Neural Networks

Chapter 18.6.3, 18.6.4 and 18.7

Introduction

- Known as:
  - Neural Networks (NNs)
  - Artificial Neural Networks (ANNs)
  - Connectionist Models
  - Parallel Distributed Processing (PDP) Models
- Neural Networks are a fine-grained, parallel, distributed, computing model

Introduction

- Attractions of NN approach
  - massively parallel
    - from a large collection of simple processing elements emerges interesting complex global behavior
  - can do complex tasks
    - pattern recognition (handwriting, facial expressions)
    - forecasting (stock prices, power grid demand)
    - adaptive control (autonomous vehicle control, robot control)
  - robust computation
    - can handle noisy and incomplete data due to fine-grained, distributed and continuous knowledge representation

Introduction

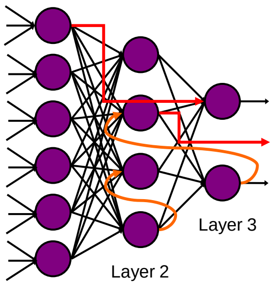
- Attractions of NN approach
  - fault tolerant
    - ok to have faulty elements and bad connections
    - isn't dependent on a fixed set of elements and connections
  - degrades gracefully
    - continues to function, at a lower level of performance, when portions of the network are faulty
  - uses inductive learning
    - useful for a wide variety of high-performance apps

Basics of NN

- Neural network composition:
  - large number of **units**
    - simple neuron-like processing elements
  - connected by a large number of **links**
    - directed from one unit to another
  - with a **weight** associate with each link
    - positive or negative real values
    - means of long term storage
    - adjusted by learning
  - and an **activation** level associated with each unit
    - result of the unit's processing
    - unit's output

Basics of NN

- Neural network configurations:
  - represent as a graph
    - nodes: units
    - arcs: links
  - single layered
  - multi-layered
  - feedback
  - layer skipping
  - fully connected?
    - N^2 links



Basics of NN

- Unit composition:
  - set of **input links**
    - from other units or sensors of the environment
  - set of **output links**
    - to other units or effectors of the environment
  - and an **activation function**
    - computes the output activation value using a simple function of the linear combination of its inputs

Basics of NN

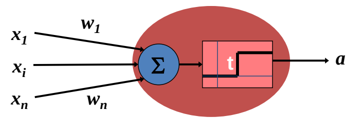
- Given n inputs, the unit's activation (i.e., output) is defined by:

  $a = g(w_1 x_1 + w_2 x_2 + \dots + w_n x_n) = g(\Sigma w_i x_i) = g(in)$

  where:
  - w_i are the weights
  - x_i are the input values
  - **$g(\,)$** is a simple, non-linear function, commonly:
    - **step**: activation *flips* from 0 to 1 when $in \geq$ threshold
    - **sign**: activation *flips* from –1 to 1 when $in \geq 0$
    - **sigmoid / logistic**: $g(in) = 1 / (1 + \exp(-in))$

Perceptrons

- Studied in the 1950s, mainly as simple networks for which there was an effective learning algorithm
- "**1-layer network**": one or more *output units*
- "Input units" don't count because they don't compute anything
- Output units are all **linear threshold units** (**LTU**s)
  - a unit's inputs, x_i, are weighted, w_i, and **linearly combined**
  - **step** function computes binary output activation value, a



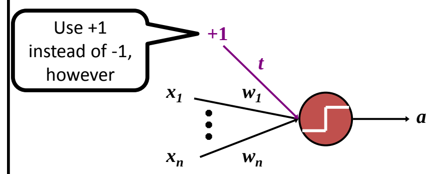
Perceptrons, Linear Threshold Units (LTU)

- Threshold is just another weight (called the **bias**):

  $(w_1 \cdot x_1) + (w_2 \cdot x_2) + ... + (w_n \cdot x_n) \geq t$

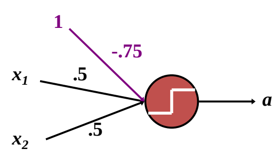
  is equivalent to

  $(w_1 \cdot x_1) + (w_2 \cdot x_2) + ... + (w_n \cdot x_n) + (t \cdot -1) \geq 0$

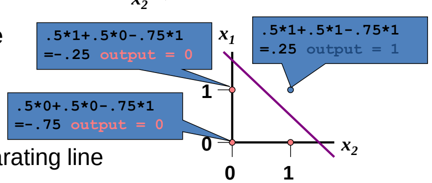


Perceptron Examples

- "AND" Perceptron:
  - inputs are 0 or 1
  - output is 1 when both x_1 and x_2 are 1

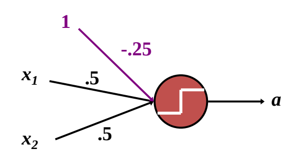


- **2D input space**
  - 4 possible data points
  - threshold is like a separating line

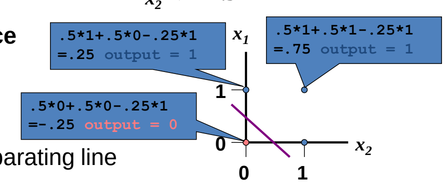


Perceptron Examples

- "OR" Perceptron:
  - inputs are 0 or 1
  - output is 1 when either x_1 or x_2 are 1



- **2D input space**
  - 4 possible data points
  - threshold is like a separating line



Perceptron Learning

How are weights learned by a Perceptron?

- Programmer specifies:
  - numbers of units in each layer
  - connectivity between units
  - so the only unknown is the set of weights
- Learning of weights is supervised
  - for each training example
    list of values for input into the input units of the network
  - the correct output is given
    list of values for the desired output of output units

Perceptron Learning Algorithm

1. Initialize the weights in the network
   (usually with random values)
2. Repeat until all examples correctly classified or some other stopping criterion is met

   foreach example, e, in training-set **do**

   a. O = neural_net_output(network, e)

   b. T = desired output, i.e., Target or Teacher's output

   c. update_weights(e, O, T)

- Each pass through *all* of the training examples is called an **epoch**

Perceptron Learning Rule

How should the weights be updated?

- Determining how to update the weights is an instance of the **credit assignment problem**

Perceptron Learning Rule:

- $w_i = w_i + \Delta w_i$
- where $\boldsymbol{\Delta w_i = \alpha\, x_i\, (T - O)}$
  - where x_i is the input associated with i^{th} input unit
  - α is a real-valued constant between 0.0 and 1.0 called the **learning rate**

Perceptron Learning Rule (PLR)

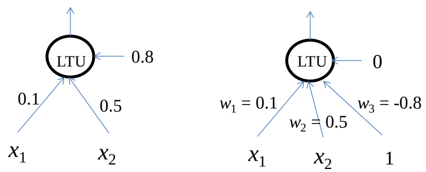
- $\Delta w_i = \alpha\, x_i\, (T - O)$ doesn't depend on w_i
- no change in weight (i.e., $\Delta w_i = 0$) if:
  - **correct output**, i.e., $T = O$ gives $\alpha \cdot x_i \cdot 0 = 0$
  - **zero input**, i.e., $x_i = 0$ gives $\alpha \cdot 0 \cdot (T - O) = 0$
- If T=1 and O=0, *increase* the weight

  so that maybe next time the result will exceed the output unit's threshold, causing it to be 1
- If T=0 and O=1, *decrease* the weight

  so that maybe next time the result won't exceed the output unit's threshold, causing it to be 0

Example: Learning OR

- $\Delta w_i = \alpha(T - O)x_i \;\; = 0.2(T - O)x_i$
- Initial network:



Example: Learning OR

bias

x1	x2	*T*	*O*	Δw1	w1	Δw2	w2	Δw3	w3
					0.1		0.5		-0.8
0	0	0	0	0	0.1	0	0.5	0	-0.8
0	1	1	0	0	0.1	0.2	0.7	0.2	-0.6
1	0	1	0	0.2	0.3	0	0.7	0.2	-0.4
1	1	1	1	0	0.3	0	0.7	0	-0.4
0	0	0	0	0	0.3	0	0.7	0	-0.4
0	1	1	1	0	0.3	0	0.7	0	-0.4
1	0	1	0	0.2	0.5	0	0.7	0.2	-0.2
1	1	1	1	0	0.5	0	0.7	0	-0.2
0	0	0	0	0	0.5	0	0.7	0	-0.2
0	1	1	1	0	0.5	0	0.7	0	-0.2
1	0	1	1	0	0.5	0	0.7	0	-0.2
1	1	1	1	0	0.5	0	0.7	0	-0.2

Perceptron Learning Rule (PLR)

- PLR is a “local" learning rule in that only local information in the network is needed to update a weight
- PLR performs gradient descent (hill-climbing) in "weight space“
- Iteratively adjusts all weights so that for each training example the error decreases (more correctly, error is monotonically non-increasing)

Perceptron Learning Rule (PLR)

Perceptron Convergence Theorem

- ***If a set of examples are learnable, then PLR will find an appropriate set of weights***
  - in a finite number of steps
  - independent of the initial weights
  - with sufficiently small α
- This theorem says that if a solution exists, PLR's gradient descent is guaranteed to find an optimal solution (i.e., 100% correct classification) for any 1-layer neural network

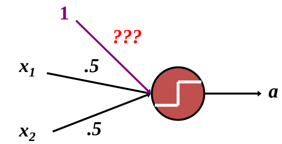
Limits of Perceptron Learning

What Can be Learned by a Perceptron?

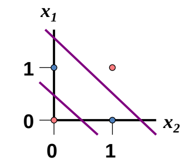
- Perceptron's output is determined by the separating hyperplane defined by $(w_1 \cdot x_1) + (w_2 \cdot x_2) + \ldots + (w_n \cdot x_n) = t$
- So, Perceptrons can only learn functions that are **linearly separable** (in input space)

Limits of Perceptron Learning

- "XOR" Perceptron?
  - inputs are 0 or 1
  - output is 1 when x_1 is 1 and x_2 is 0 or x_1 is 0 and x_2 is 1



- **2D input space with 4 possible data points**
- ***How do you separate + from – using a straight line?***



Perceptron Learning Summary

In general, the goal of learning in a Perceptron is to adjust the **separating hyperplane** that divides an *n*-dimensional space, where *n* is the number of input units (+ 1), by modifying the weights (and biases) until all of the examples with target value 1 are on one side of the hyperplane, and all of the examples with target value 0 are on the other side of the hyperplane

Beyond Perceptrons

- Perceptrons are too weak a computing model because they can only learn linearly-separable functions
- General NN's can have multiple layers of units, which enhance their computational ability; the challenge is to find a learning rule that works for multi-layered networks

Beyond Perceptrons

- *A **feed-forward multi-layered network** computes a function of the inputs and the weights*
- Input units
  - Input values are given by the environment
- Output units
  - activation is the output result
- Hidden units (between input and output units)
  - cannot observe directly
- Perceptrons have input units followed by one layer of output units, i.e., no hidden units

Beyond Perceptrons

- NN's with one hidden layer of a sufficient number of units, can compute functions associated with convex classification regions in input space
- NN's with two hidden layers are universal computing devices, although the complexity of the function is limited by the number of units
  - **If too few**, the network will be unable to represent the function
  - **If too many**, the network can memorize examples and is subject to "overfitting"

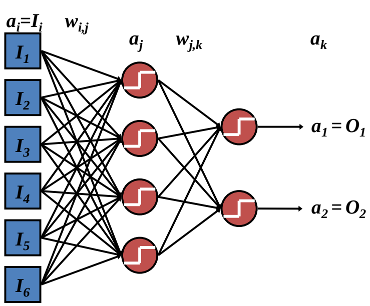
Two-Layer, Feed-Forward Neural Network

Input Units
Hidden Units
Output Units

Weights on links
from input to hidden

Weights on links
from hidden to output

Network Activations



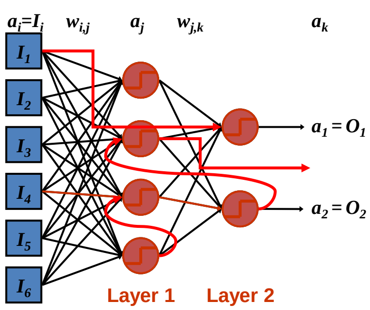
Two-Layer, Feed-Forward Neural Network

Two Layers:
count layers with units computing an activation

Feed-Forward:
each unit in a layer connects forward to all of the units in the next layer

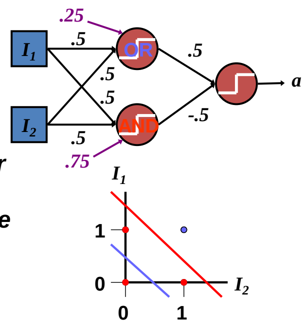
no cycles
- links within the same layer
- links to prior layers

no skipping layers



XOR Example

- XOR Perceptron?:
  - inputs are 0 or 1
  - output is 1 when I_1 is 1 and I_2 is 0 or I_1 is 0 and I_2 is 1
- ***Each unit in hidden layer acts like a Perceptron learning a separating line***
  - top hidden unit acts like an **OR** Perceptron
  - bottom hidden unit acts like an **AND** Perceptron

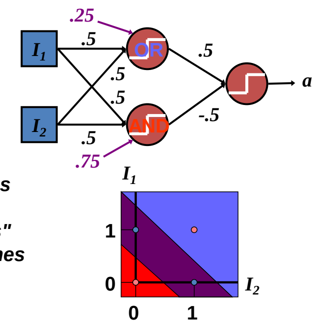


XOR Example

- XOR Perceptron?:
  - inputs are 0 or 1
  - output is 1 when I_1 is 1 and I_2 is 0 or I_1 is 0 and I_2 is 1
- ***To classify an example each unit in output layer combines these separating lines by intersecting the "half-planes" defined by the separating lines***

  when OR is 1 and AND is 0

  then output a, is 1



Learning in Multi-Layer, Feed-Forward Neural Nets

- PLR doesn't work in multi-layered feed-forward nets because the desired values for hidden units aren't known
- Must again solve the Credit Assignment Problem
  - determine which weights to credit/blame for the output error in the network
  - determine which weights in the network should be updated and how to update them

Learning in Multi-Layer, Feed-Forward Neural Nets

- **Back-Propogation**
  - method for learning weights in these networks
  - generalizes PLR
  - Rumelhart, Hinton and Williams, 1986
- Approach
  - gradient-descent algorithm to minimize the error on the training data
  - errors are propagated through the network starting at the output units and working *backwards* towards the input units

Back-Propagation Algorithm

1. Initialize the weights in the network (usually random values)
2. **Repeat until** all examples correctly classified or other stopping criterion is met

   foreach example, e, in training set **do**

   a. forward pass: $\boldsymbol{O}$ = neural_net_output(network, e)
   b. $\boldsymbol{T}$ = desired output, i.e., Target or Teacher's output
   c. calculate error ($\boldsymbol{T} - \boldsymbol{O}$) at the output units
   d. backward pass:
      i. **compute $\Delta w_{j,k}$ for all weights from hidden to output layer**
      ii. **compute $\Delta w_{i,j}$ for all weights from inputs to hidden layer**
   e. update_weights(network, $\Delta \boldsymbol{w}_{j,k}$, $\Delta \boldsymbol{w}_{i,j}$)

Computing Change in Weights

- Back-Propagation performs a ***gradient descent search*** in “weight space” to learn the network weights
- Given a network with *n* weights:
  - each configuration of weights is a vector, $\boldsymbol{W}$, of length *n* that defines an instance of the network
  - $\boldsymbol{W}$ can be considered a point in an *n*-dimensional **weight space**, where each dimension is associated with one of the connections in the network

Computing Change in Weights

- Given a training set of *m* examples:
  - each network defined by the vector W has an associated total error, $\boldsymbol{E}$, on all of training data
  - $\boldsymbol{E}$ the sum of the squared error (SSE) is defined as

    $$\boldsymbol{E} = \boldsymbol{E}_1 + \boldsymbol{E}_2 + \ldots + \boldsymbol{E}_m$$

    where each $\boldsymbol{E}_i$ is the squared error of the network on the $\boldsymbol{i}^{th}$ training example
- Given *n* output units in the network:

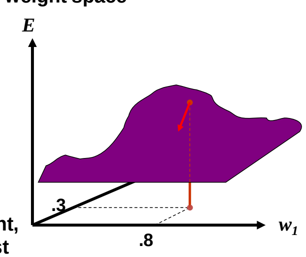
  $$E_i = ((T_1 - O_1)^2 + (T_2 - O_2)^2 + \ldots + (T_n - O_n)^2) / 2$$

  - $\boldsymbol{T}_i$ is the target value for the $\boldsymbol{i}^{th}$ example
  - $\boldsymbol{O}_i$ is the network output value for the $\boldsymbol{i}^{th}$ example

Computing Change in Weights

Visualized as a 2D error surface in weight space

- **Each point in $w_1 w_2$ plane is a weight configuration**
- **Each point has a total error E**
- **2D surface represents errors for all weight configurations**
- **Goal is to find a lower point on the error surface (local minima)**
- **Gradient descent follows the direction of the steepest descent, i.e., where E decreases the most**



Computing Change in Weights

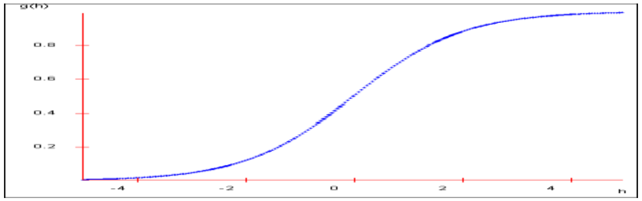
- The gradient is defined as
  ***Gradient_E* = [$\partial E/\partial w_1$, $\partial E/\partial w_2$, …, $\partial E/\partial w_n$]**
- Then change the *i*th weight by
  $\Delta w_i = -\alpha\, \partial E/\partial w_i$
- To compute the derivatives for calculating the gradient direction requires an activation function that is continuous, differentiable, non-decreasing and easily computed
  - can't use the Step function as in LTU's
  - instead use the Sigmoid function

Sigmoid Activation Function

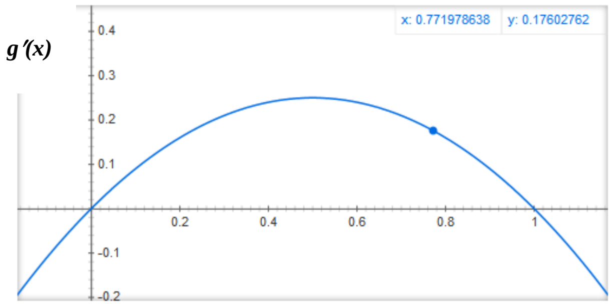
- The problem with LTU: step function is discontinuous, so cannot do gradient descent
- Solution: Replace with **Sigmoid function** (aka **Logistic function**):

$$g_w(x) = 1 / (1 + \exp(-wx))$$

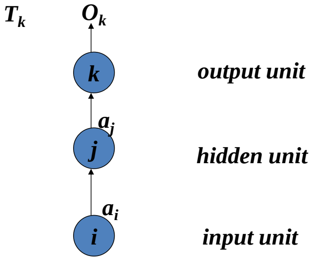
- Note: $g'(x) = g(x)\,(1 - g(x))$



First Derivative of Sigmoid Function



Updating the Weights



Updating Weights for 2-Layer Neural Network

- For **weights between hidden and output units**, generalized PLR for sigmoid activation is

$$\Delta w_{j,k} = -\alpha\ \partial E/\partial w_{j,k}$$
$$= -\alpha\ -a_j\ (T_k - O_k)\ g'(in_k)$$
$$= \alpha\ a_j\ (T_k - O_k)\ O_k\ (1 - O_k)$$
$$= \alpha\ a_j\ \Delta_k$$

$w_{j,k}$ weight on link from hidden unit *j* to output unit *k*
α learning rate parameter
a_j activation (i.e. output) of hidden unit *j*
T_k teacher output for output unit *k*
O_k actual output of output unit *k*
g' derivative of sigmoid activation function, which is $g' = g(1 - g)$

Updating Weights for 2-Layer Neural Network

- For **weights between input and hidden units**:
  - we don't have teacher-supplied correct output values
  - infer the error at these units by "back-propagating"
  - error at an output units is "distributed" back to each of the hidden units in proportion to the weight of the connection between them
  - total error is distributed to all of the hidden units that contributed to that error
- *Each hidden unit accumulates some error from each of the output units to which it is connected*

Updating Weights for 2-Layer Neural Network

- For **weights between inputs and hidden units**:

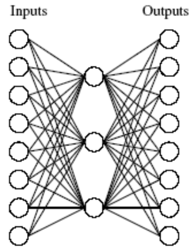
$$\Delta w_{i,j} = -\alpha\ \partial E/\partial w_{i,j}$$
$$= -\alpha\ -a_i\ g'(in_j) \sum w_{j,k}\ (T_k - O_k)\ g'(in_k)$$
$$= \alpha\ a_i\ a_j\ (1 - a_j) \sum w_{j,k}\ (T_k - O_k)\ O_k\ (1 - O_k)$$
$$= \alpha\ a_i\ \Delta_j \quad \text{where } \Delta_j = g'(in_j) \sum w_{j,k}\ \Delta_k$$

$w_{i,j}$ weight on link from input *i* to hidden unit *j*
$w_{j,k}$ weight on link from hidden unit *j* to output unit *k*
α learning rate parameter
a_j activation (i.e. output) of hidden unit *j*
T_k teacher output for output unit *k*
O_k actual output of output unit *k*
a_i input value *i*
g' derivative of sigmoid activation function, which is $g' = g(1-g)$

Back-Propagation Algorithm

1. Initialize the weights in the network (usually random values)
2. **Repeat until** all examples correctly classified or other stopping criterion
   foreach example *e* in training set **do**
   a. **forward pass**: O = neural_net_output(network, *e*)
   b. T = desired output, i.e., Target or Teacher's output
   c. calculate error ($T - O$) at the output units
   d. **backward pass**:
      i. compute $\Delta w_{j,k} = \alpha\ a_j\ \Delta_k = \alpha\ a_j\ (T_k - O_k)\ O_k\ (1 - O_k)$
      ii. compute $\Delta w_{i,j} = \alpha\ a_i\ \Delta_j = \alpha\ a_i\ a_j(1-a_j) \sum w_{j,k}\ (T_k - O_k)\ O_k\ (1- O_k)$
   e. update_weights(network, $\Delta w_{j,k}$, $\Delta w_{i,j}$)

Learning Hidden Layer Representation



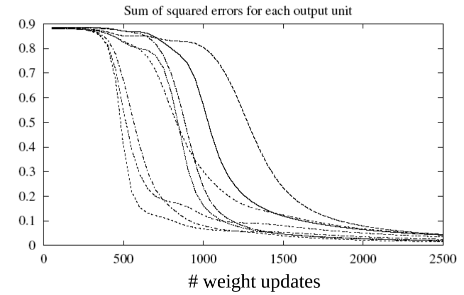
Input		Output
10000000	→	10000000
01000000	→	01000000
00100000	→	00100000
00010000	→	00010000
00001000	→	00001000
00000100	→	00000100
00000010	→	00000010
00000001	→	00000001

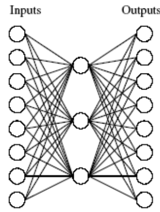
Can this be learned?

Slide by Guoping Qiu

Learning Hidden Layer Representation

Training

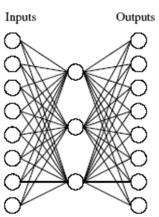




The evolving sum of squared errors for each of the eight output units

Slide by Guoping Qiu

Learning Hidden Layer Representation



Input		Hidden Values				Output
10000000	→	.89	.04	.08	→	10000000
01000000	→	.01	.11	.88	→	01000000
00100000	→	.01	.97	.27	→	00100000
00010000	→	.99	.97	.71	→	00010000
00001000	→	.03	.05	.02	→	00001000
00000100	→	.22	.99	.99	→	00000100
00000010	→	.80	.01	.98	→	00000010
00000001	→	.60	.94	.01	→	00000001

Learned hidden layer representation

Slide by Guoping Qiu

Multi-Layer Feedforward Networks

- Every Boolean function can be represented by a network with 1 hidden layer, but it might require an exponential number of hidden units
- Back-propagation algorithm performs gradient descent over “weight space” of entire network
- Will in general find a local, not global, error minimum
- Training can take thousands of epochs

Other Issues

- How should a network's performance be estimated?
- How should the learning rate parameter, α, be set?

  Use a **tuning set** to train using several candidate values for α, and then select the value that gives the lowest error on the test set

Other Issues

- How many hidden layers should be in the network?
  - usually just one hidden layer is used
- How many hidden units should be in a layer?
  - too few and the concept *can't* be learn
  - too many:
    - examples just memorized
    - overfitting, poor generalization
  - Use a **tuning set** or cross-validation to determine experimentally the number of units that minimizes error

Setting Parameters

- some learning algorithms require setting learning parameters
- they must be set without looking at the test data
- one approach: use a ***tuning set***

Using Data

- **Training set** is used to *learn a "model"* (e.g., a neural network's weights)
- **Tuning set** is used to judge and *select parameters* (e.g., learning rate and number of hidden units)
- **Testing set** is used to judge in a fair manner the *model's accuracy*
- All 3 datasets should be disjoint

Setting Parameters

Use a **Tuning set** for setting parameters:

1. Partition training examples into TRAIN, TUNE, and TEST sets
2. For each candidate parameter value, learn a neural network using the TRAIN set
3. Use TUNE set to evaluate error rates and determine which parameter value is best
4. Compute new neural network using selected parameter values and *both* TRAIN and TUNE sets
5. Use TEST set to compute performance accuracy

Other Issues

- How many examples should be in the training set?
  - ***the larger the better, but training is longer***

To obtain $1 - e$ correct classification on testing set:

- training set should be of size approximately n/e:
  - n is the number of weights in the network
  - e is test set error fraction between 0 and 1
- train to classify $1 - e/2$ of the training set correctly

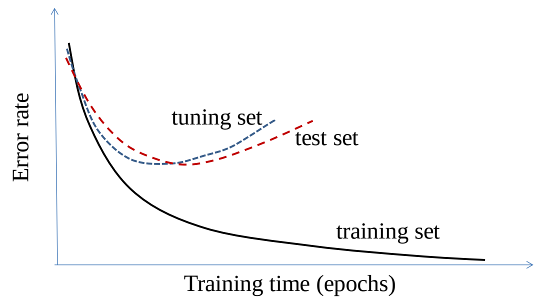
e.g., if $n = 80$ and $e = 0.1$ (i.e. 10% error on test set)

- training set of size is 800
- train until 95% correct classification
- should produce ~90% correct classification on test set

Other Issues

- When should training stop?
  - too soon and the concept isn't learned
  - too late:
    - overfitting, poor generalization
    - error rate will go up on the testing set
- Train the network until the error rate on a **tuning set** begins increasing rather than training until the error (i.e., SSE) is minimized

Tuning Sets



Summary

- Advantages
  - parallel processing architecture
  - robust with respect to node failure
  - fine-grained, distributed representation of knowledge
  - robust with respect to noisy data
  - incremental algorithm (i.e., learn as you go)
  - simple computations
  - empirically shown to work well for many problem domains

Summary

- Disadvantages
  - slow training (i.e., takes many epochs)
  - poor semantic interpretability
  - ad hoc network topologies (i.e., layouts)
  - hard to debug
  - may converge to local, not global, minimum of error
  - hard to describe a problem in terms of features with numerical values
  - not known how to model higher-level cognitive mechanisms with NN model

Applications

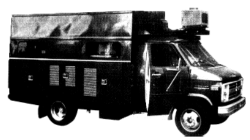
- NETtalk (Sejnowski & Rosenberg, 1987)
  learns to say text by mapping character strings to phonemes
- Neurogammon (Tesauro & Sejnowski, 1989)
  learns to play backgammon
- Speech recognition (Waibel, 1989)
  learns to convert spoken words to text
- Character recognition (Le Cun *et al.*, 1989)
  learns to convert page image to text

Application: Autonomous Driving

- ALVINN (Pomerleau, 1988)
  learns to control vehicle steering to stay in the middle of its lane
- Topology: 2-layer, feed-forward network using back-propagation learning
  - **Input layer:** 480 × 512 image @ 15 frames per second
    - color image is preprocessed to obtain a 30 × 32 image
    - each pixel is one byte, an integer from 0 to 255 corresponding to the brightness of the image
    - networks has 960 input units (= 30 × 32)

ALVINN

steering direction



[Pomerleau, 1995]

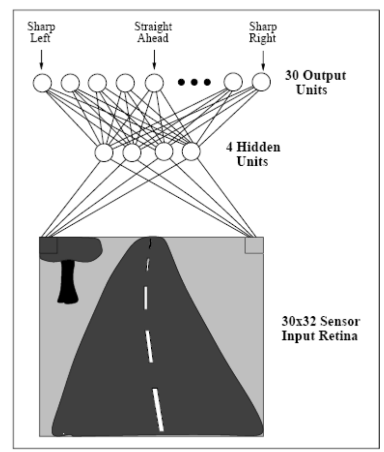
ALVINN

- Topology: **Output layer**
  - output is one of 30 discrete steering positions
    - output unit 1 means sharp left
    - output unit 30 means sharp right
  - target output is a set of 30 values
    - Gaussian distribution with a variance of 10 centered on the desired steering direction: $O_i = e^{[-(i-d)^2/10]}$
  - actual output for steering determined by
    - compute a least-squares best fit of output units' values to a Gaussian distribution with a variance of 10
    - peak of this distribution is taken as the steering direction
  - error for learning is: target output - actual output

ALVINN

- Topology: **Hidden layer**
  - only 4 hidden units with complete connectivity from
    - 960 input units to 4 hidden units
    - 4 hidden units to 30 output units

ALVINN’s 2-Layer Network



[Pomerleau, 1995]

ALVINN

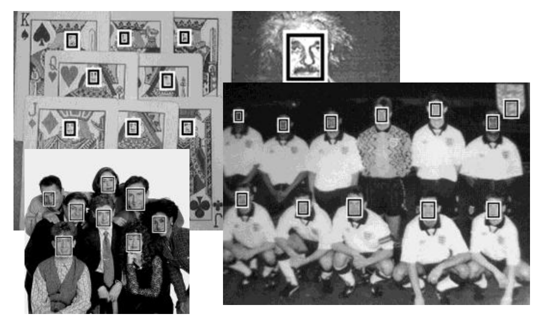
- Learning
  - continuously learns *on the fly* by observing
    - human driver (takes ~5 minutes from random initial weights)
    - itself (do an epoch of training every 2 seconds there after)
  - problem with using real continuous data:
    - there aren't negative examples
    - network may overfit data in recent images (e.g., straight road) at the expense of past images (e.g., road with curves)
  - solutions
    - generate negative examples by synthesizing views of the road that are incorrect for current steering
    - maintain a buffer of 200 real and synthesized images that keeps some images in many different steering directions

ALVINN

- Results
  - drove at speeds up to 70 mph
  - drove continuously for distances up to 90 miles
  - drove across the U.S. during different times of the day and with different traffic conditions
  - drove on:
    - single lane roads and highways
    - multi-lane highways
    - paved bike paths
    - dirt roads

ALVINN Demo

Application: Face Detection



Face Detection in Most Digital Cameras

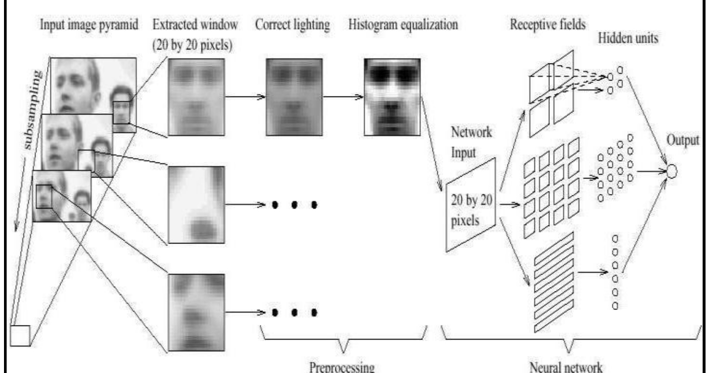
Canon Powershot

Also, smile and blink detection too in some cameras

Application: Face Detection

- Input = 20 x 20 pixel window, outputs a value ranging from –1 to +1 signifying the presence or absence of a face in the region
- The window is positioned at every location of the image
- To detect faces larger than 20 x 20 pixel, the image is repeatedly reduced in size

2-Layer Network



Application: Face Detection

- 2-layer feed-forward neural network
- Three types of hidden units
  - 4 look at 10 x 10 subregions
  - 16 look at 5 x 5 subregions
  - 6 look at 20 x 5 horizontal stripes of pixels
- Training set
  - 1,050 initial face images. More face examples generated from this set by rotation and scaling. Desired output: +1
  - Non-face training samples: 8,000 non-face training samples from 146,212,178 subimage regions! Desired output: -1

Face Detection Results

Results

- Notice detection at multiple scales

Results

Face Detection Demos

(but *not* using neural nets)

- http://flashfacedetection.com/
- http://flashfacedetection.com/camdemo2.html

Evaluating Performance

How might the performance be evaluated?

- **Predictive accuracy of classifier**
- Speed of learner
- Speed of classifier
- Space requirements

Training Set Error

- For each example in the training set, use the classifier to see what class it predicts

  For what number of examples does the classifier's prediction disagree with the teacher value in the database?
- This quantity is called the ***training set error***. The smaller the better.
- But why are we doing learning anyway?
  - More important to assess how well the classifier predicts output for **future data**

Test Set Error

- Suppose we are forward thinking
- We hide some data away when we learn the classifier
- But once learned, we see how well the classifier predicts that data
- This is a good simulation of what happens when we try to predict future data
- Called the **Test Set Error**

Evaluating Classifiers

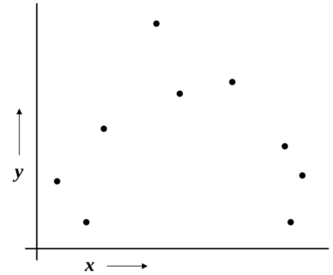
- During training
  - Train a classifier from a training set $(x_1,y_1), (x_2,y_2), \dots, (x_n,y_n)$
- During testing
  - For ***new*** test data, $x_{n+1}, \dots, x_{n+m}$, your classifier generates predicted labels $y'_{n+1}, \dots, y'_{n+m}$
- Test set accuracy:
  - You need to know the true test labels: $y_{n+1}, \dots, y_{n+m}$
  - Test set accuracy: $acc = \frac{1}{m} \sum_{i=n+1}^{n+m} 1_{y_i = y'_i}$
  - Test set error rate = $1 - acc$

Evaluating Performance Accuracy

- Use separate test examples to estimate accuracy

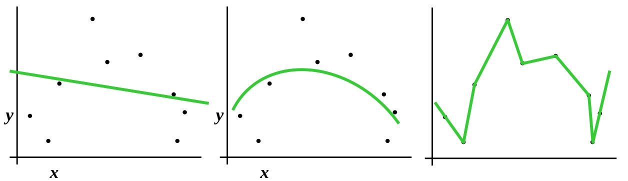
1. randomly partition training examples into:
   TRAIN set (~70% of all training examples)
   TEST set (~30% of all training examples)
2. generate decision tree using the TRAIN set
3. use TEST set to evaluate accuracy
   `accuracy = #correct / #total`

A Regression Problem



y = f(x) + noise
Can we learn f from this data?

Which is Best?



Linear f ***Quadratic f*** ***Piecewise Linear f***

Fits the data best, including the noise!

What we *really* want: Fit that will give the best results on future data (drawn from the same distribution)

The **Overfitting Problem**
Example: Predicting US Population

x=Year	***y=Million***
1900	***75.99***
1910	***91.97***
1920	***105.71***
1930	***123.2***
1940	***131.67***
1950	***150.7***
1960	***179.32***
1970	***203.21***
1980	***226.51***
1990	***249.63***
2000	***281.42***

- We have some training data (n=11)
- What will the population be in 2010?

Regression: Polynomial Fit

- The degree *d* (complexity of the model) is important

$$f(x) = c_d x^d + c_{d-1} x^{d-1} + \cdots + c_1 x + c_0$$

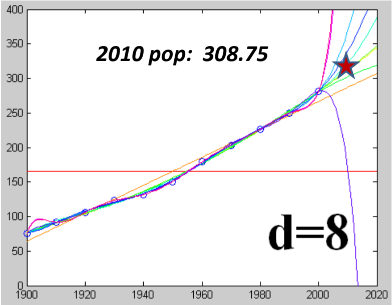
- Fit (= learn) coefficients $c_d, \ldots c_0$ to minimize Mean Squared Error (MSE) on training data

$$MSE = \frac{1}{n} \sum_{i=1}^{n} (y_i - f(x_i))^2$$

Overfitting

- As *d* increases, MSE on training data improves, but **prediction** outside training data **worsens**

degree=0 MSE=4181.451643
degree=1 MSE=79.600506
degree=2 MSE=9.346899
degree=3 MSE=9.289570
degree=4 MSE=7.420147
degree=5 MSE=5.310130
degree=6 MSE=2.493168
degree=7 MSE=2.278311
degree=8 MSE=1.257978
degree=9 MSE=0.001433
degree=10 MSE=0.000000

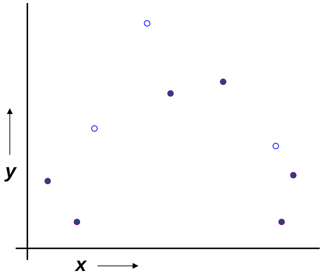


Experimental Evaluation of Performance

Test Set Method

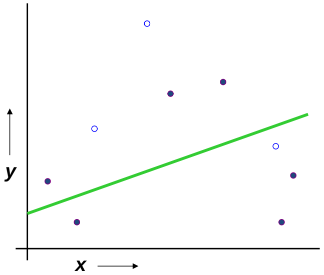
1. Randomly choose say 30% of the data to be the test set, and the remaining 70% is the training set
2. Build classifier using the training set
3. Estimate future performance using the test set

Test Set Method



1. Randomly choose 30% of the data to be in a test set
2. The remainder is a training set

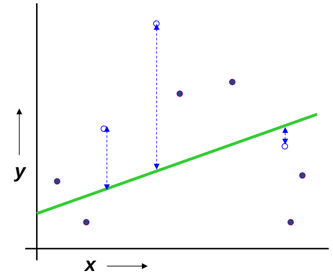
Test Set Method



1. *Randomly choose 30% of the data to be in a test set*
2. *The remainder is a training set*
3. *Perform your regression on the training set*

(Linear regression example)

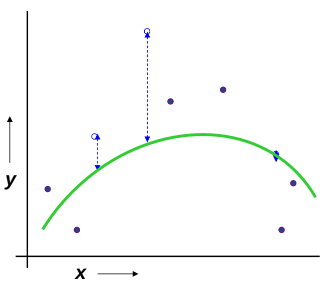
Test Set Method



1. *Randomly choose 30% of the data to be in a test set*
2. *The remainder is a training set*
3. *Perform your regression on the training set*
4. *Estimate your future performance with the test set*

(Linear regression example)
Mean Squared Error = 2.4

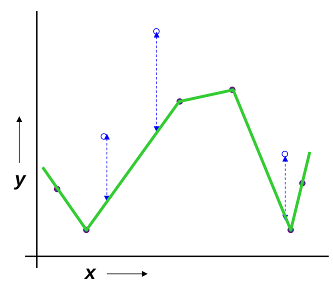
Test Set Method



1. *Randomly choose 30% of the data to be in a test set*
2. *The remainder is a training set*
3. *Perform your regression on the training set*
4. *Estimate your future performance with the test set*

(Quadratic regression example)
Mean Squared Error = 0.9

Test Set Method



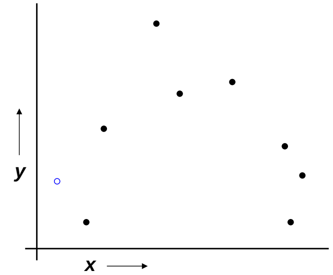
1. *Randomly choose 30% of the data to be in a test set*
2. *The remainder is a training set*
3. *Perform your regression on the training set*
4. *Estimate your future performance with the test set*

(Join the dots example)
Mean Squared Error = 2.2

Test Set Method

- Strengths
  - Very simple
- Weaknesses
  - Wastes data because test set is not used to construct the best classifier
  - If we don't have much data, our test set might be lucky or unlucky in terms of what's in it, making the results on the test set a "high variance" estimator of the real performance

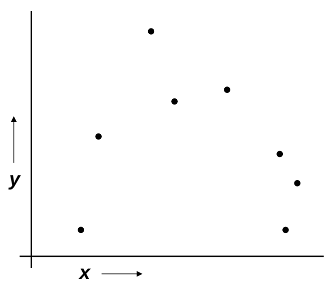
LOOCV (Leave-one-out Cross Validation)



For k=1 to R

1. ***Let (x_k,y_k) be the k^{th} record***

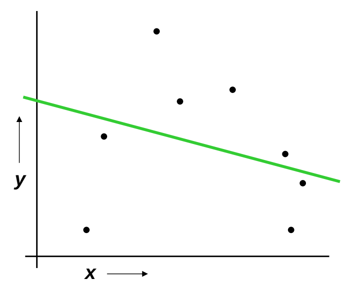
LOOCV (Leave-one-out Cross Validation)



For k=1 to R

1. ***Let (x_k,y_k) be the k^{th} record***
2. ***Temporarily remove (x_k,y_k) from the dataset***

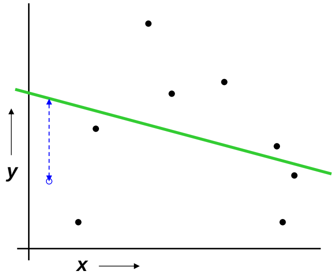
LOOCV (Leave-one-out Cross Validation)



For k=1 to R

1. ***Let (x_k,y_k) be the k^{th} record***
2. ***Temporarily remove (x_k,y_k) from the dataset***
3. ***Train on the remaining R-1 datapoints***

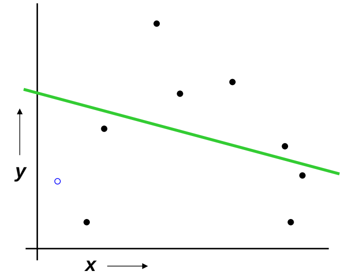
LOOCV (Leave-one-out Cross Validation)



For k=1 to R

1. *Let (x_k, y_k) be the k^{th} record*
2. *Temporarily remove (x_k, y_k) from the dataset*
3. *Train on the remaining R-1 datapoints*
4. *Note your error (x_k, y_k)*

LOOCV (Leave-one-out Cross Validation)

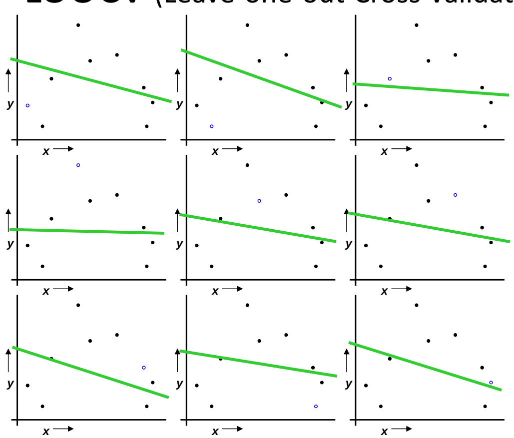


For k=1 to R

1. *Let (x_k, y_k) be the k^{th} record*
2. *Temporarily remove (x_k, y_k) from the dataset*
3. *Train on the remaining R-1 datapoints*
4. *Note your error (x_k, y_k)*

When you've done all points, report the mean error.

LOOCV (Leave-one-out Cross Validation)



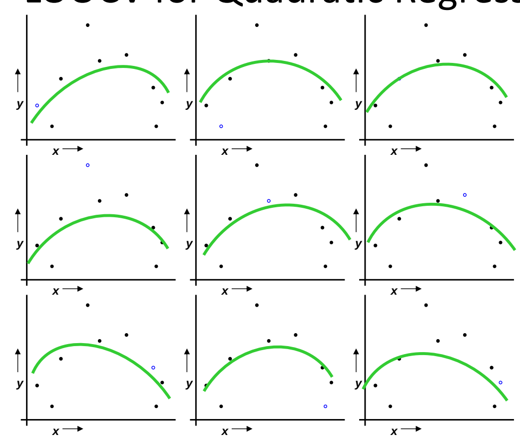
For k=1 to R

1. *Let (x_k, y_k) be the k^{th} record*
2. *Temporarily remove (x_k, y_k) from the dataset*
3. *Train on the remaining R-1 datapoints*
4. *Note your error (x_k, y_k)*

When you've done all points, report the mean error.

MSE_{LOOCV} = 2.12

LOOCV for Quadratic Regression



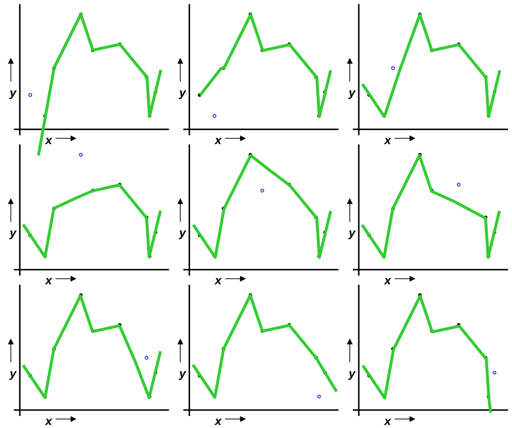
For k=1 to R

1. *Let (x_k, y_k) be the k^{th} record*
2. *Temporarily remove (x_k, y_k) from the dataset*
3. *Train on the remaining R-1 datapoints*
4. *Note your error (x_k, y_k)*

When you've done all points, report the mean error.

MSE_{LOOCV}=0.962

LOOCV for Join The Dots



For k=1 to R

1. *Let (x_k,y_k) be the k^{th} record*
2. *Temporarily remove (x_k,y_k) from the dataset*
3. *Train on the remaining R-1 datapoints*
4. *Note your error (x_k,y_k)*

When you've done all points, report the mean error.

MSE_{LOOCV}=3.33

Experimental Evaluation of Performance

- **Leave-One-Out Cross Validation**

  For i = 1 to N do   // N = number of examples

  1. Let (x_i, y_i) be the i^{th} example
  2. Remove (x_i, y_i) from the dataset
  3. Train on the remaining N-1 examples
  4. Compute error on i^{th} example

- Accuracy = mean accuracy on all N runs
- Doesn't waste data but is expensive
- **Use when you have a small dataset**

Experimental Evaluation of Performance

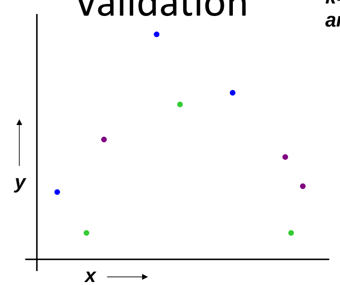
- **Cross-Validation Method**

Often use K = 3 or 10

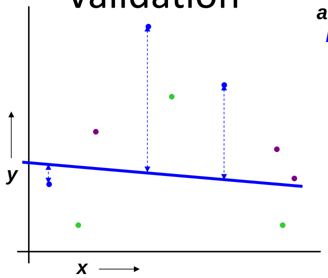
1. divide all examples into K disjoint subsets $E = E_1, E_2, \ldots, E_K$
2. for each $i = 1, \ldots, K$
   - let TEST set = E_i and TRAIN set = $E - E_i$
   - compute decision tree using TRAIN set
   - determine accuracy PA_i using TEST set
3. compute ***K*-fold cross-validation** estimate of performance = mean error = $(PA_1 + PA_2 + \ldots + PA_K)/K$

k-fold Cross Validation

Randomly break the dataset into k partitions (in our example we'll have k=3 partitions colored Red, Green and Blue)



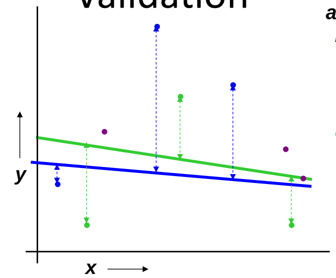
k-fold Cross Validation



Randomly break the dataset into k partitions (in our example we'll have k=3 partitions colored Red, Green and Blue)

- ***For the red partition: Train on all the points not in the red partition. Find the test-set sum of errors on the red points.***

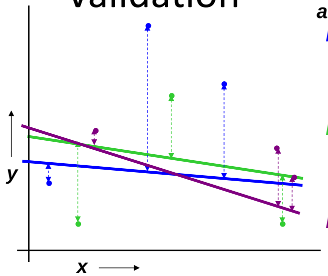
k-fold Cross Validation



Randomly break the dataset into k partitions (in our example we'll have k=3 partitions colored Red, Green and Blue)

- ***For the red partition: Train on all the points not in the red partition. Find the test-set sum of errors on the red points.***
- ***For the green partition: Train on all the points not in the green partition. Find the test-set sum of errors on the green points.***

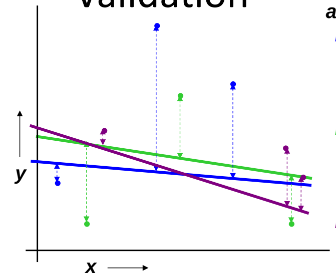
k-fold Cross Validation



Randomly break the dataset into k partitions (in our example we'll have k=3 partitions colored Red, Green and Blue)

- ***For the red partition: Train on all the points not in the red partition. Find the test-set sum of errors on the red points.***
- ***For the green partition: Train on all the points not in the green partition. Find the test-set sum of errors on the green points.***
- ***For the blue partition: Train on all the points not in the blue partition. Find the test-set sum of errors on the blue points.***

k-fold Cross Validation



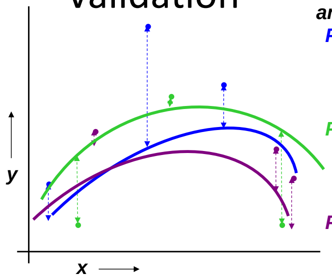
Linear Regression MSE_{3FOLD}=2.05

Randomly break the dataset into k partitions (in our example we'll have k=3 partitions colored Red, Green and Blue)

- ***For the red partition: Train on all the points not in the red partition. Find the test-set sum of errors on the red points.***
- ***For the green partition: Train on all the points not in the green partition. Find the test-set sum of errors on the green points.***
- ***For the blue partition: Train on all the points not in the blue partition. Find the test-set sum of errors on the blue points.***

Then report the mean error

k-fold Cross Validation

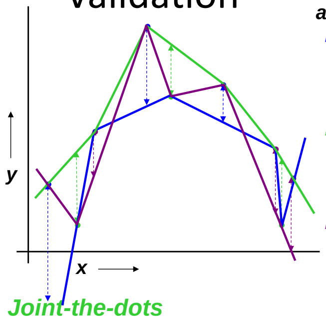


Quadratic Regression
MSE_{3FOLD}=1.11

Randomly break the dataset into k partitions (in our example we'll have k=3 partitions colored Red, Green and Blue)

- ***For the red partition: Train on all the points not in the red partition. Find the test-set sum of errors on the red points.***
- ***For the green partition: Train on all the points not in the green partition. Find the test-set sum of errors on the green points.***
- ***For the blue partition: Train on all the points not in the blue partition. Find the test-set sum of errors on the blue points.***
- ***Then report the mean error***

k-fold Cross Validation



Joint-the-dots
MSE_{3FOLD}=2.93

Randomly break the dataset into k partitions (in our example we'll have k=3 partitions colored Red, Green and Blue)

- ***For the red partition: Train on all the points not in the red partition. Find the test-set sum of errors on the red points.***
- ***For the green partition: Train on all the points not in the green partition. Find the test-set sum of errors on the green points.***
- ***For the blue partition: Train on all the points not in the blue partition. Find the test-set sum of errors on the blue points.***
- ***Then report the mean error***