training set;
  - Test the network accuracy on the validation set.
    - If accuracy on validation set is low, tune the network then reiterate the process.
  - When (and only when) the network is fully specified, test the network accuracy on an unseen test set.

Train, Validation, Test Sets (4)

- How do we specify the train, validation and test set?
  - Split the available data according to a suitable ratio. This is called the "Hold-out" validation.

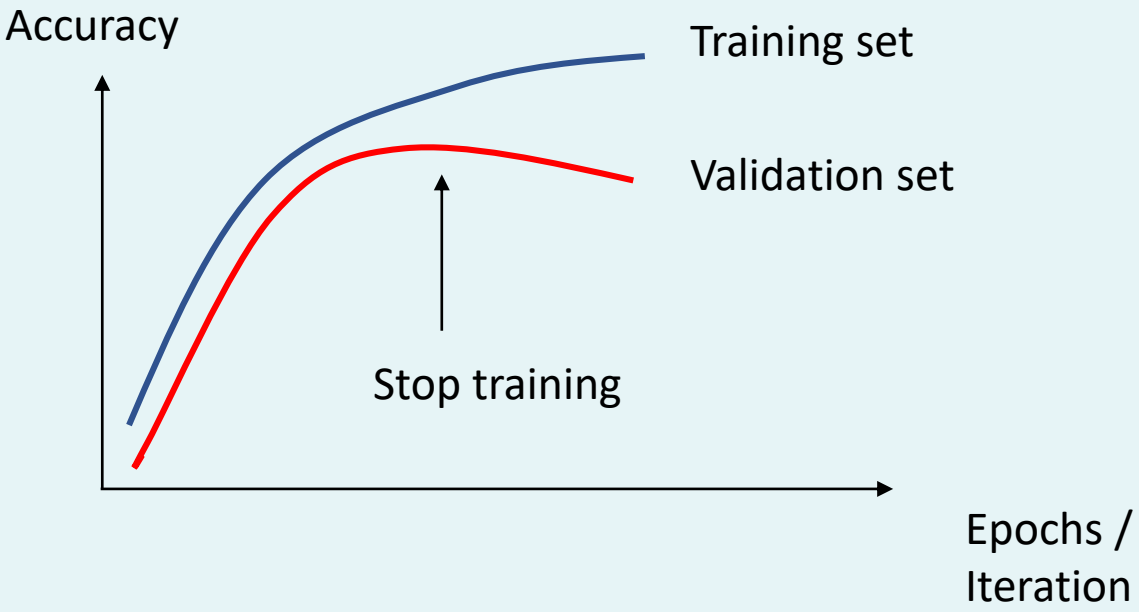
  - E.g. you have 10,000 data.
    - You may decide to split the data into 60:20:20, i.e. 6000 data for training, 2000 data for validation, and 2000 data for testing.
  - Make sure the train, validation and test sets come from the same distribution!

Train, Validation, Test Sets (5)

- The same method is also done for the second strategy, i.e. stop the learning before it has time to overfit.
  - Train the network using the training set;
  - Test the network accuracy on the validation set, after training for fixed numbers of epochs / iteration (e.g. 100, 200, 300…)
    - At the point when accuracy on validation set starts to decrease, stop the training.
  - When (and only when) the network is fully trained, test the network accuracy on an unseen test set.

Train, Validation, Test Sets (6)

- Examples:



Regularization (1)

- The third strategy to improve generalization (limit the size of the weights), is called “Regularization”.
  - One way to achieve this is to add a regularization term on the loss / cost function which would penalize large weights.
  - E.g. L1-regularization:

$$L = \frac{1}{2}(d_1 - y_1)^2 + \cdots + \frac{1}{2}(d_{m3} - y_{m3})^2 + \lambda \sum \|w\|$$

  - E.g. L2-regularization:

$$L = \frac{1}{2}(d_1 - y_1)^2 + \cdots + \frac{1}{2}(d_{m3} - y_{m3})^2 + \lambda \sum \|w\|^2$$

Regularization (2)

- Therefore, if the weights grow too large, then $\|w\|$ or $\|w\|^2$ will also be large, meaning that we are far from the optimal (minimal) value of the loss function.
- The optimization algorithm thus balances between the accuracy and the weights.

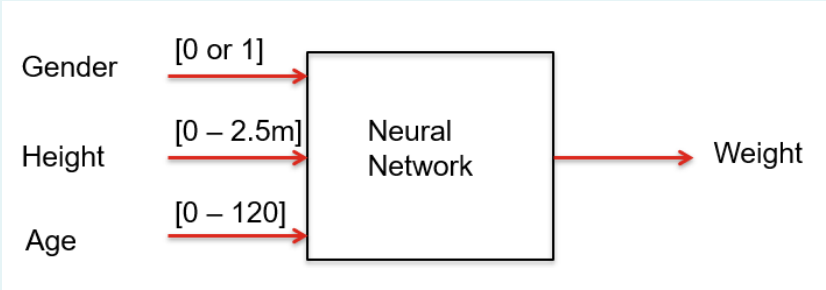
- Note: λ is a tuning parameter. Recommended to set as 0.01 and tune if needed.

Content

- Revision: Optimization
- Backpropagation
- Generalization
- Discussions
- MATLAB NN Toolbox Example

Normalization (1)

- Consider the following example:



  - Logically, the weight is more related to height than gender and age.
  - If the data for someone is [1,1.8, 60], a small change (e.g. 0.1) in the neural network weight changes the height factor by 0.18 and the age factor by 6.
  - It makes it harder to find the weights such that the effect of height has larger influence on the weight.

Normalization (2)

- Normalization (making the inputs into similar range) can be useful in such situations.
  - One way is to shift the inputs so that the average over training set is zero,
  - Followed by scaling the data to have unit variance.

$$\bar{x}_j = \frac{\sum_{k=1}^{n} x_j(k)}{n} \quad \& \quad \sigma = \sqrt{\frac{\sum_{k=1}^{n} \left(x_j(k) - \bar{x}_j\right)^2}{n}} \rightarrow x_{j,normalized}(k) = \frac{x_j(k) - \bar{x}_j}{\sigma}$$

Encoding Categorical Data (1)

- Assume you want to classify some images into dog, cat, lion, tiger.
- One way is to specify the desired outputs as dog = 1, cat = 2, lion = 3 and tiger = 4.
  - Your network would then have one output node, possibly with a linear or ReLU activation function.
  - If given a new image, and the network outputs 3.3, the result can be rounded to the closest integer (3) giving a prediction of lion.

Encoding Categorical Data (2)

- A more common way, however, is to use the following one-hot encoding:

Animal	Code
Dog	[1 0 0 0]
Cat	[0 1 0 0]
Lion	[0 0 1 0]
Tiger	[0 0 0 1]

  - You now have 4 output nodes, each one being trained on one digit.
  - Log sigmoid would be a suitable activation function.
  - If given a new image, and the network outputs [0.13, 0.19, 0.82, 0.21], we choose the max prediction (0.82) corresponding to lion.

Encoding Categorical Data (3)

- However, we should avoid 1 and 0 because these numbers can only be reached asymptotically by logistic function. The weights are driven to be larger and larger.
  - If using log sigmoid, 0.2 & 0.8 are good values.
  - If using tanh, -0.6 and 0.6 are good values.

Animal	Code if logsig	Code if tanh
Dog	[0.8 0.2 0.2 0.2]	[+0.6 -0.6 -0.6 -0.6]
Cat	[0.2 0.8 0.2 0.2]	[-0.6 +0.6 -0.6 -0.6]
Lion	[0.2 0.2 0.8 0.2]	[-0.6 -0.6 +0.6 -0.6]
Tiger	[0.2 0.2 0.2 0.8]	[-0.6 -0.6 -0.6 +0.6]

Nonlinear Optimization Problem (1)

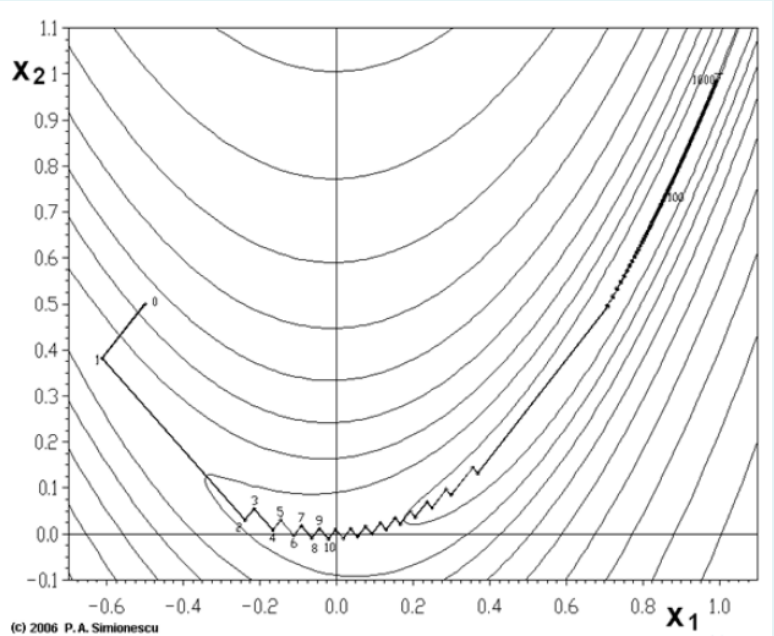
- We have so far looked at the steepest gradient method to solve the nonlinear optimization problem.

$$w(t+1) = w(t) - \eta \frac{\partial(L)}{\partial(w)}$$

- It can however be very slow.
- E.g. Rosenbrock's valley:

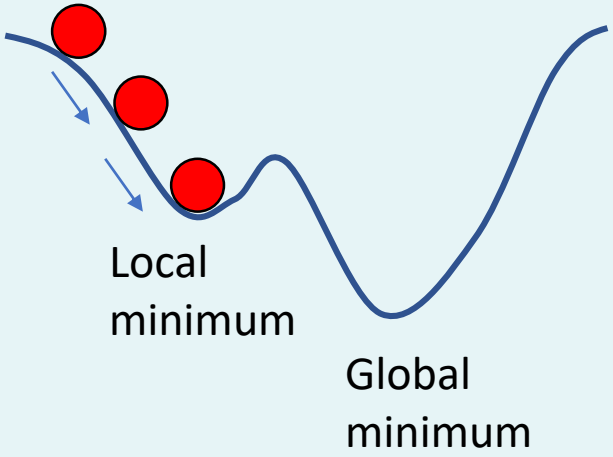
$$f(x,y) = (1-x)^2 + 100(y-x^2)^2$$

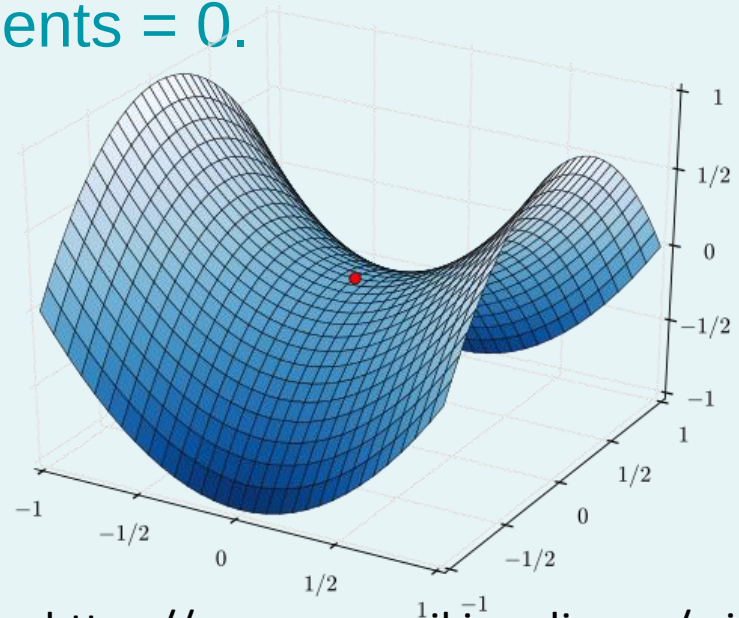
  - It has a global minimum at (1,1) but this is inside a long, narrow, parabolic shaped flat valley.



Nonlinear Optimization Problem (2)

- The solution can also get stuck in local minimum, or at saddle points, where gradients = 0.

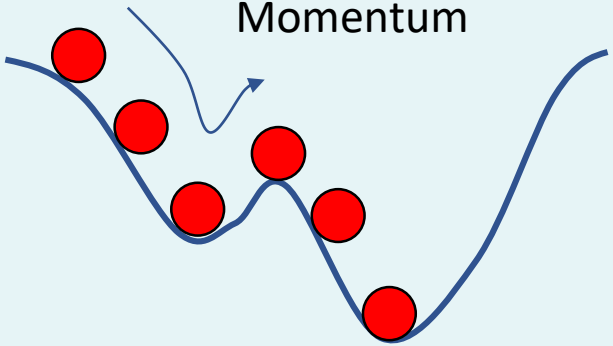




https://commons.wikimedia.org/wiki/File:Saddle_point.svg

SGD + Momentum (1)

- However, imagine a ball is rolling down the hill.
  - If it already has some velocity (momentum) before reaching the local minimum, it should have some chances to “get out” from the local minimum.



  - Researchers have therefore proposed to add a momentum term to the SGD algorithm.

SGD + Momentum (2)

SGD:

$$w(t+1) = w(t) - \eta \frac{\partial(L)}{\partial(w)}$$

SGD + Momentum:

$$v(t+1) = \rho v(t) + \frac{\partial(L)}{\partial(w)}$$
$$w(t+1) = w(t) - \eta v(t+1)$$

- For convergence, $0 \leq \rho < 1$.
- The momentum term builds up "velocity" as moving average of the gradients.
  - If dL/dW same sign on consecutive iteration, then v grows in magnitude.
  - If dL/dW different signs on consecutive iteration, then v shrinks, avoiding oscillations.

Adaptive Gradient AdaGrad (1)

- Another situation which might occur is that some input features may be sparse.
  - For e.g. your input has 4 dimensions, but x_2 is mostly zero.
  - Now, imagine the weights corresponding to that dimension:

$$\boxed{w_{2j,new}^{(1)} = w_{2j,old}^{(1)} + \eta^{(1)} \cdot \delta_j^{(1)} \cdot x_2}$$

  - So if x_2 is mostly zero, it will not get updated adequately.

Adaptive Gradient AdaGrad (2)

- One idea is therefore to change the learning rate associated with a feature according to the frequency of that feature.
  - Reduce if frequent, and increase if sparse.
- The algorithm is called AdaGrad (Adaptive Gradient):

$$v(t+1) = v(t) + \left(\frac{\partial(L)}{\partial(w)}\right)^2$$

$$w(t+1) = w(t) - \frac{\eta}{\sqrt{v(n+1) + \varepsilon}} \cdot \frac{\partial(L)}{\partial(w)}$$

RMS Prop (1)

- AdaGrad reduces the learning rate too aggressively, so after a while, the weight updates will be very small.
- To fix this, we should decay the denominator as well.

$$v(t+1) = \beta v(t) + (1-\beta)\left(\frac{\partial(L)}{\partial(w)}\right)^2$$

$$w(t+1) = w(t) - \frac{\eta}{\sqrt{v(t+1)+\varepsilon}} \cdot \frac{\partial(L)}{\partial(w)}$$

  - This is called RMS Prop (Root Mean Square Propagation).

Adam (1)

- Adam stands for Adaptive Moment Estimation, which combines the properties of Momentum (average of past gradients) and RMS Prop (decaying learning rate with decaying denominator):

$$m(t+1) = \beta_m m(t) + (1-\beta_m)\left(\frac{\partial(L)}{\partial(w)}\right)$$

$$v(t+1) = \beta_v v(t) + (1-\beta_v)\left(\frac{\partial(L)}{\partial(w)}\right)^2$$

$$w(t+1) = w(t) - \frac{\eta}{\sqrt{v(t+1)+\varepsilon}} \cdot m(t+1)$$

Adam (2)

- There is a problem.
  - We might have initialized $v(0) = 0$.
  - Assuming β_2 is a "large" number close to one, e.g. 0.999. Then:

$$v(1) = \beta_v \underbrace{v(0)}_{0} + \left(\underbrace{1 - \beta_v}_{0.001}\right)\left(\frac{\partial(L)}{\partial(w)}\right)^2 = \text{a small number}$$

$$w(1) = w(0) - \frac{\eta}{\sqrt{\text{a small number} + \varepsilon}} \cdot m(1)$$

  - The gradient step becomes very big and could lead to bad results.

Adam (3)

- Therefore, the actual Adam algorithm contains some bias correction terms:

$$m(t+1) = \beta_m m(t) + (1-\beta_m)\left(\frac{\partial(L)}{\partial(w)}\right), \quad \hat{m}(t+1) = \frac{m(t+1)}{1-\beta_m^{t+1}}$$

$$v(t+1) = \beta_v v(t) + (1-\beta_v)\left(\frac{\partial(L)}{\partial(w)}\right)^2, \quad \hat{v}(t+1) = \frac{v(t+1)}{1-\beta_v^{t+1}}$$

$$w(t+1) = w(n) - \frac{\eta}{\sqrt{\hat{v}(t+1)+\varepsilon}} \cdot \hat{m}(t+1)$$

- Adam is a good choice of optimizer for many problems!

Second Order Methods (1)

- What we have seen so far are first-order methods.
- To speed up convergence, second-order methods, using second-order derivatives have been proposed.
- Consider the second-order Taylor series of the loss function:

$$L\big(w(t+1)\big) = L\big(w(t) + \Delta w(t)\big) \approx L\big(w(t)\big) + g^T(t)\Delta w(t) + \frac{1}{2}\Delta w^T(t)H(t)\Delta w(t)$$

  - Where $g^T(t) = \frac{\partial L(w)}{\partial w}$ and $H(t) = \frac{\partial^2 L(w)}{\partial w^2}$ are the gradient vector and Hessian matrix respectively.

Second Order Methods (2)

- We want $L\big(w(t+1)\big)$ to be minimized, i.e.

$$\boxed{\frac{\partial L\big(w(t+1)\big)}{\partial \Delta w(t)} = g^T(t) + H(t)\Delta w(t) = 0} \rightarrow \boxed{\Delta w(t) = -H^{-1}(t)g^T(t)}$$

  - This is call the Newton method.
  - The convergence is faster than simple steepest descent.
  - However, the computational cost is high!
  - Can we avoid directly computing H?

Second Order Methods (3)

- There are several algorithms available:
  - Newton and Gauss-Newton Algorithms
  - Levenberg-Marquardt Algorithms
  - Quasi-Newton Algorithms
  - Conjugate Gradient Algorithms
- The best method depends on the problem to be solved.
- Good news: most of these algorithms are available in MATLAB and the codes will be shown shortly.

Content

- Revision: Optimization
- Backpropagation
- Generalization
- Discussions
- MATLAB NN Toolbox Example

Regression (1)

```
%% the training data input x [-1, -0.95, -0.9, ..., 0.95, 1]

x=-1:0.05:1; % Note: need to put as row vector
len = length(x);
```

```
%% the training data output y, added with noise

y=0.8*x.^3 + 0.3 * x.^2 -0.4 * x + normrnd(0,0.02,[1,len]); % Note: need to put as row vector
figure,plot(x,y,'k+')
```

```
%% specify the structure and learning algorithm for MLP

net=newff(minmax(x),[3,1],{'tansig','purelin'},'trainlm');
net.trainparam.show=2000; % epochs between display
net.trainparam.lr=0.01; % learning rate
net.trainparam.epochs=10000; % maximum epochs to train
net.trainparam.goal=1e-4; % performance goal, training will stop if this is reached
```

Regression (2)

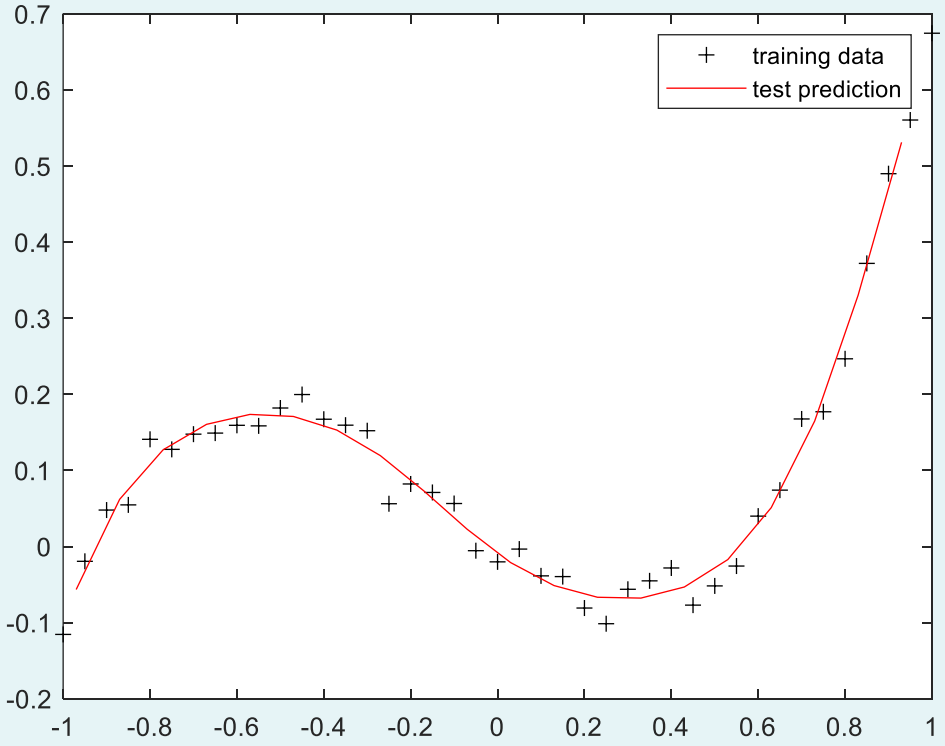
```
%% Train the MLP %

[net,tr]=train(net,x,y);
```

```
%% Test the MLP at different xtest

xtest=-0.97:0.1:0.93; % Data is -0.97, -0.87, ..., 0.83, 0.93 (never seen before)
net_output=sim(net,xtest); % Output of network at given xtest
```

```
%% Plot out the test results

hold on;
plot(xtest,net_output,'r-');
legend('training data','test prediction')
```

Regression (3)

- Result:
  - Good generalization!



Regression (4)

Levenberg-Marquardt method

- About the code:

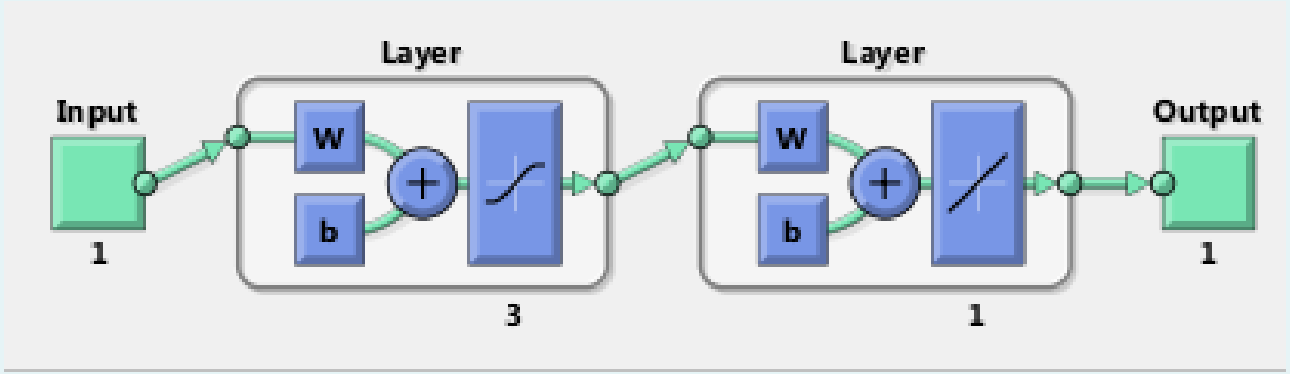
```
net=newff(minmax(x),[3,1],{'tansig','purelin'},'trainlm');
```

3 nodes in hidden layer

1 node in output layer

Hidden layer uses tanh

Output layer uses linear



Regression (5)

- We can try other structures:

Levenberg-Marquardt method

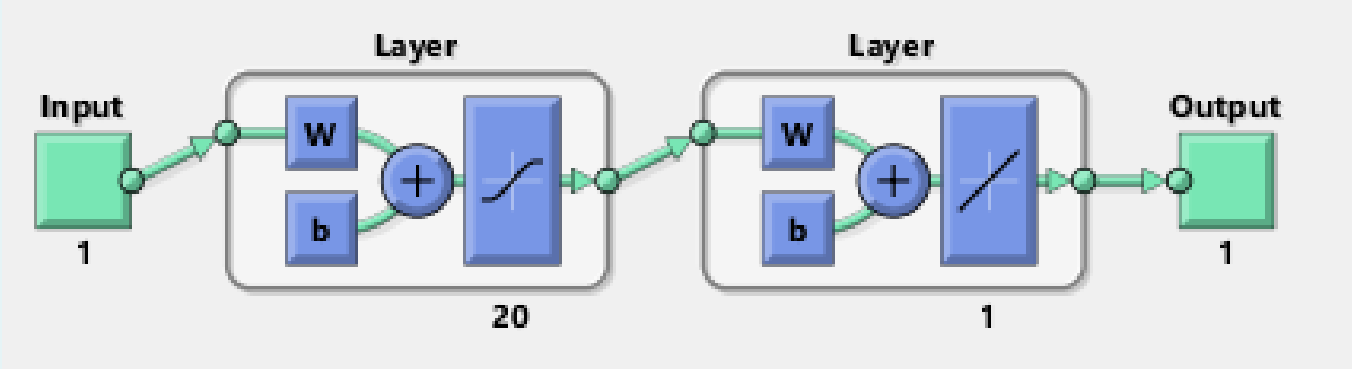
```
net=newff(minmax(x),[20,1],{'tansig','purelin'},'trainlm');
```

20 nodes in hidden layer

1 node in output layer

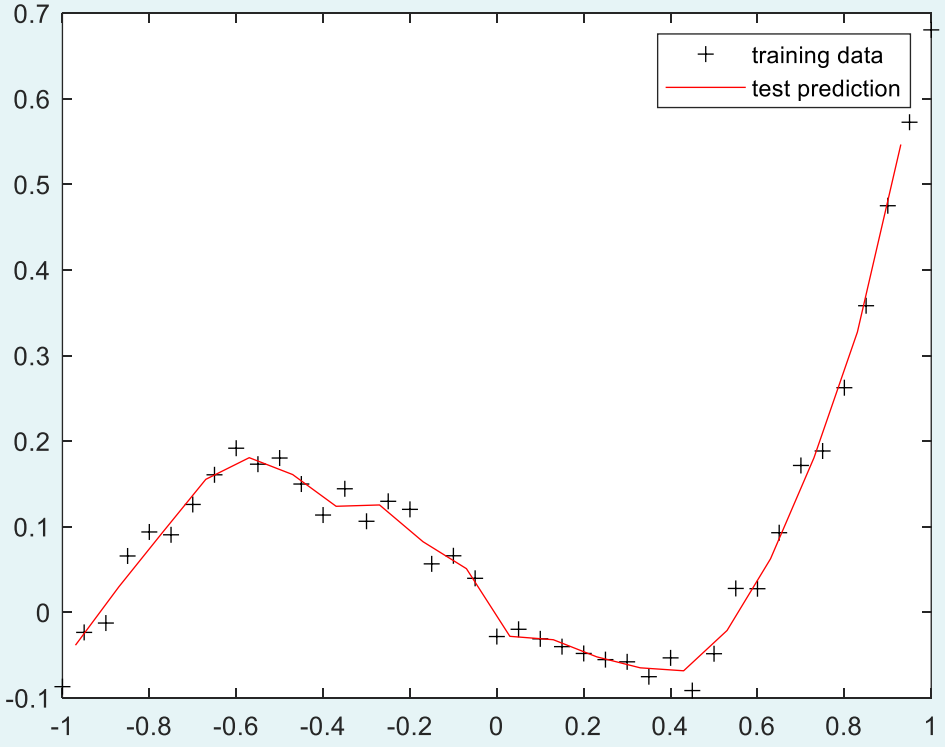
Hidden layer uses tanh

Output layer uses linear



Regression (6)

- Result:
  - Some overfitting!



Regression (7)

Bayesian regularization method

- Use regularization:

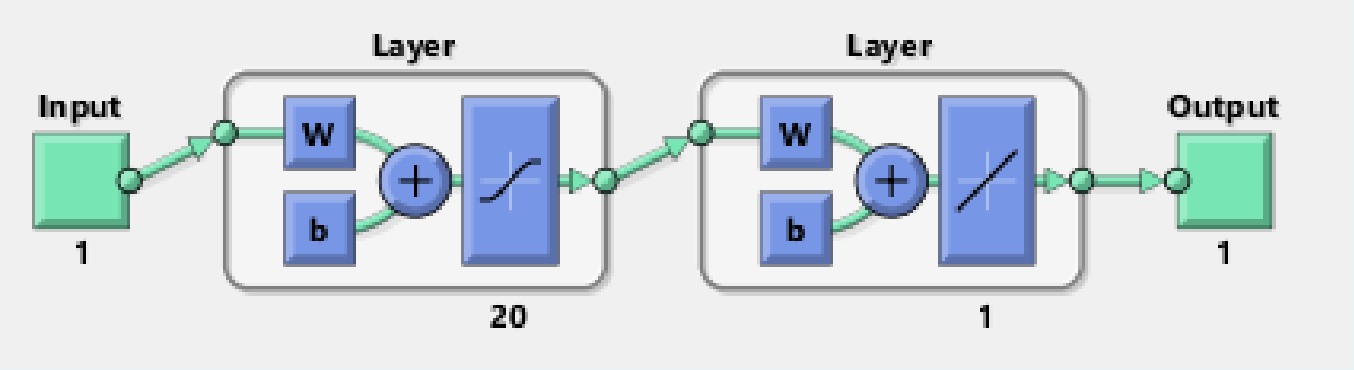
```
net=newff(minmax(x),[20,1],{'tansig','purelin'},'trainbr');
```

20 nodes in hidden layer

1 node in output layer

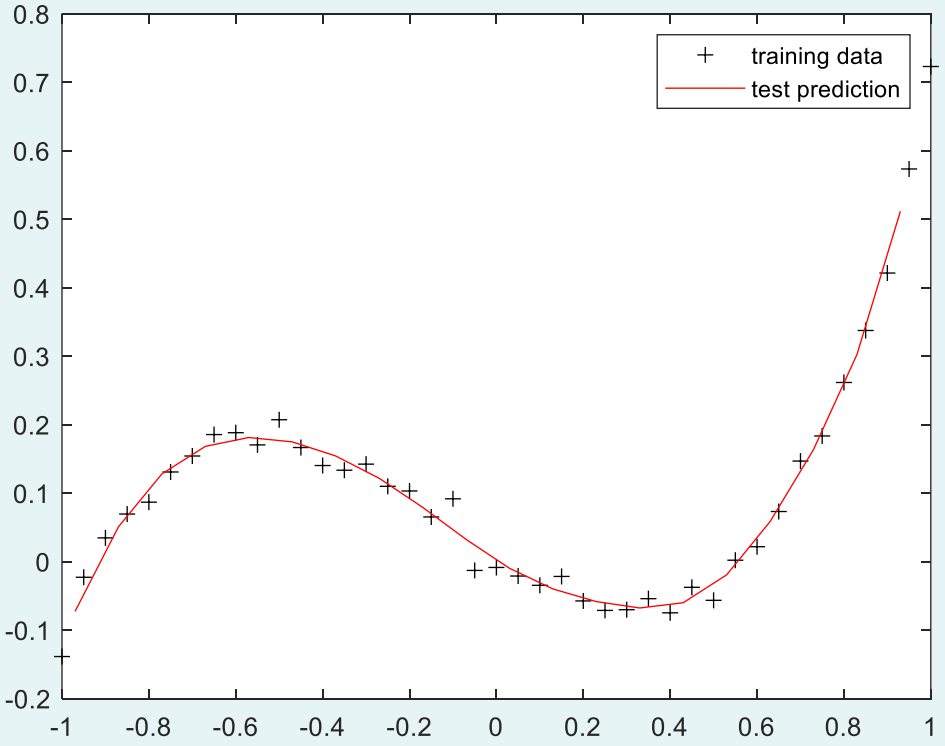
Hidden layer uses tanh

Output layer uses linear



Regression (8)

- Result:
  - Improved with regularization!



Regression (9)

- More layers:

2nd hidden layer uses log

Levenberg-Marquardt method

```
net=newff(minmax(x),[5,3,1],{'logsig','tansig','purelin'},'trainlm');
```

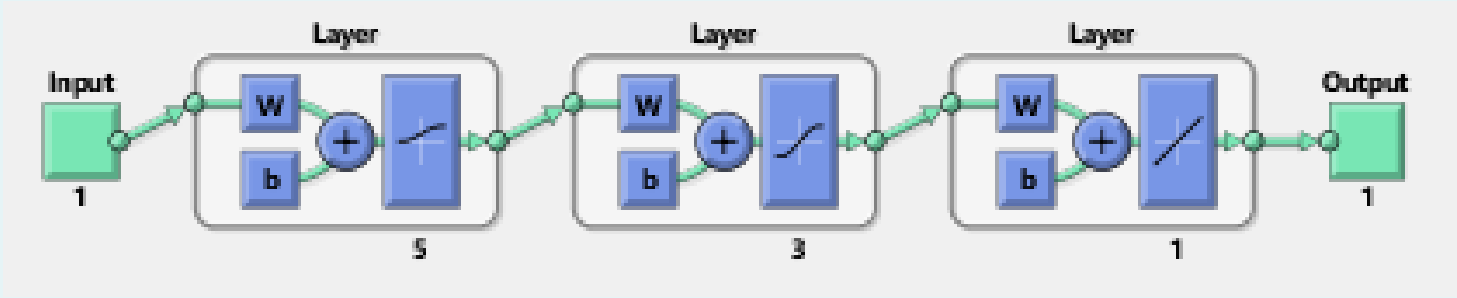
5 nodes in 1st hidden layer

3 nodes in 2nd hidden layer

1 node in output layer

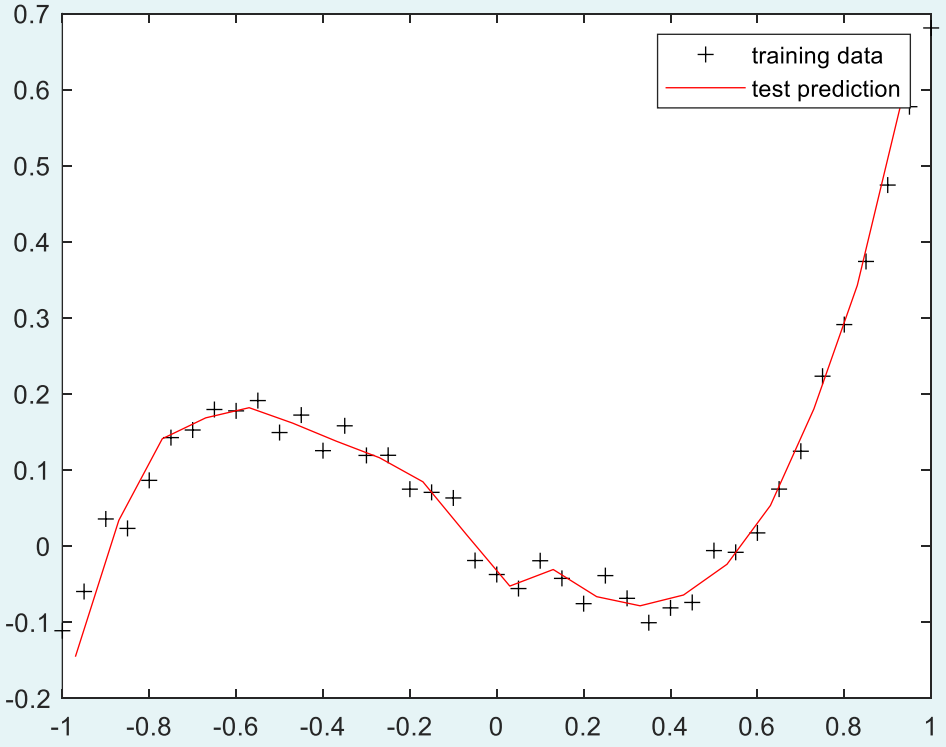
2nd hidden layer uses tanh

Output layer uses linear



Regression (10)

- Result:
  - Some overfitting!



Regression (11)

- Use regularization:

2nd hidden layer uses log

Bayesian regularization method

```
net=newff(minmax(x),[5,3,1],{'logsig','tansig','purelin'},'trainbr');
```

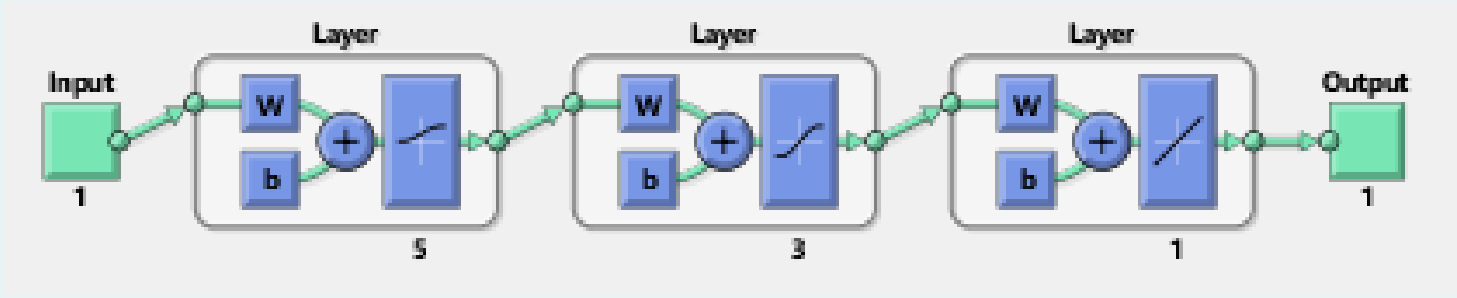
5 nodes in 1st hidden layer

3 nodes in 2nd hidden layer

1 node in output layer

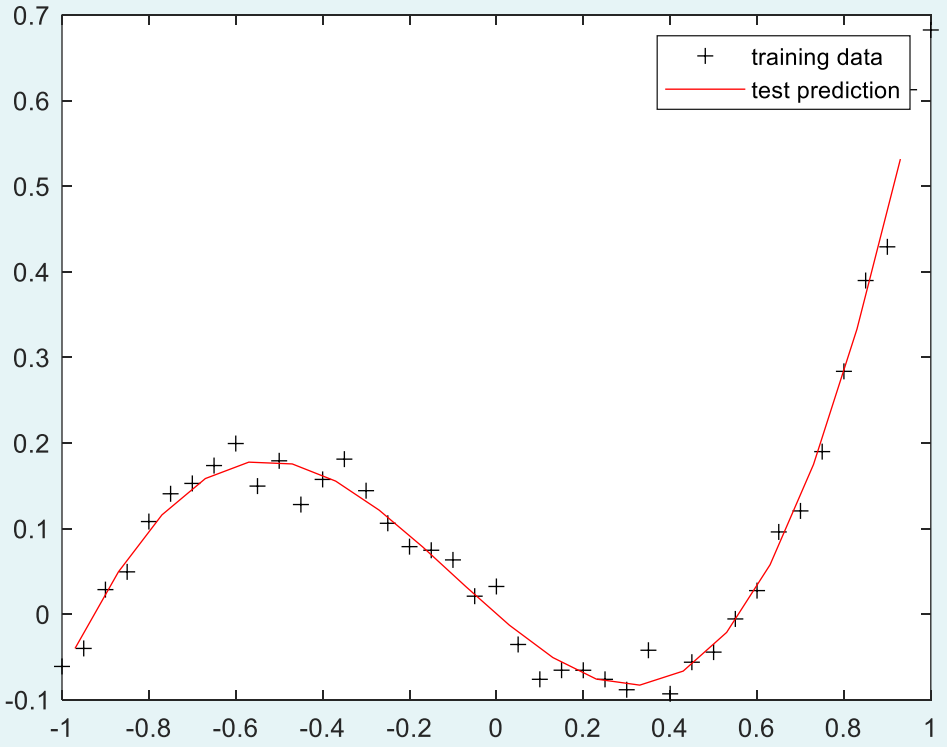
2nd hidden layer uses tanh

Output layer uses linear



Regression (12)

- Result:
  - Improved after regularization!



Regression (13)

- As you can see, different network structures (# hidden layers, # nodes) and different training algorithm gives different restuls.
  - Some trial and error is required.
  - Start small! – If the problem can be solved using a small network, then there is no need to use a large network.
    - In the given example, the 1-3-1 works very well!

MATLAB Functions (1)

- The activation functions in MATLAB are as follows:
  - Linear – purelin (pure linear)
  - ReLU – poslin (positive linear)
  - Log sigmoid – logsig
  - Tanh – tansig

MATLAB Functions (2)

- Training methods:
  - "Traingd" is gradient descent.
  - "Traingdm" is gradient descent with momentum.
  - "Traindga" is gradient descent with adaptive learning rate.
  - "Traingdx" is gradient descent with momentum and adaptive learning rate.

MATLAB Functions (3)

- "Trainlm" (Levenberg-Marquardt) is good for small number of weights.
  - Memory requirement is proportional to the square of the number of weights.
- "Trainbr" (Bayesian regularization) produces a network which generalizes well.
- "Traincgf" is conjugate-gradient method, which is suitable for large number of weights.
  - Memory requirement is proportional to the number of weights.
  - If you run into memory problems when using "trainlm", then you can try changing to "traincgf".

Classification (1)

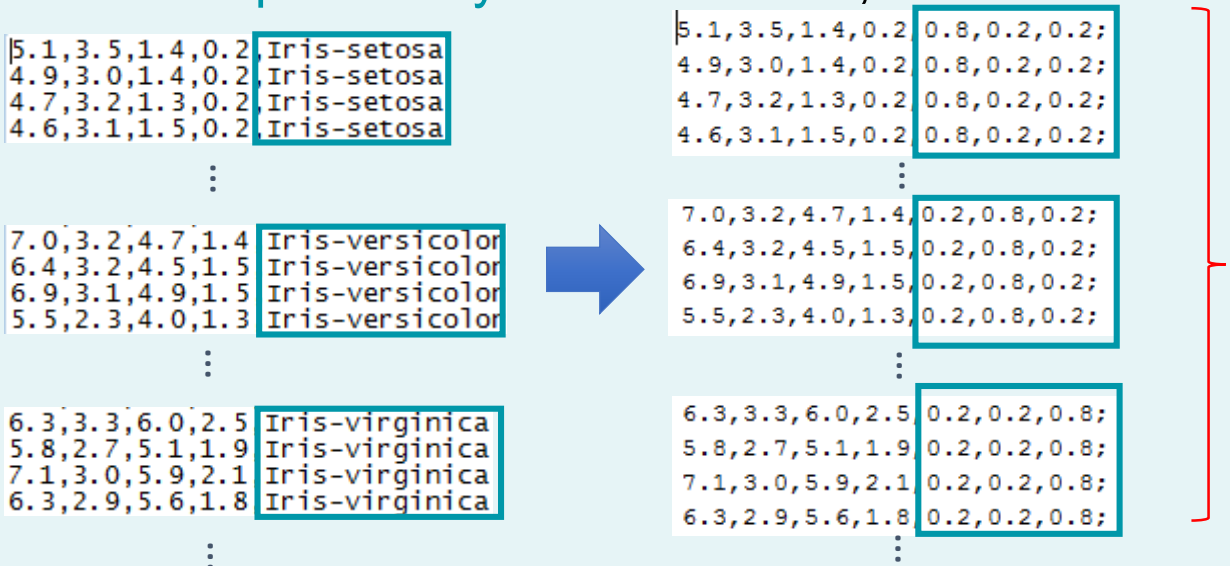
- Sample datasets can be found in University of California at Irvine's Machine Learning Repository: https://archive.ics.uci.edu/ml/index.php
  - E.g. Iris Dataset.
    - Based on:
      - Sepal length in cm
      - Sepal width in cm
      - Petal length in cm
      - Petal width in cm
    - Classify the iris into different types:
      - Iris Setosa / Iris Versicolour / Iris Virginica

https://en.wikipedia.org/wiki/Iris_setosa

https://en.wikipedia.org/wiki/Iris_virginica

Classification (2)

- Preparation:
  - The original data are shown on the left.
  - We replace the three types of iris using three numbers representing the "probability" of the classes, i.e.



As mentioned, avoid 1 and 0 because these numbers can only be reached asymptotically by logistic function. The weights are driven to be larger and larger.

Classification (3)

- Randomise Order:
  - The original data was ordered based on the types of iris.
  - We jumble up the data to achieve a random order.

	1	2	3	4	5	6	7
1	5.1000	3.5000	1.4000	0.2000	0.8000	0.2000	0.2000
2	4.9000	3	1.4000	0.2000	0.8000	0.2000	0.2000
3	4.7000	3.2000	1.3000	0.2000	0.8000	0.2000	0.2000
4	4.6000	3.1000	1.5000	0.2000	0.8000	0.2000	0.2000
5	5	3.6000	1.4000	0.2000	0.8000	0.2000	0.2000

	1	2	3	4	5	6	7
1	5.4000	3	4.5000	1.5000	0.2000	0.8000	0.2000
2	7.2000	3	5.8000	1.6000	0.2000	0.2000	0.8000
3	7.7000	3	6.1000	2.3000	0.2000	0.2000	0.8000
4	5.2000	2.7000	3.9000	1.4000	0.2000	0.8000	0.2000
5	5.5000	2.3000	4	1.3000	0.2000	0.8000	0.2000

Classification (4)

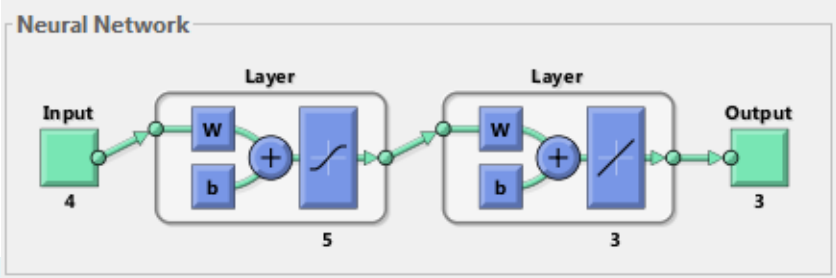
- Finally, we choose the first 70% of the randomised data as training set, and the remaining 30% as validation set.
- The codes up to here are:

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Import Data and Randomise Order %
% Choose 70% as training set      %
% And remaining 30% as test set   %
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

IrisData;

[n,m] = size(IrisAttributesAndTypes);
i = randperm(n);
IrisDataJumbled = IrisAttributesAndTypes(i,:);

SeventyPercent = round(0.7*n,0);
IrisDataTrain = IrisDataJumbled(1:SeventyPercent,:);
IrisDataTest = IrisDataJumbled(SeventyPercent+1:n,:);

x = IrisDataTrain(:,1:4)';
y = IrisDataTrain(:,5:7)';
```

Classification (5)

- The next part is to set up the network architecture and train the network:

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% specify the structure and learning algorithm for MLP %
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

net=newff(minmax(x),[5,3],{'tansig','purelin'},'trainlm');
net.trainparam.show=2000; % epochs between display
net.trainparam.lr=0.01; % learning rate
net.trainparam.epochs=10000; % maximum epochs to train
net.trainparam.goal=1e-4; % performance goal, training will stop if this is reached

%%%%%%%%%%%%%%%%%%
% Train the MLP %
%%%%%%%%%%%%%%%%%%

[net,tr]=train(net,x,y);
```



Classification (6)

- Finally, we test the network using the validation data.

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Test the MLP, net_output is the output of the MLP, ytest is the desired output. %
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

xtest = IrisDataTest(:,1:4)';
ytest = IrisDataTest(:,5:7)';
net_output=sim(net,xtest);

%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Plot out the test results %
%%%%%%%%%%%%%%%%%%%%%%%%%%%%

[p,q] = size(ytest);
for j = 1:q
    [val_Actual(j),idx_Actual(j)] = max(ytest(:,j)); % Switch back from 3 numbers to specific class
    [val_Predicted(j),idx_Predicted(j)] = max(net_output(:,j));
end
```

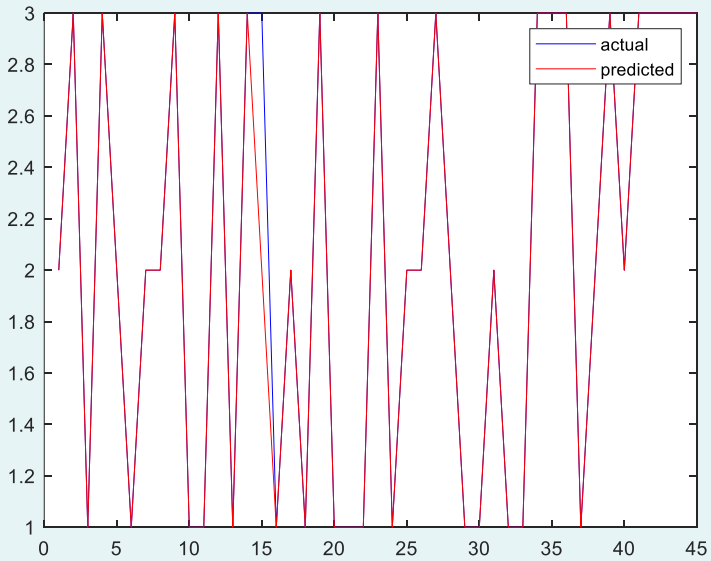
Classification (7)

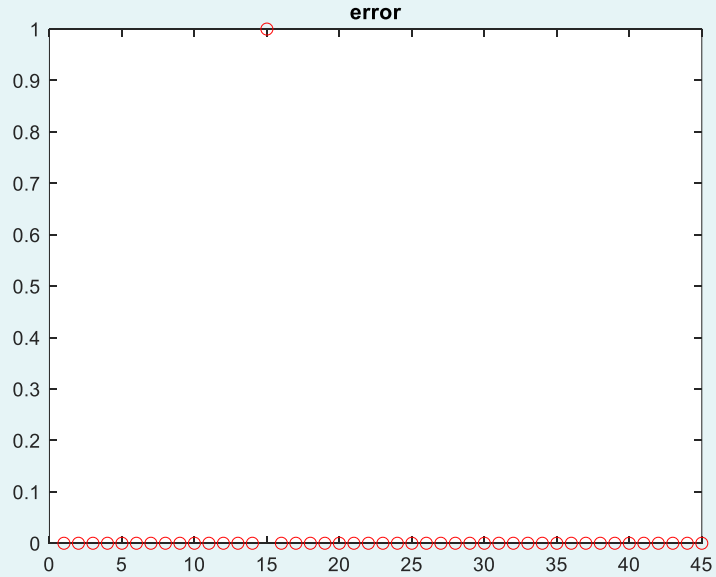
- Plot the classification results & errors:

```
plot(idx_Actual,'b-');
hold on;
plot(idx_Predicted,'r-');
hold off
legend('actual','predicted')

figure,plot(abs(idx_Actual-idx_Predicted),'ro')
title('error')
```

Classification (8)

- The test results are mostly correct, with only one misclassifications.





Exercise

- Download other datasets from University of California at Irvine's Machine Learning Repository: https://archive.ics.uci.edu/ml/index.php
- Use MATLAB's neural network toolbox to perform data classification on the data.

Thank you for your attention!

Any questions?