CSCI3230
Introduction to Neural Network II

Antonio Sze-To

Week 11, Fall 2013

Outline

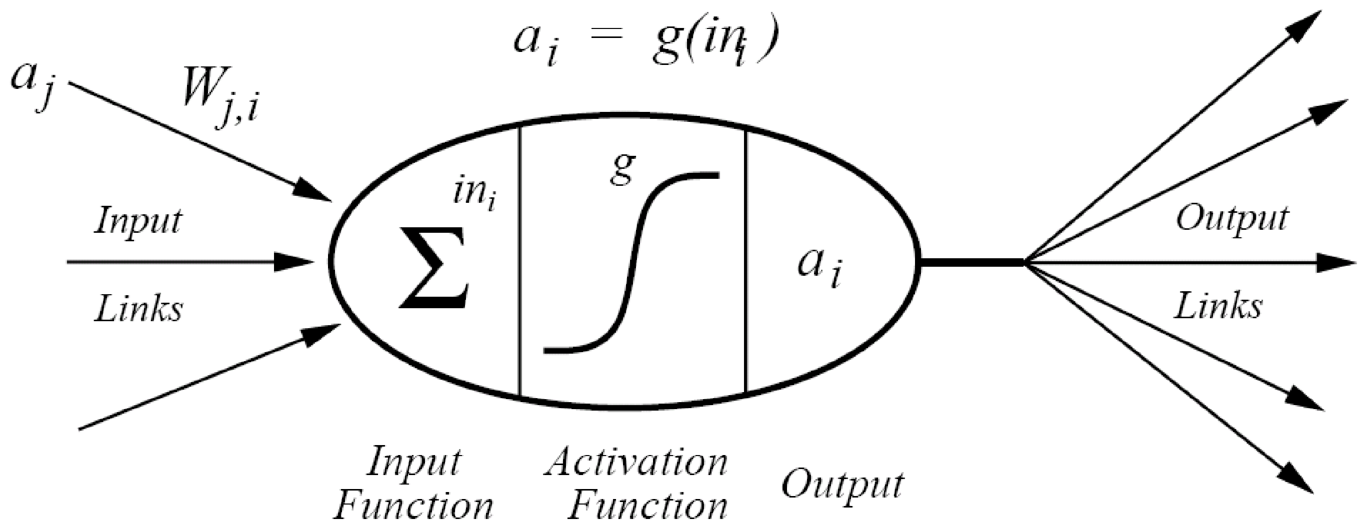
1. Review on Basics
2. Backward Propagation Algorithm
   1. Feed-forward property
   2. General Learning Principle
   3. Optimization Model & Gradient Descent
   4. Backward Propagation
3. Practical Issues
   1. Over-fitting
   2. Local Minima

Artificial Neuron (Perceptron)

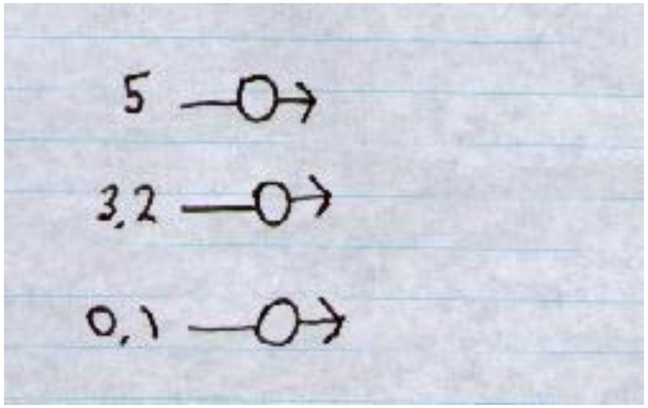
This is a Neuron i.

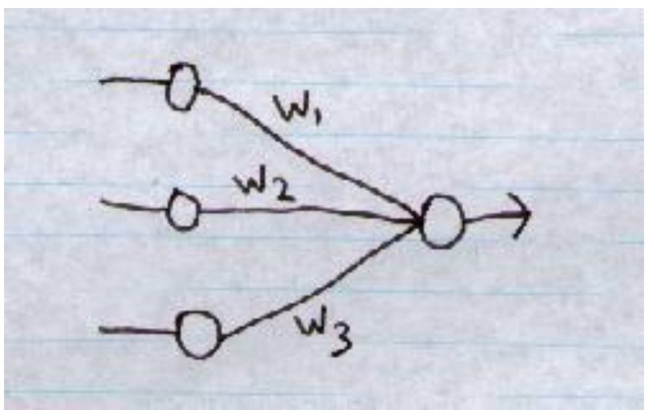
$in_i = \sum_j W_{j,i} a_j$, summing the input from the neurons in the previous layer

$$a_i = g(in_i) = g(\sum_j a_j W_{j,i})$$



Example





If $w_1 = 0.5$, $w_2 = -0.75$, $w_3 = 0.8$, and sigmoid function g is used as activation function, what is the output?

input $x = (I_1, I_2, I_3) = (5, 3.2, 0.1)$.

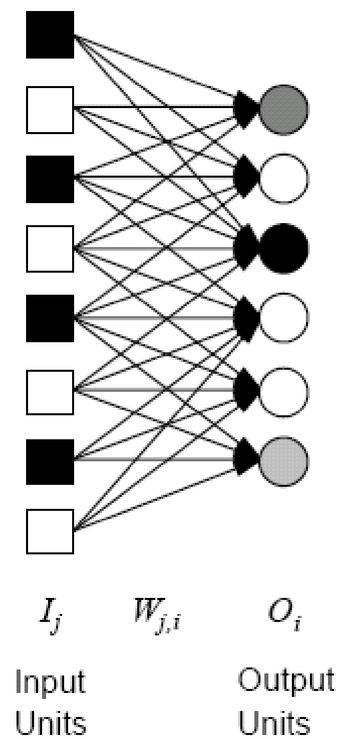
Summed input $= \sum_i w_i I_i = 5\,w_1 + 3.2\,w_2 + 0.1\,w_3$

Summed input = 5(0.5) + 3.2(-0.75) + 0.1(0.8) =
= 0.18

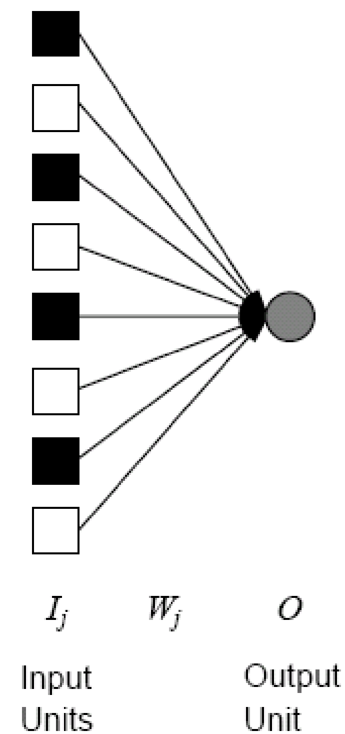
Output = g(Summed input) $= \frac{1}{1+e^{-0.18}} = 0.54488$

Single Perceptron & Single-Layer Perceptron

- Perceptron = Neuron
- Single-layer perceptron = single-layer neural network
- Multi-layer perceptron = multi-layer neural network
- The existence of **one or more hidden layer** is the difference between single-layer perceptron and multi-layer perceptron



Perceptron Network

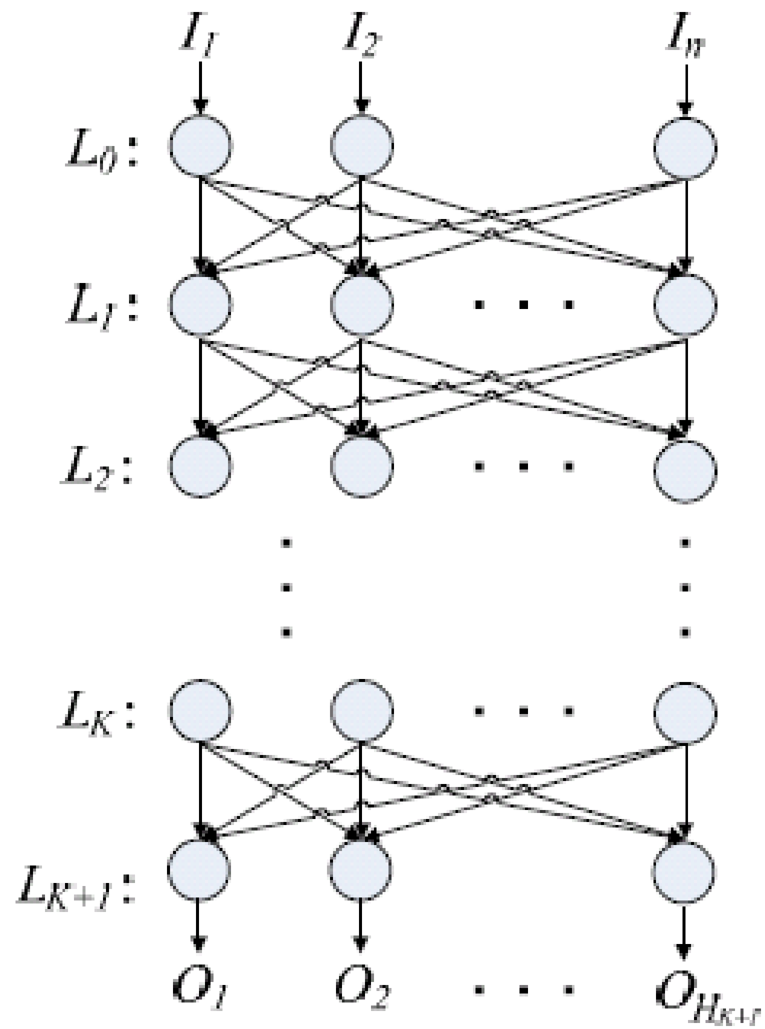


Single Perceptron

What are their limitations ?

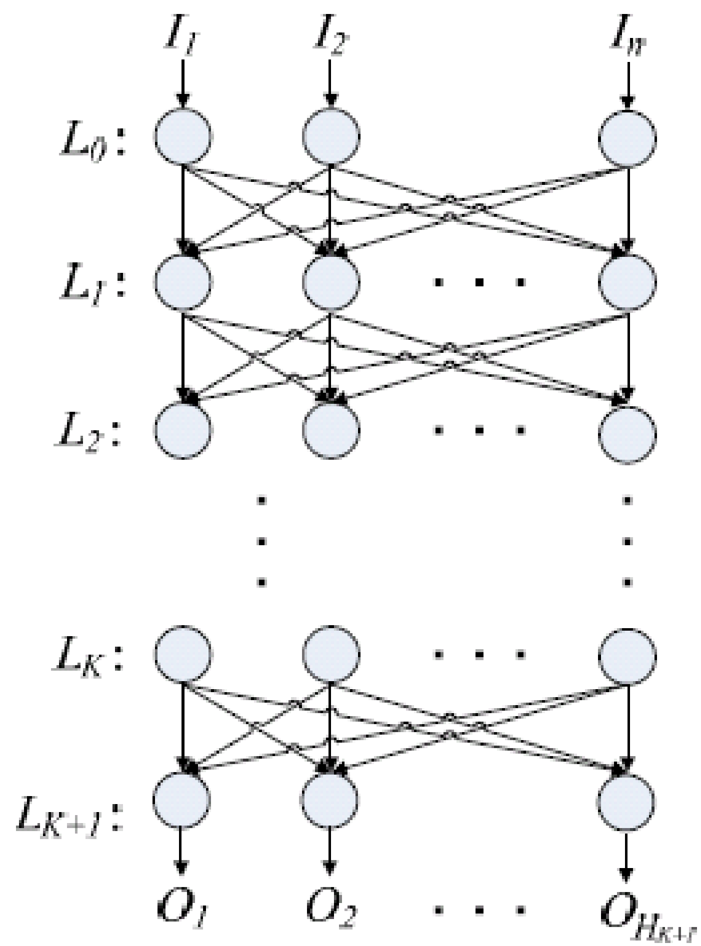
Multi-Layer Perceptron

- Multi-Layer
  - Input
  - Hidden layer(s)
  - Output layer
- Feed-forward
  - Links go one direction only



Feed forward property

- Given weights and inputs, outputs of neurons in L1 can be calculated.
- Outputs of neurons in L2 can be calculated and so on…
- Finally, outputs of neurons in the output layer can be calculated



Backward Propagation Algorithm

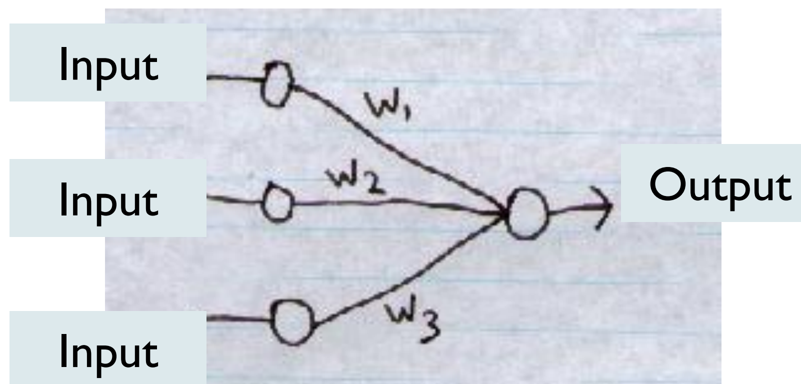
How to update weights to minimize the error between the target and output?

General Learning Principle

1. For supervised learning, we provide the model a set of inputs and targets.
2. The model returns the outputs
3. Reduce the difference between the outputs and targets by updating the **weights**
4. Repeat step 1-3 until some stopping criteria is encountered

Target   Input

	A	B	C	D	E	F
1	Y	X1	X2	X3	X4	X5
2	0.754146	0.762883	0.827033	0.340149	0.834167	0.145904
3	0.805558	0.159068	0.790367	0.50529	0.368874	0.287317
4	0.935603	0.196626	0.750472	0.005161	0.124383	0.338216
5	0.475093	0.941213	0.004147	0.920922	0.692663	0.75291
6	0.765624	0.780739	0.609544	0.796086	0.664215	0.733656
7	0.872094	0.677136	0.28072	0.418843	0.83341	0.01389
8	0.381704	0.270435	0.95286	0.561531	0.709781	0.90491
9	0.273056	0.462169	0.214569	0.378295	0.898127	0.058751
10	0.313691	0.668121	0.439105	0.399858	0.682646	0.906417
11	0.648147	0.353544	0.454941	0.442574	0.544956	0.748201
12	0.792175	0.163763	0.96012	0.585589	0.630684	0.107818



Optimization Model (Error minimization)

- Given observed data, we want to make the network give the same output as the observed target variable(s)

- For a given topology of network, (number of layers, number of neurons, *how* they are connected) we want to find **weights** to minimize

$$E = \frac{1}{2}\sum_{i}(O_i - T_i)^2$$

Gradient descent

- Activation function smooth ➔E(W) is a smooth function of the weights W
- Can use calculus!
- But difficult to analytically solve
- Use iterative approach
  - Gradient Descent

$$\frac{\partial E}{\partial W} = 0$$

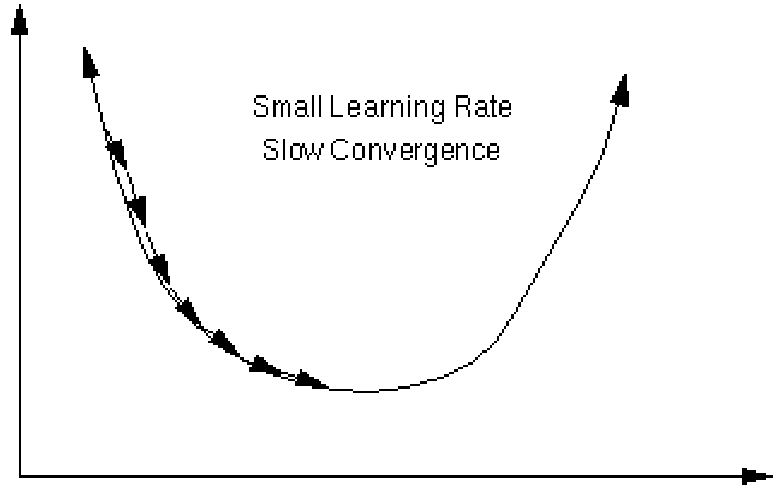
There may be many points with zero derivative. I.e. there may be many local optima.

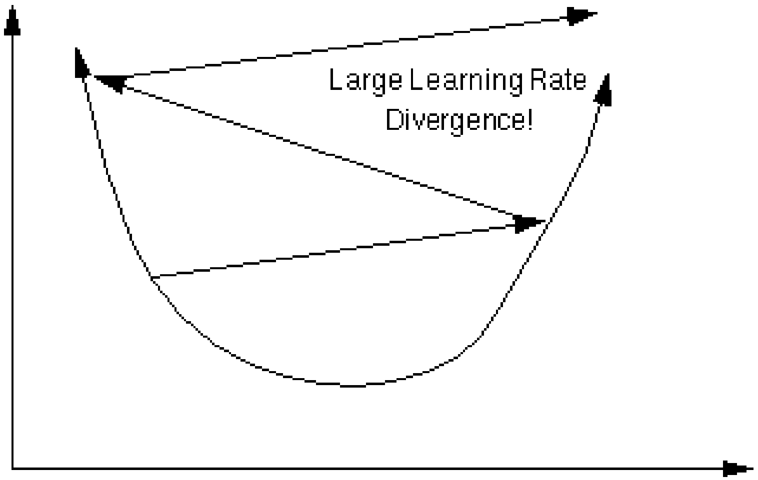
Gradient descent

For a smooth function $f(\vec{x})$,

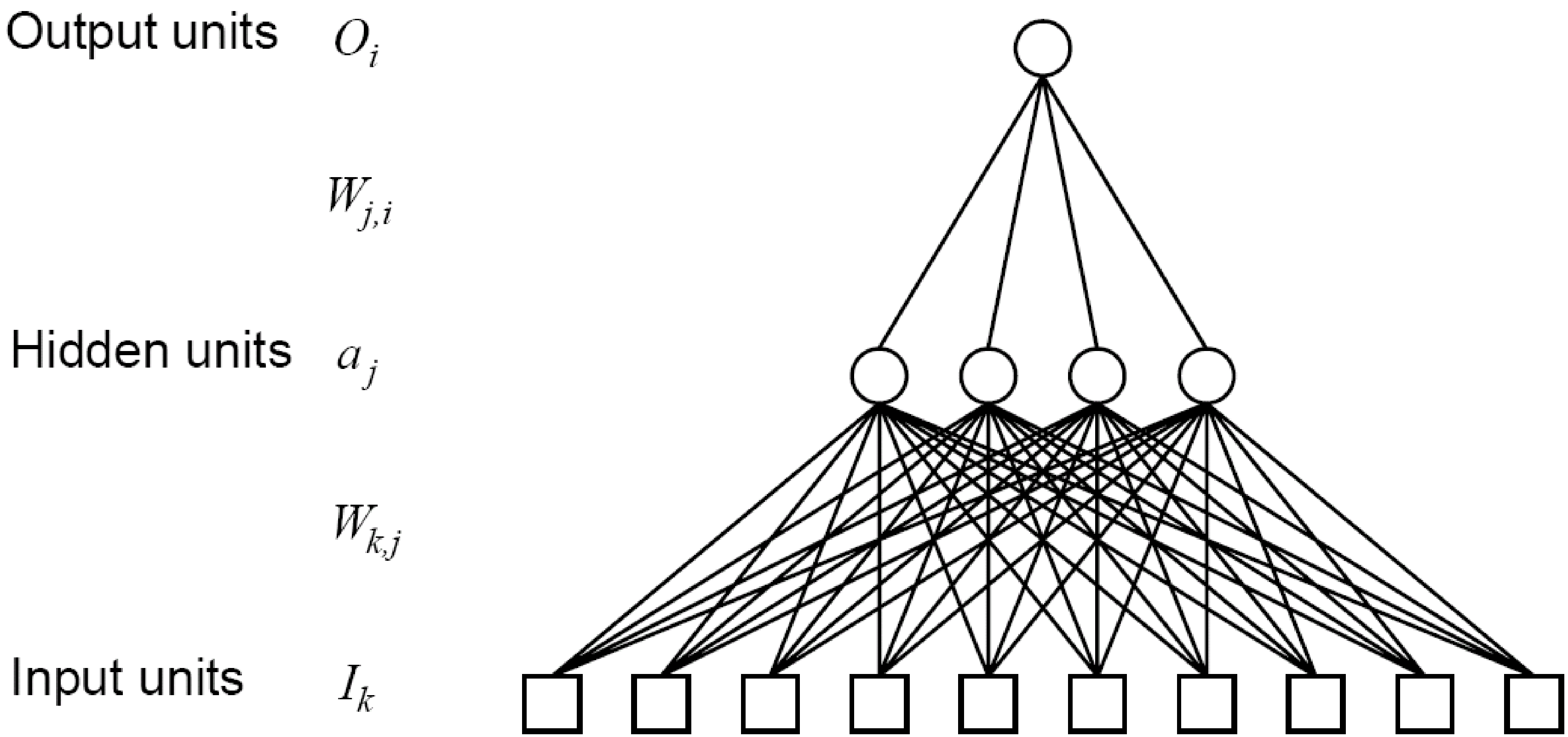
$-\dfrac{\partial f}{\partial \vec{x}}$ is the direction that f decreases most rapidly.

So $\vec{x}_{t+1} = \vec{x}_t - \eta \left.\dfrac{\partial f}{\partial \vec{x}}\right|_{\vec{x}=\vec{x}_t}$ until $\vec{x}$ converges

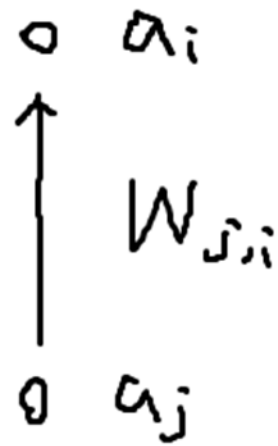




Weight Update Rules



Secret Formula



$$W_{j,i} \leftarrow W_{j,i} - \alpha \cdot \Delta_i \cdot a_j$$

Secret Formula

How to compute Δ for the output layer?

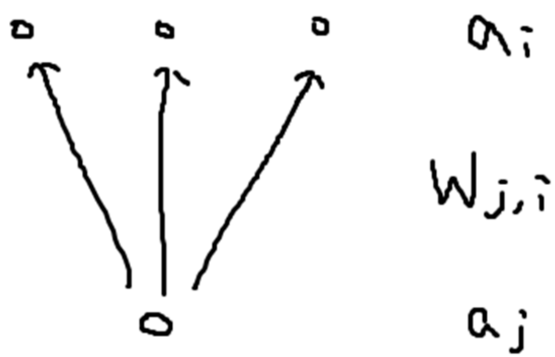
$$\Delta_i = (E_i - O_i) f'(In_i)$$

$$= err_i \cdot f(In_i) \cdot (1 - f(In_i))$$

$$= err_i \cdot O_i \cdot (1 - O_i)$$

Secret Formula

How to compute Δ for the hidden layers?

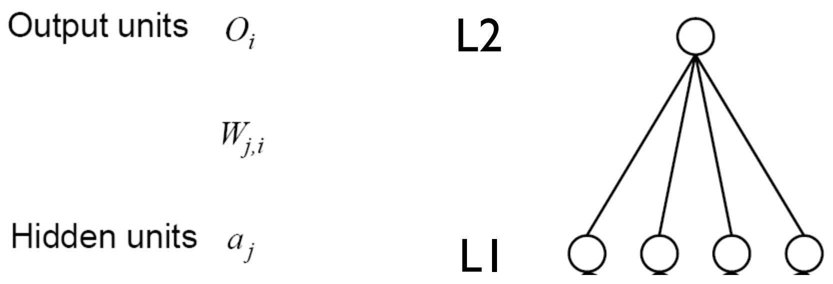


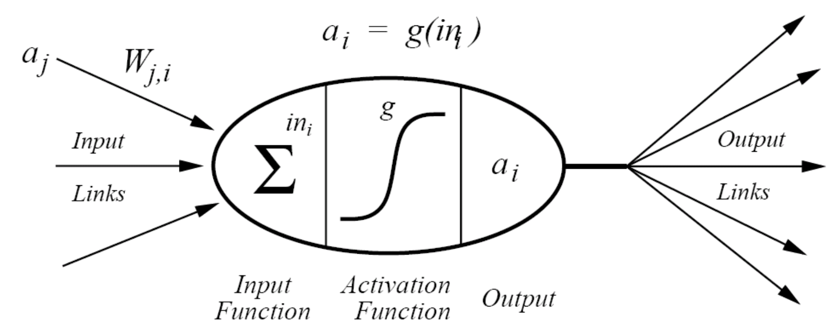
$$\begin{aligned}\Delta_j &= err_j \cdot f'(In_j)\\ &= \left(\sum_i W_{j,i}\Delta_i\right)(a_j)(1-a_j)\end{aligned}$$

Weight Update Rules (Output Layer)

- $\Delta_i = Err_i \times g'(in_i)$
- $W_{j,i} \leftarrow W_{j,i} - \alpha \times a_j \times \Delta_i$

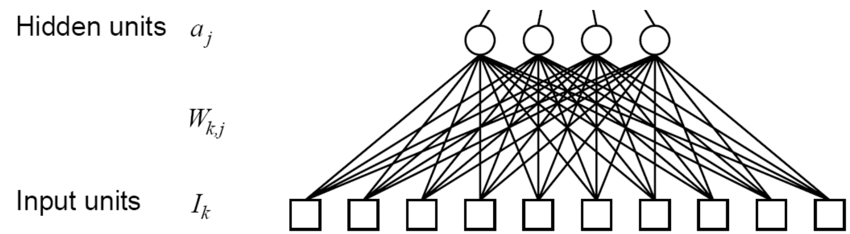
- $Err_i = O_i - T_i$
- $W_{j,i}$ is the weight between the j^{th} unit at the first hidden layer (L1) and the i^{th} unit at the output layer (L2)
- α is the learning rate
- a_j is the output of the j^{th} unit at the first hidden layer (L1)
- in_i is the total input to the i^{th} unit at the output layer (L2)
- $g'(in_i)$ is a value got by substituting $x = in_i$ into the first derivative of the activation function



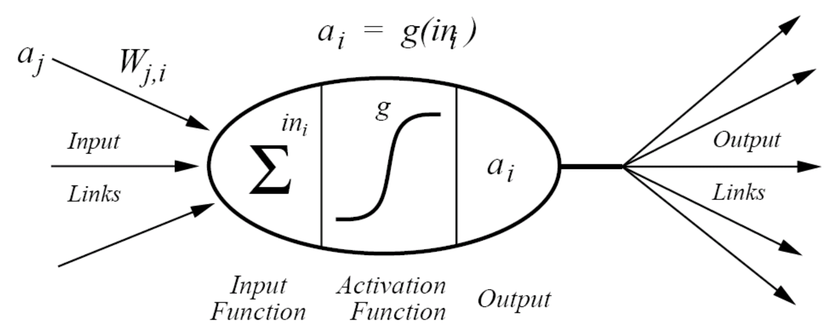


Weight Update Rules (Hidden Layer)

- When we update the weights connecting to the output layer, we use : ($\Delta_i = Err_i \times g'(in_i)$)
- $W_{j,i} \leftarrow W_{j,i} - \alpha \times a_j \times \Delta_i$

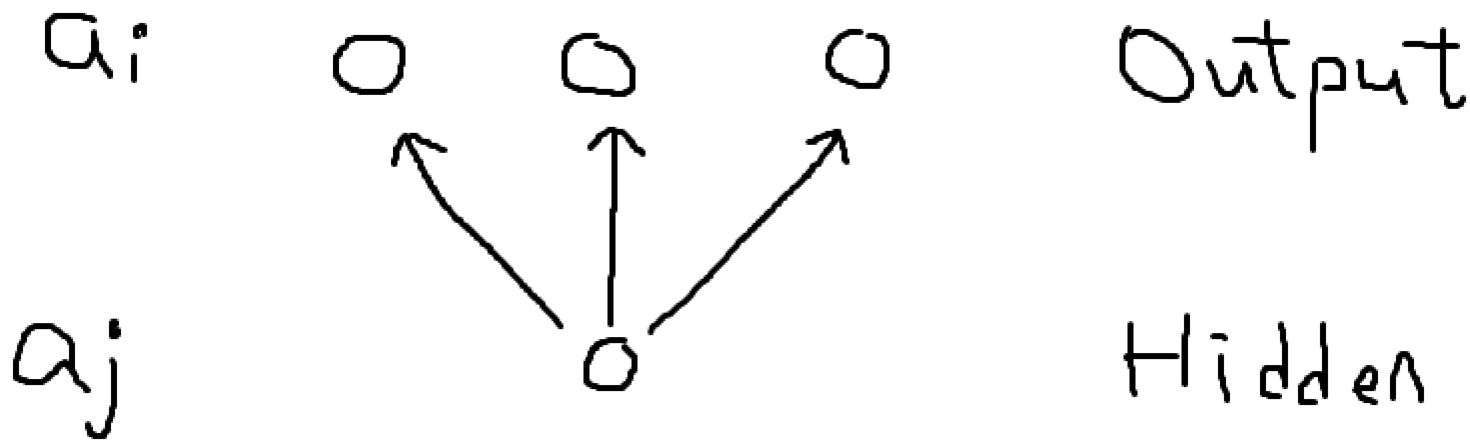


- Can we use something similar to update the weights connecting to the hidden layer ? i.e:
- $W_{k,j} \leftarrow W_{k,j} - \alpha \times a_k \times \Delta_j$
- $W_{k,j} \leftarrow W_{k,j} - \alpha \times a_k \times Err_j \times g'(in_j)$
- But what is Err_j ?



$$Err_j = \sum_i W_{j,i}\,\Delta_i$$

Error Back-Propagation

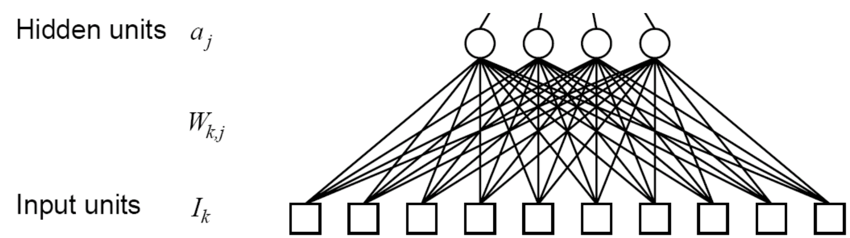


$$Err_j = \sum_i w_{j,i} \Delta_i$$

General Weight Update Rules

- To update weights connecting to the output layer, we use:
- $\Delta_i = Err_i \times g'(in_i)$
- $W_{j,i} \leftarrow W_{j,i} - \alpha \times a_j \times \Delta_i$

- To update weights connecting to the hidden layer, we use:
- $\Delta_j = \sum_i W_{j,i}\, \Delta_i \times g'(in_j)$
- $W_{k,j} \leftarrow W_{k,j} - \alpha \times a_k \times \Delta_j$



Pseudo Code

- **function** Back-Prop-Update(*network*, *examples*, α) **returns** a network with modified weights
- **inputs**: *network*, a multilayer network
- *examples*, a set of input/output pairs
- α, the learning rate
- **repeat**
- **for each** *e* **in** *examples* **do**
- /* Compute the output for this example */
- **O** ← Run-Network(*network*, $\mathbf{I^e}$)
- /* Compute the error and Δ for units in the output layer */
- $\mathbf{Err^e \leftarrow T^e - O}$
- /* Update the weights leading to the output layer */
- $\Delta_i \leftarrow g'(in_i)\, Err^e_i$
- $W_{j,i} \leftarrow W_{j,i} - \alpha \times a_j \times \Delta_i$
- **for each** subsequent layer **in** network **do**
- /* Compute the error at each node */
- $\Delta_j \leftarrow g'(in_j)\Sigma_i W_{j,i}\, \Delta_i$
- /* Update the weights leading into the layer */
- $W_{k,j} \leftarrow W_{k,j} - \alpha \times a_k \times \Delta_j$
- **end**
- **end**
- **until** network has converged
- **return** network

Feed-Forward

Back-propagate

Practical Issues

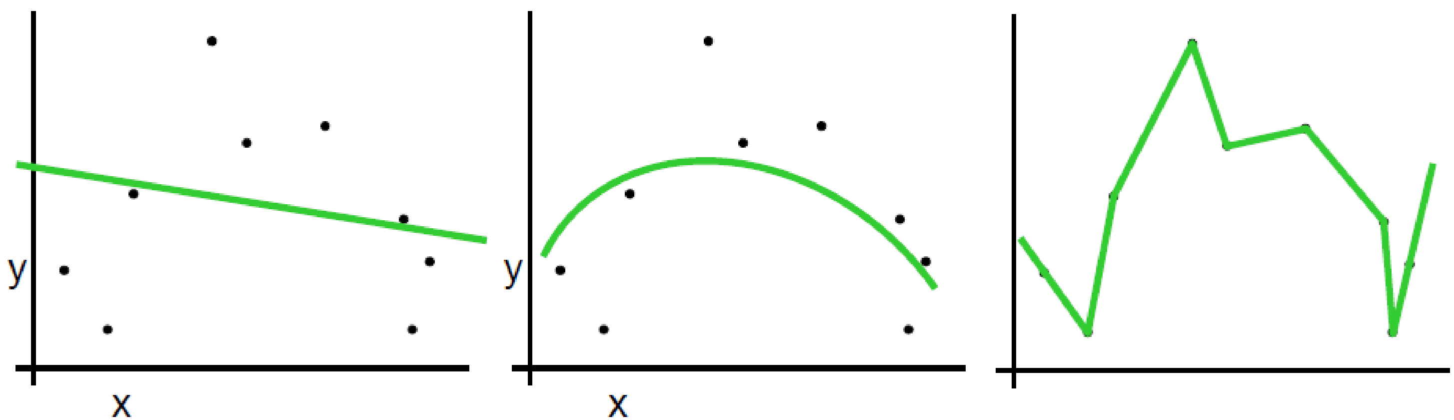
Over-fitting and Local Minima

Outline

1. Over-fitting
    1. Splitting
    2. Early stopping
    3. Cross-validation
2. Local Minima
    1. Randomize initial weights & Train multiple times
    2. Tune the learning rate appropriately

Over-fitting

Which is best?



Why not choose the method with the best fit to the data?

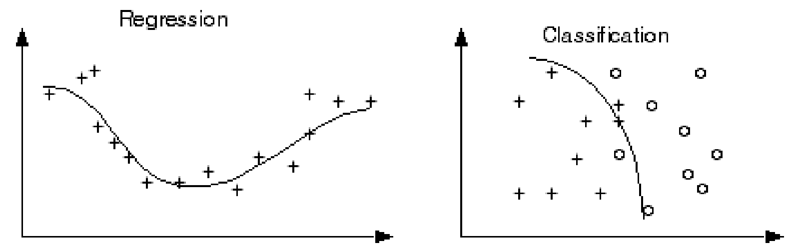
Preventing over-fitting

- Goal: To train a neural network having the **BEST** generalization power

1. When to stop the algorithm?
2. How to evaluate the neural network fairly?
3. How to select a neural network with the best generalization power?

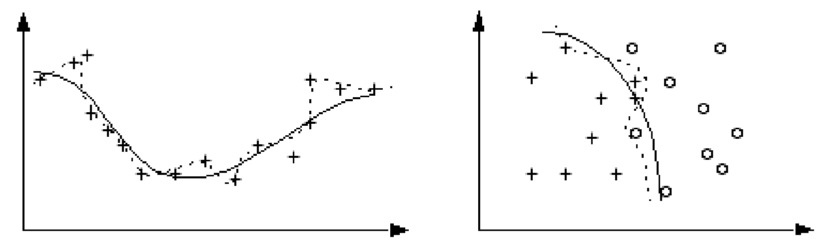
Over-fitting

- Motivation of learning
  - Build a model to learn the underlying knowledge
  - Apply the model to predict the unseen data
  - “Generalization”



Normal

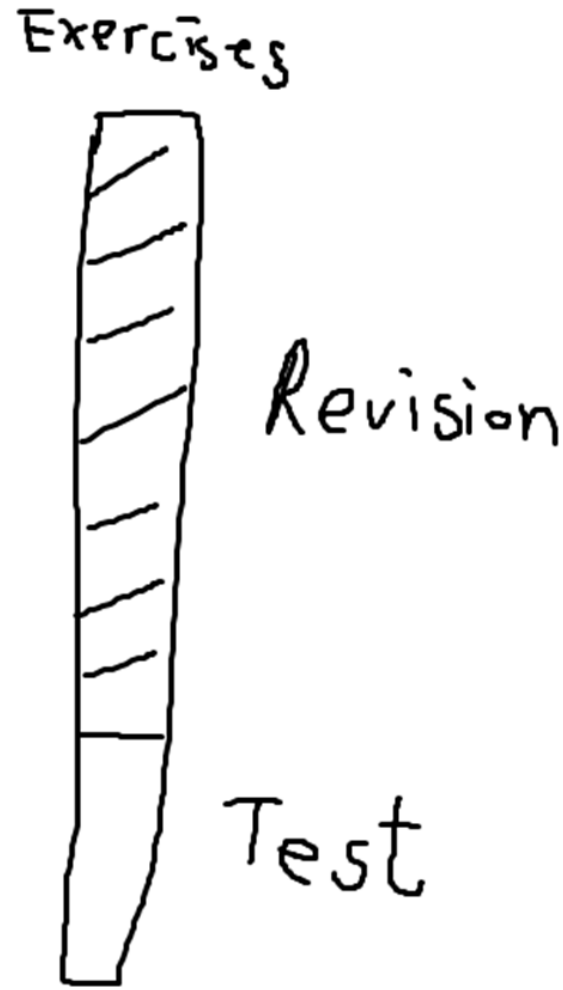
- What is over-fitting?
  - Perform well on the training data but perform poorly on the unseen data
  - Memorize the training data
  - “Specialization”



Over-fitted

Splitting method

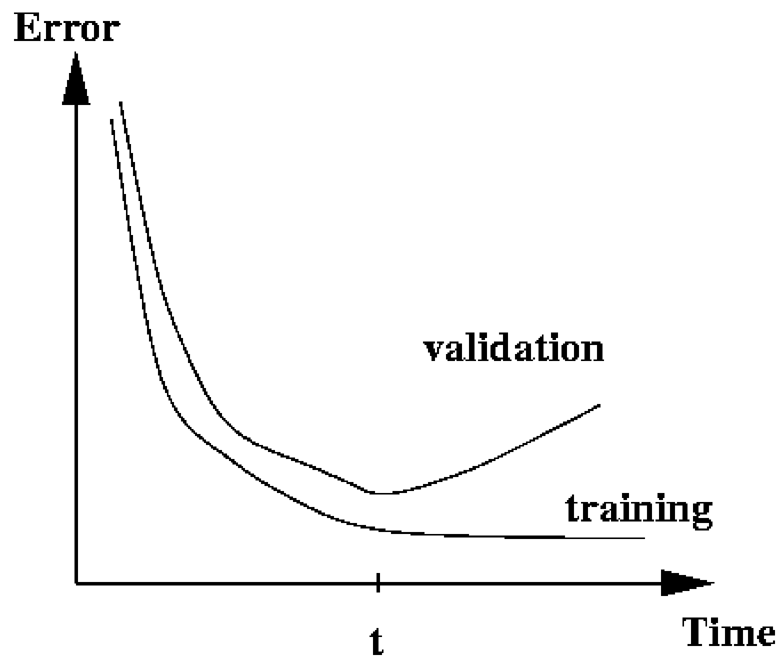
- Splitting method
  1. Random shuffle
  2. Divide data into two sets
     - Training set: 70% of data
     - Validation set: 30% of data
  3. Train (<u>update weights</u>) the network using the training set
  4. Evaluate (<u>do not update weights</u>) the network using validation set
  5. If stopping criteria is encountered, quit; else repeat step 3-4



Early stopping

- Any large enough neural networks may lead to over-fitting if over-trained.
- If we can know the time to stop training, we can prevent over-fitting

- **Early stopping**

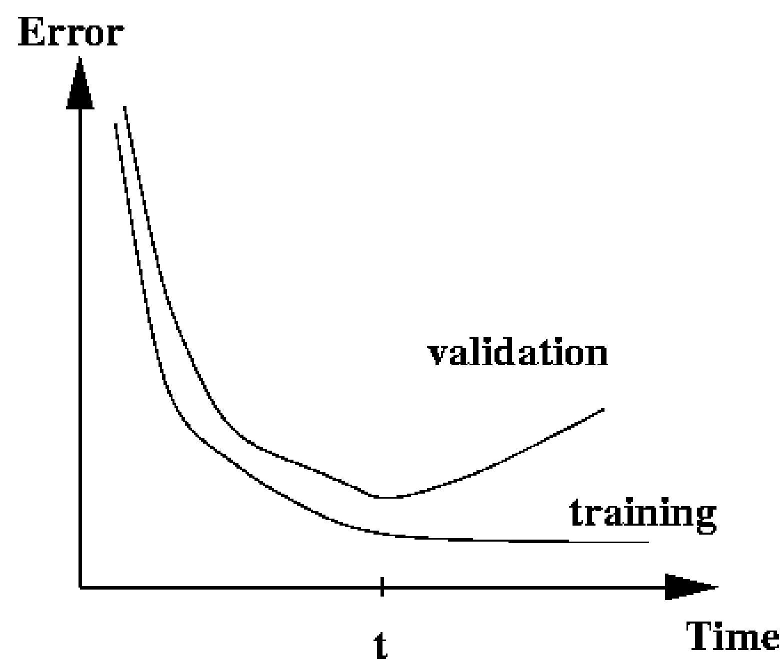


Early stopping

Early stopping

1. Validation error increases
2. Weights do not update much
3. Training epoch (maximum number of iterations) is larger than a constant defined by you

One epoch means all the data (including both training and validation) has been used <u>once</u>.



Early stopping

Splitting method

- However, there is a problem:

As only **<u>part</u>** of the data is used as testing data, there may be errors on the training methodology, which **<u>coincidentally</u>** cannot be reflected by the validation data.

Analogy:

A children knows only addition but not subtraction. However, as in the Maths quiz, no questions related to subtraction were asked, he still got 100% marks.

Cross-validation

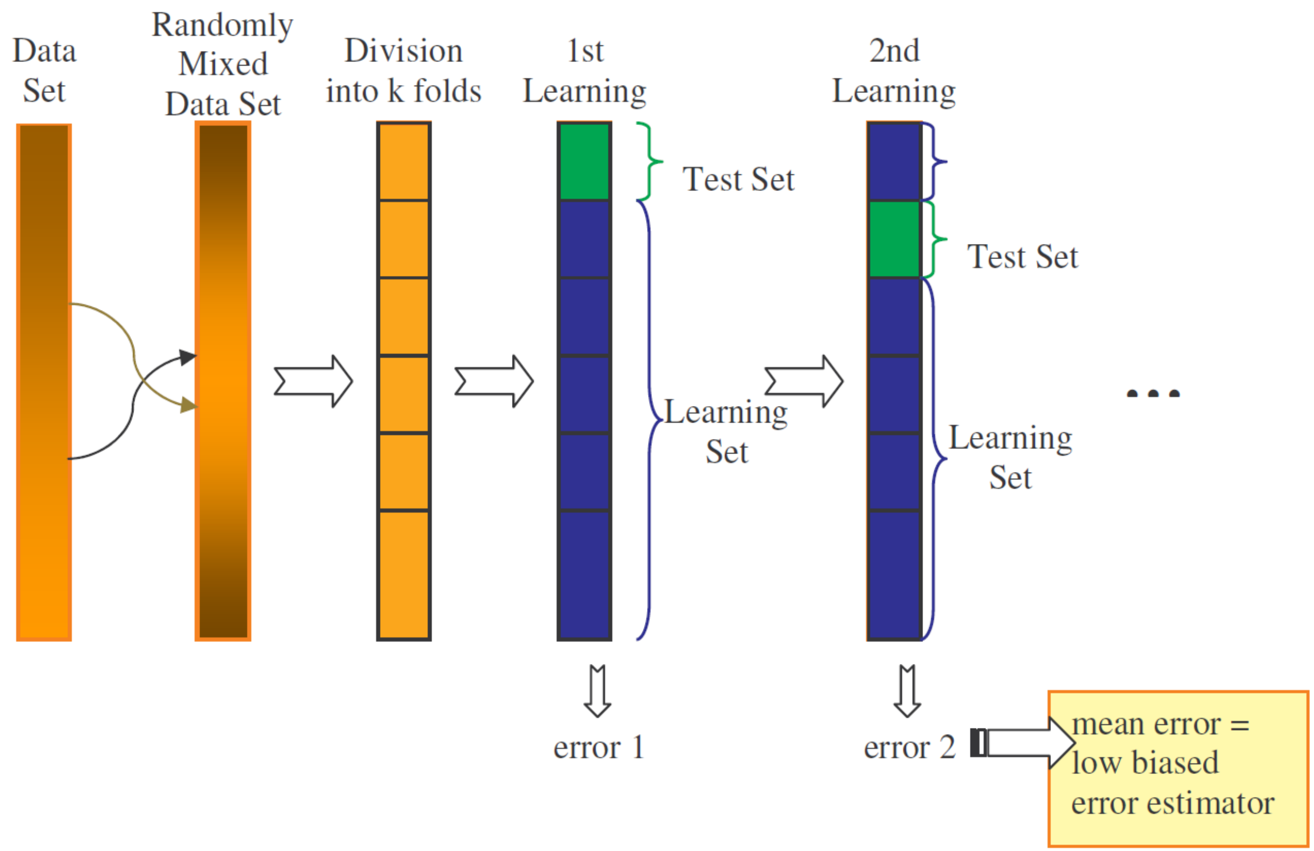


Figure 3 – Cross-Validation

Cross-validation

- Cross-validation: 10-fold
- Divide the input data into 10 parts: $p_1, p_2, \ldots, p_{10}$
- for i = 1 to 10 do
  - Use p_i for validation and get performance$_i$
  - Use parts other than p_i for training with some stopping criteria
- End for
- Summing up the performance$_i$ and get the mean.

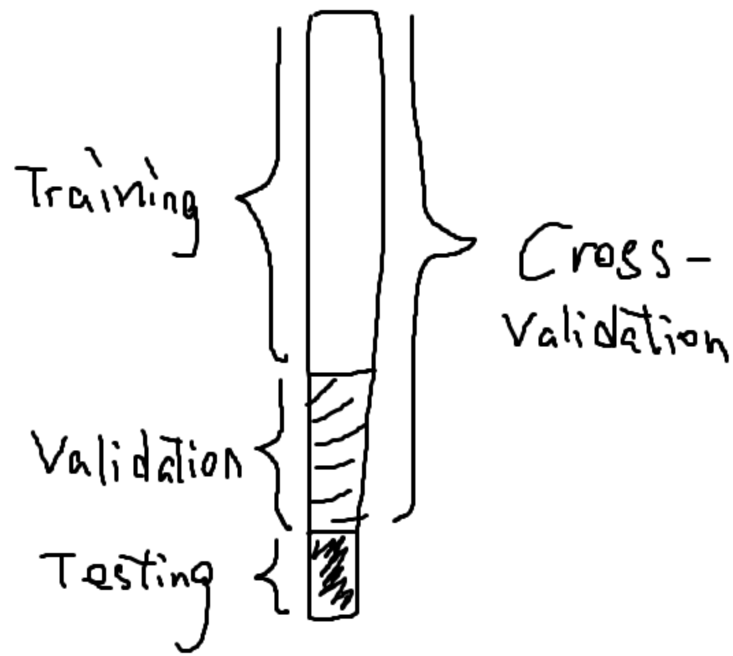
	Fold 1	Fold 2	…	Fold 10	Mean
Accuracy	0.99	0.97		0.87	0.90
Precision	0.90	0.92		0.95	0.92
Recall	0.85	0.84		0.90	0.91
Fmeasue	0.86	0.84		0.86	0.87

How to choose the best model ?

- Keep a part of the training data as testing data and evaluate your model using the **secretly kept** dataset.

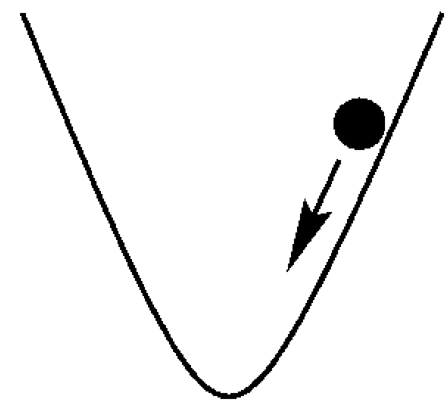
	F-measure on the testing data
Fold 1's Model	0.90
Fold 2's Model	0.92
…	0.91
Fold 10's Model	0.99

Best Model: Fold 10's model

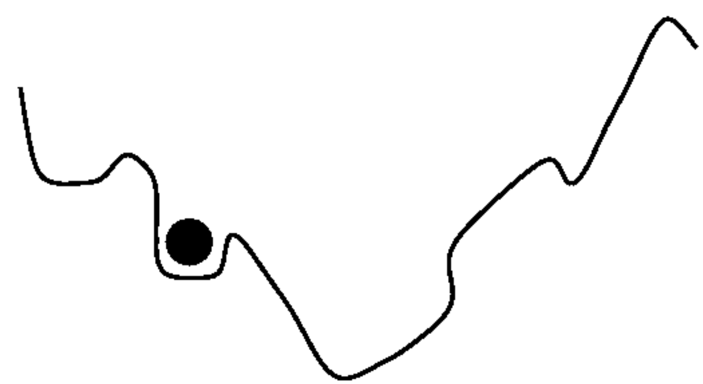


Local minima

- $w_{i,j} = w_{i,j} + \left(-\mu \frac{\partial E}{\partial w_{i,j}}\right)$
- Gradient descent will not always guarantee a global minimum
- We may get stuck in a local minimum !
- Can you think of some ways to solve the problem?



Global minimum



Local minimum

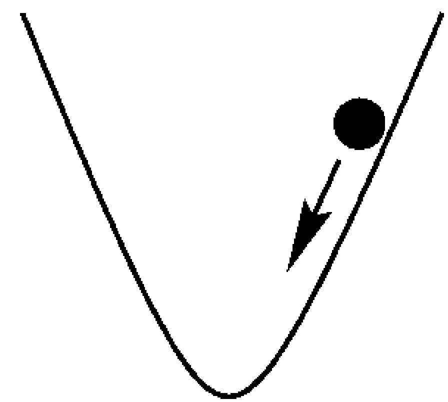
Local minima

- $w_{i,j} = w_{i,j} + \left(-\mu \frac{\partial E}{\partial w_{i,j}}\right)$

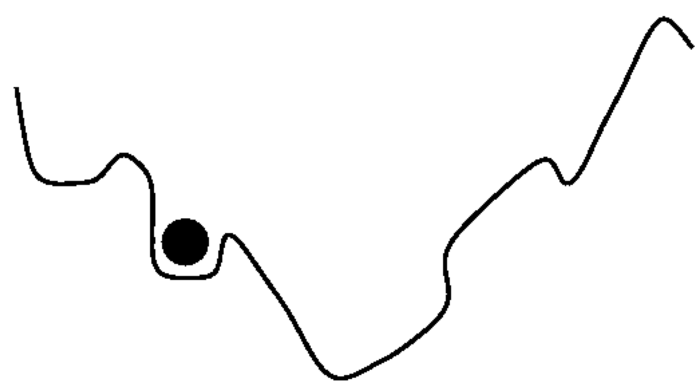
1) Randomize initial weights & Train multiple times

2) Tune the learning rate appropriately

3) Anymore?



Global minimum



Local minimum

Reference

- **Backward Propagation Tutorial**

  http://clemens.bytehammer.com/papers/BackProp/index.html

- **CS-449: Neural Networks**

  http://www.willamette.edu/~gorr/classes/cs449/intro.html

- **Cross-validation for detecting and preventing over-fitting**

  http://www.autonlab.org/tutorials/overfit10.pdf