

## Chapter 6

# Model selection and overfitting

Neural networks and deep learning



# Model Selection



In most pattern recognition techniques, there exist one or more free parameters. For example in MLPs, learning parameters, no of layers, number of neurons, etc.

Every set of parameters of the network leads to a specific model.

How do we determine the “optimum” parameter(s) or the model for a given regression or classification problem?



# Performance Estimation



How do we measure the performance of the network?

Some metrics:

1. Mean-square error/Root-mean square error for regression — the mean-squared error or its square root. A measure of the deviation from actual.

$$MSE = \frac{1}{P} \sum_{p=1}^P \sum_{k=1}^K (d_{pk} - y_{pk})^2 \text{ and } RMSE = \sqrt{\frac{1}{P} \sum_{p=1}^P \sum_{k=1}^K (d_{pk} - y_{pk})^2}$$

2. Misclassification error for a classifier.

$$\text{misclassification error} = \sum_{p=1}^P 1(d_p \neq y_p)$$

where  $d_p$  is the target and  $y_p$  is the predicted output of pattern  $p$ .  $1(\cdot)$  is the indicator function.



# Terminology

**Apparent error** (training error): the error on the training data.  
What the learning algorithm tries to optimize.

**True error**: the error that will be obtained in use (over the whole sample space). What we want to optimize *but* unknown.

However, the **apparent error** is not always a good estimate of the **true error**. It is just an optimistic measure of it.

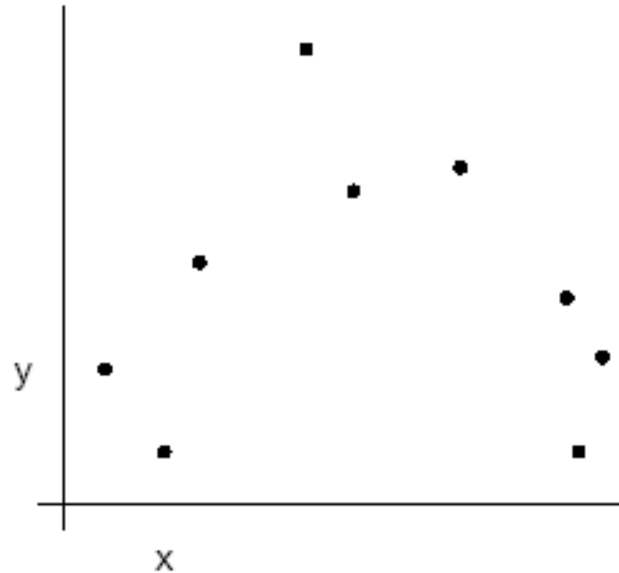
**Test error**: (out-of-sample error) an estimate of the true error obtained by testing the network on some independent data.

Generally, a larger test set helps provide a greater confidence on the accuracy of the estimate.



# Example: Regression

Given the following sample data:

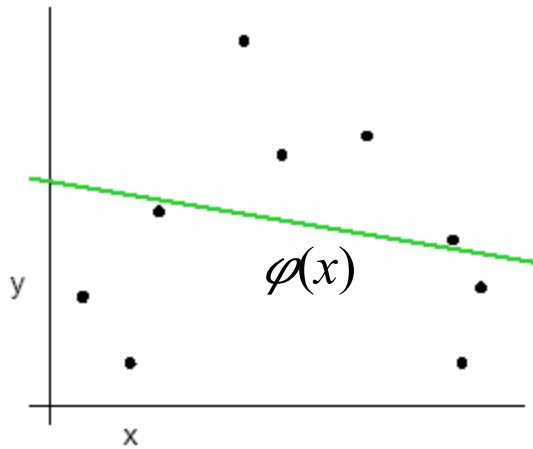


Approximate  $y \approx \varphi(x)$  using an ANN

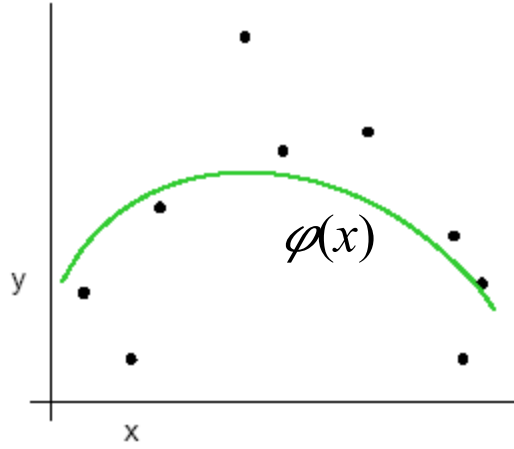


## Example: Regression

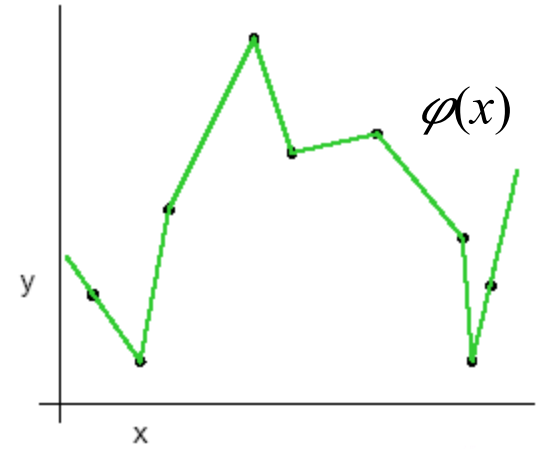
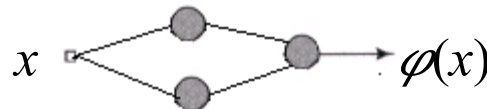
## Which one is the best approximation model?



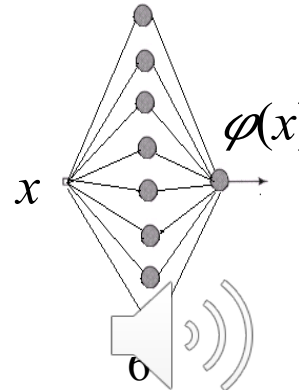
## Linear regression, Linear Activation fn



Quadratic (non-linear)  
regression, Sigmoid  
Activation fn

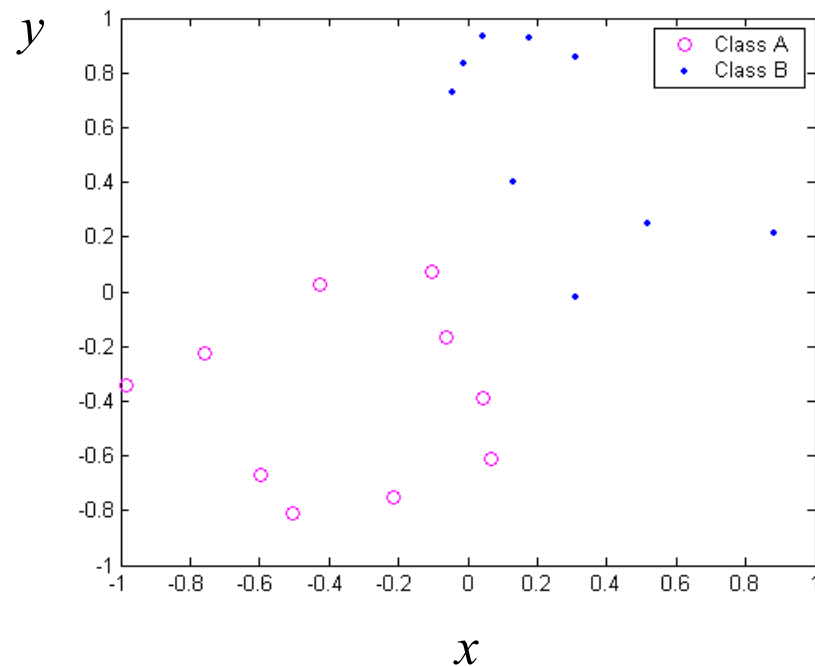


## Piecewise Linear regression, Linear Activation fn



# Example: Classification

Given the following sample data:

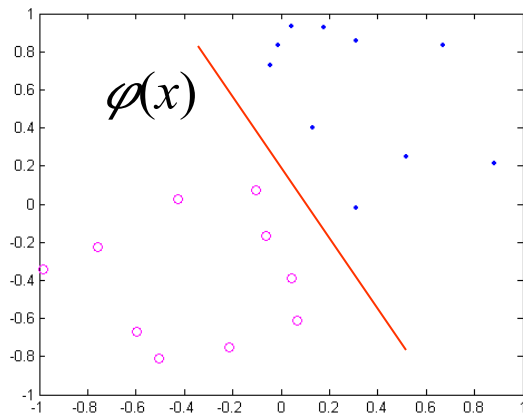


Classify the following data using an ANN.

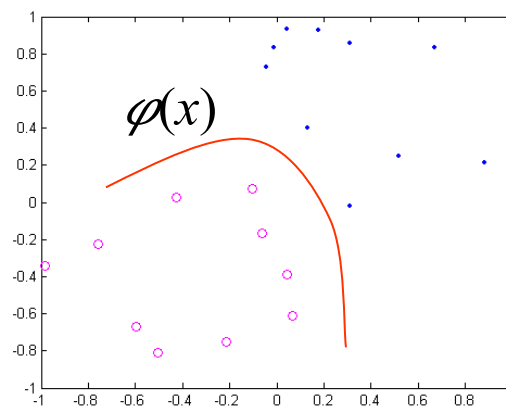


# Example: Classification

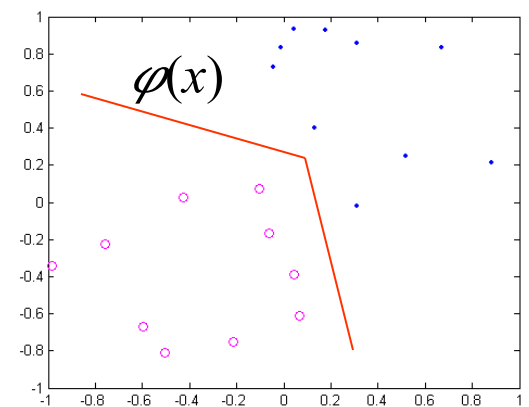
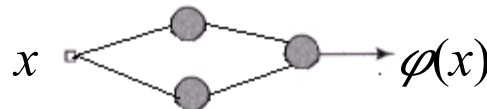
Which one is the best classification model?



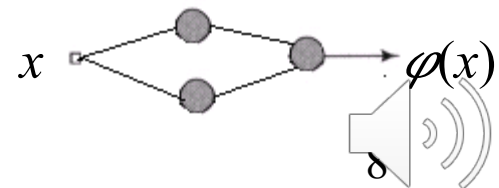
Linear Classification,  
Threshold Activation fn



Quadratic  
Classification, *Sigmoid*  
*Activation fn*



Piecewise Linear  
Classification, *Linear*  
*Activation fn*





# True Error Estimation

***Intuition:*** Choose the model with the best fit to the data?

***Meaning:*** Choose the model that provides the lowest error rate on the entire sample population. Of course, that error rate is the *true error rate*.

**“However, to choose a model, we must first know how to estimate the error of a model.”**

**The entire sample population is often unavailable.**



# Validation

In real applications, we only have access to a finite set of examples, usually smaller than we wanted.

*Validation* is the approach to use the entire example data available to select the model and estimate the error rate. The validation uses a part of the data to select the model, which is known as the *validation test*.

Validation attempts to solve fundamental problems encountered:

- The final model tends to *overfit* the training data. It is not uncommon to have 100% correct classification on training data.
- There is no way of knowing how well the model performs on unseen data
- The error rate estimate will be overly optimistic (usually lower than the true error rate) .



# Validation Methods

An effective approach is to split the entire data into subsets, i.e., Training/Validation/Testing datasets.

Some Validation Methods:

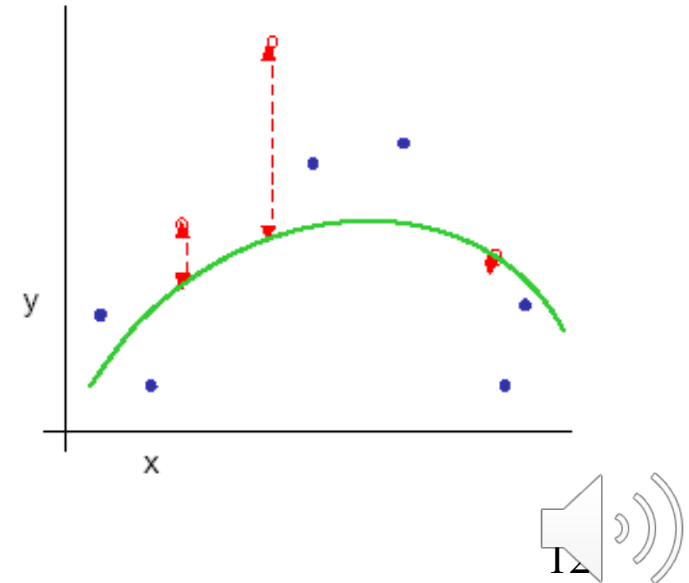
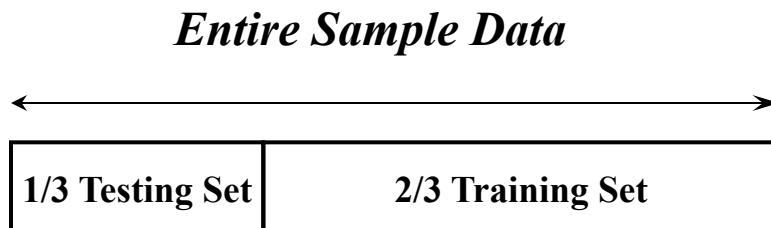
- The Holdout (1/3 - 2/3 Rule)
- Re-sampling techniques
  - Random Subsampling
  - K-fold Cross-Validation
  - Leave one out Cross-Validation
- Three-way data splits



# Holdout Method

**Split entire dataset into two sets:**

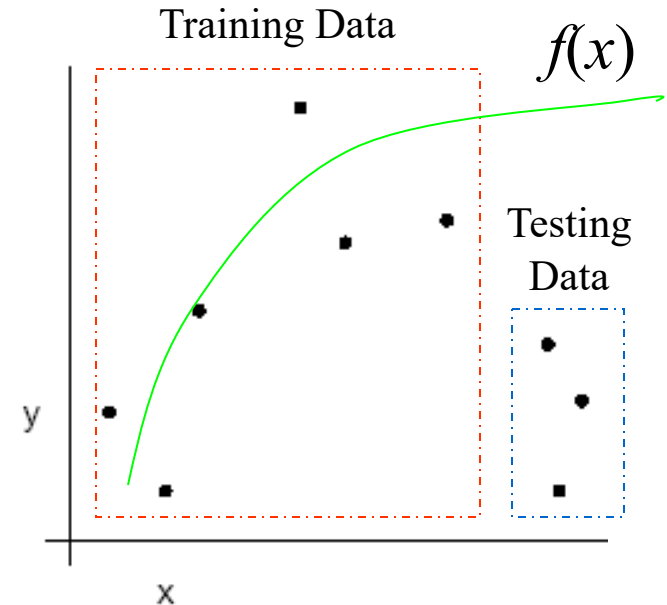
- *Training set* (blue): used to train the classifier
- *Testing set* (red): used to estimate the error rate of the trained classifier on unseen data samples



# Holdout Method

## The holdout method has two basic drawbacks:

- In problems where we have a sparse dataset we may not be able to afford the “luxury” of setting aside a portion of the dataset for testing
- Since it is a single train-and-test experiment, the holdout estimate of error rate will be misleading if we happen to get an “unfortunate” split.



An “unfortunate” split for an approximation example. Results in large test error.



# Random Sampling Methods

**Limitations of the holdout can be overcome with a family of resampling methods at the expense of more computations:**

- Random Subsampling
- K-Fold Cross-Validation
- Leave-one-out Cross-Validation

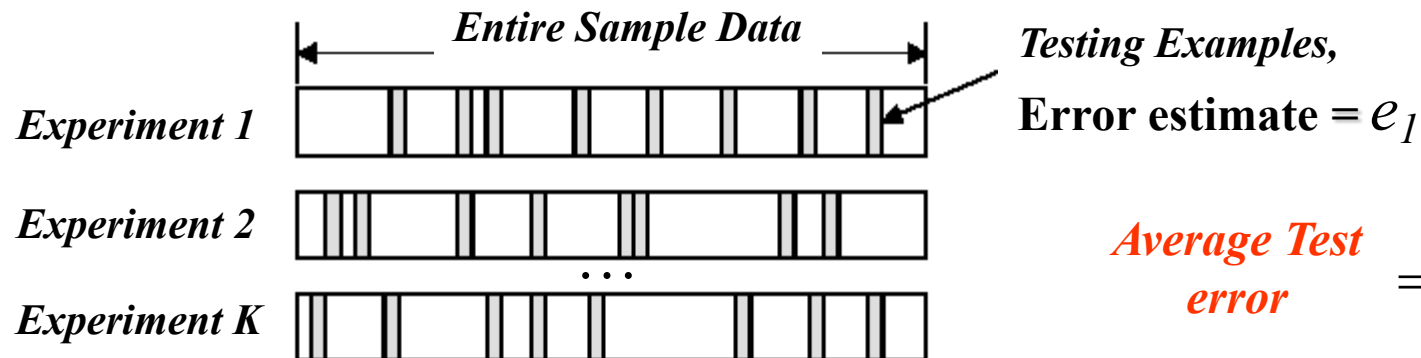


# K Data Splits

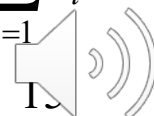
## Random SubSampling

Random Subsampling performs  $K$  data splits of the dataset

- Each split randomly selects a (fixed) no. of examples
- For each data split we retrain the classifier from scratch with the training data
- Examples and estimate  $e_i$  with the test examples

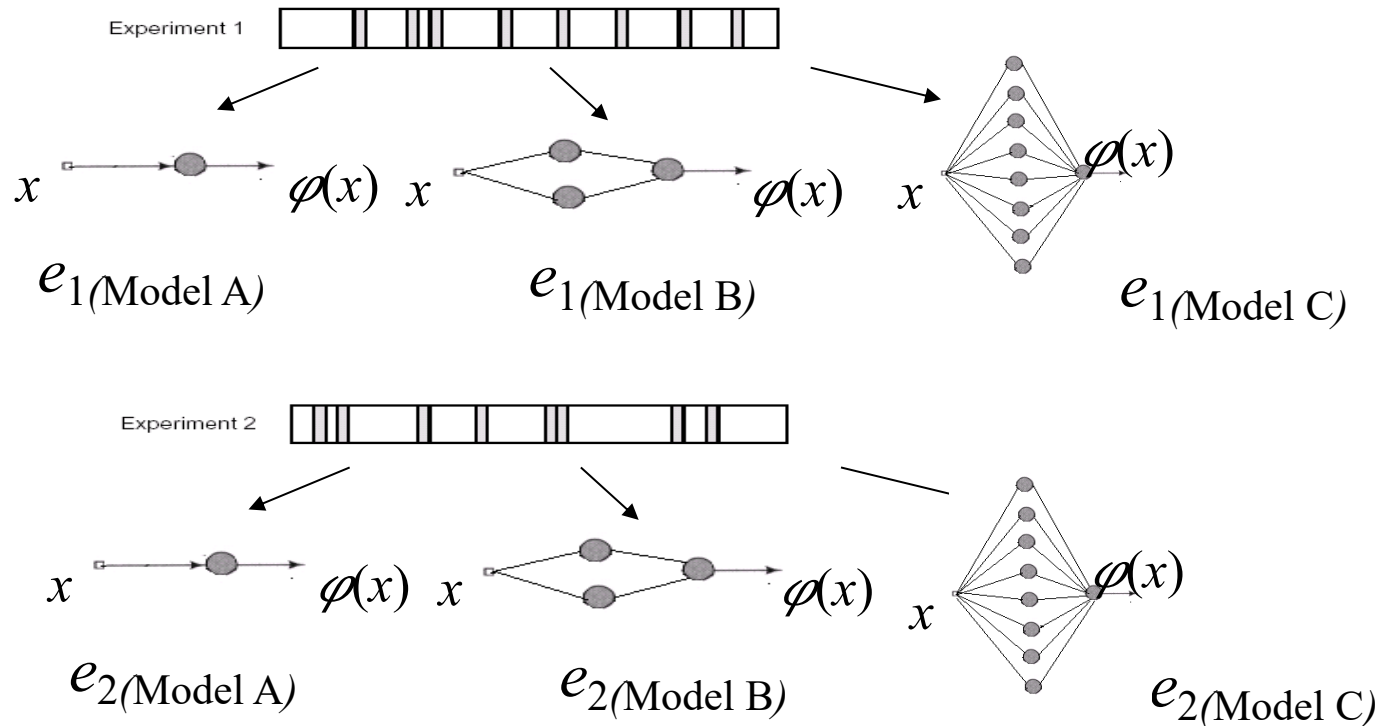


**Average Test error**

$$= \frac{1}{K} \sum_{i=1}^K e_i$$


# Example: K=2 Data Splits

## Random SubSampling



**Average Test error**

$$= \frac{1}{2}(e_1 + e_2)$$

Choose the model with the best test error, i.e., lowest Average test error!

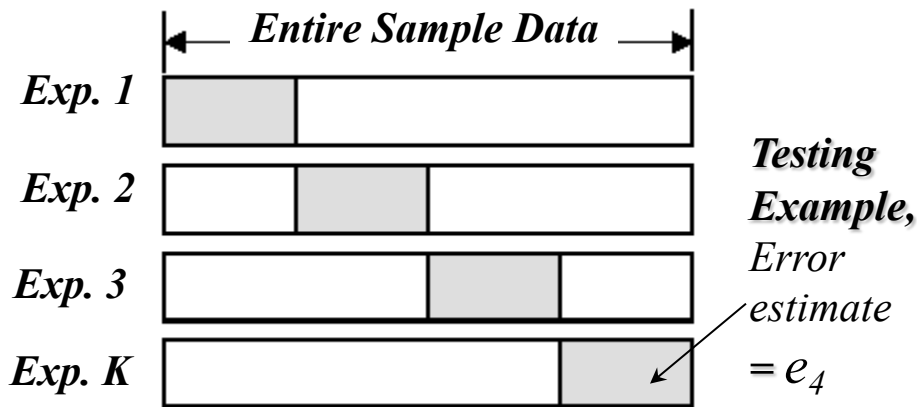




# K-fold Cross Validation

Create a  $K$ -fold partition of the the dataset:

- For each of  $K$  experiments, use  $K-1$  folds for training and the remaining one fold for testing



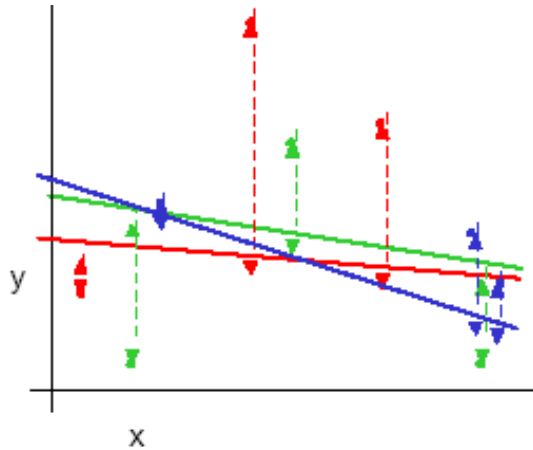
**Cross-validation error**

$$= \frac{1}{K} \sum_{i=1}^K e_i$$

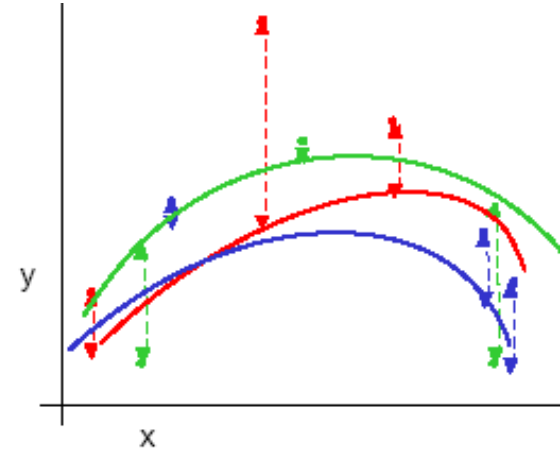
$K$ -fold cross validation is similar to Random Subsampling. The *advantage* of  $K$ -Fold Cross validation is that all examples in the dataset are eventually used for both training and testing



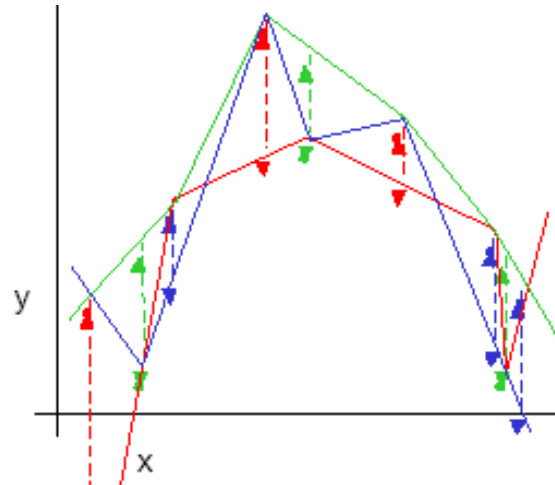
# 3-fold Cross Validation



$e_1(\text{Model A})$



$e_1(\text{Model B})$



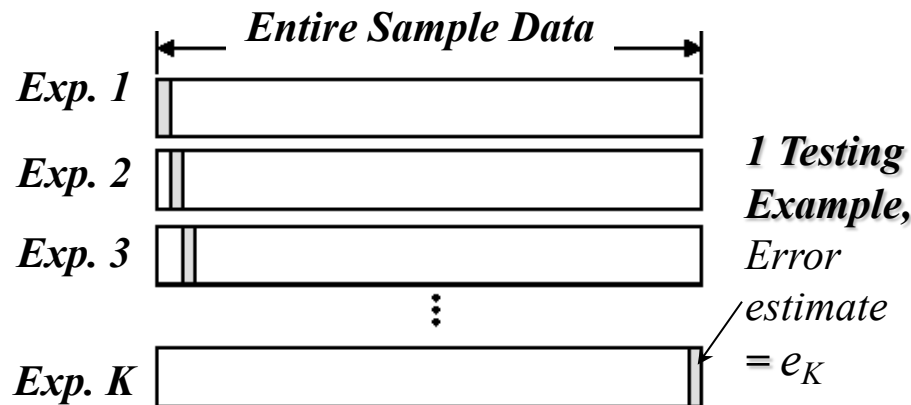
$e_1(\text{Model C})$



# Leave-One-Out (LOO) Cross Validation

Leave-One-Out is the degenerate case of  $K$ -Fold Cross Validation, where  $K$  is chosen as the total number of examples:

- For a dataset with  $N$  examples, perform  $N$  experiments, i.e.,  $N=K$ .
- For each experiment use  $N-1$  examples for training and the remaining one example for testing.

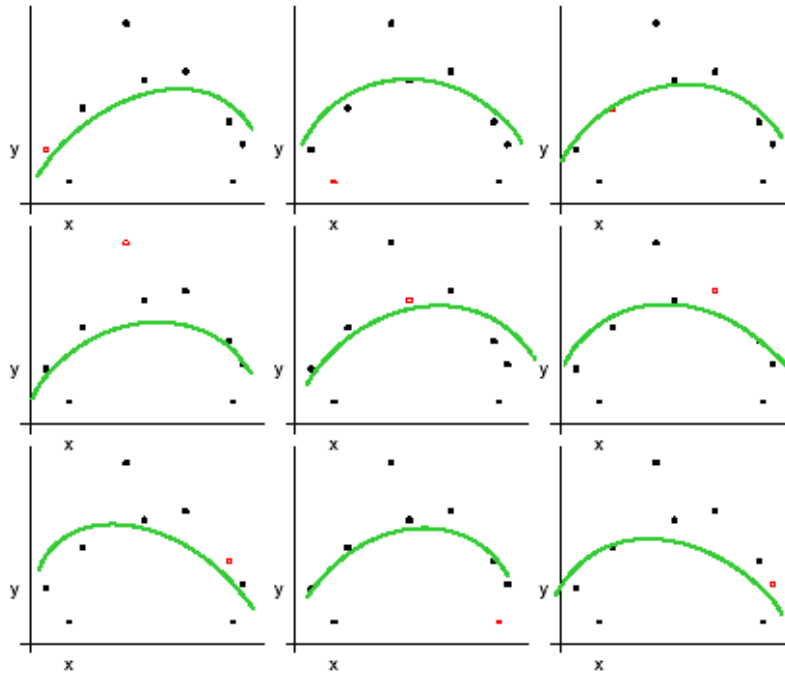


**LOO CV error**

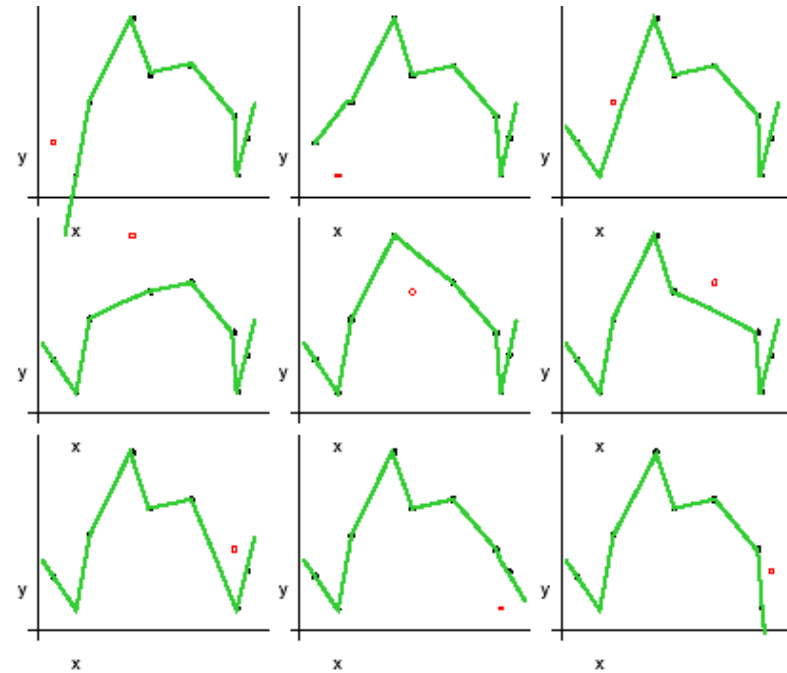
$$= \frac{1}{K} \sum_{i=1}^K e_i$$



# Leave-One-Out Cross Validation



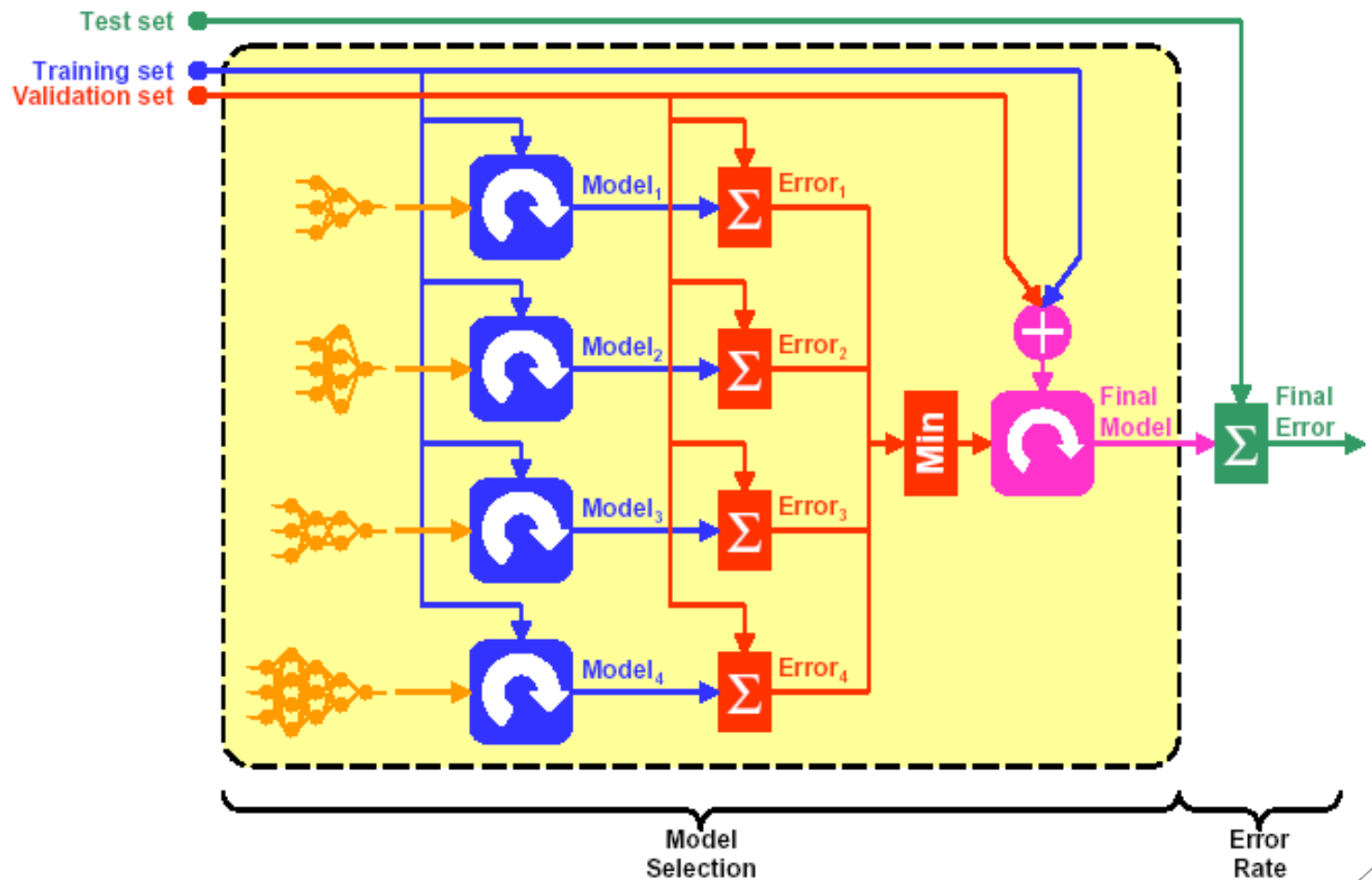
LOOCV Quadratic  
Approximation



LOOCV Piecewise  
Linear Approximation



# Three-Way Data Splits Method



# Three-Way Data Splits Method

If model selection and true error estimates are to be computed simultaneously, the data needs to be divided into three disjoint sets:

- **Training set:** examples for *learning* to fit the parameters of several possible classifiers. In the case of DNN, we would use the training set to find the “optimal” weights with the gradient descent rule.
- **Validation set:** examples to *determine* the error  $J_m$  of different models  $m$ , using the validation set. The optimal model  $m^*$  is given by
$$m^* = \operatorname{argmin}_m J_m$$
- **Training + Validation set:** combine examples used to re-train/redesign  $model_{m^*}$ , and find new “optimal” weights.
- **Test set:** examples used only to *assess* the performance of a *trained model*  $m^*$ . We will use the test data to estimate the error rate after we have trained the final model with train + validation data.



# Three-Way Data Splits Method

## Why separate test and validation sets?

- The error rate estimate of the final model on validation data will be biased (smaller than the true error rate) since the validation set is also used to select in the process of final model selection.
- After assessing the final model, an independent test set is required to estimate the performance of the final model.

**“NO FURTHERING TUNNING OF THE MODEL IS ALLOWED!”**



# Iris dataset

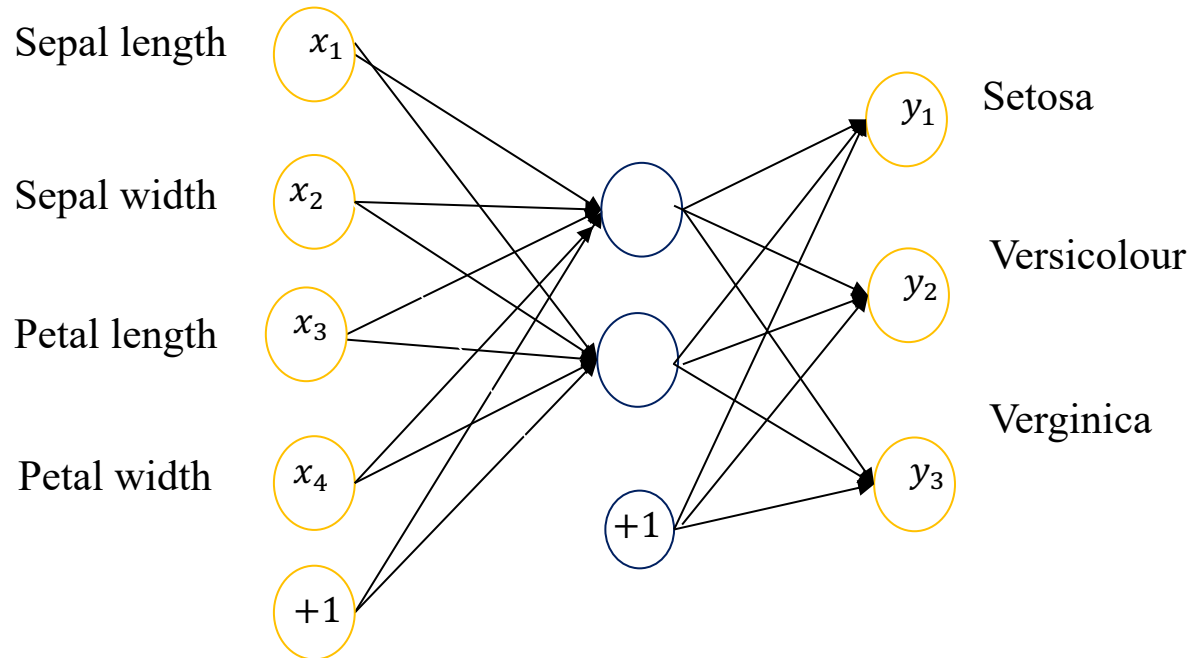
<https://archive.ics.uci.edu/ml/datasets/Iris>

Three classes of iris flower: Setosa, versicolour, and virginica

Four features: Sepal length, sepal width, petal length, petal width

150 data points

FFN with one hidden layer. Determine number of hidden neurons?



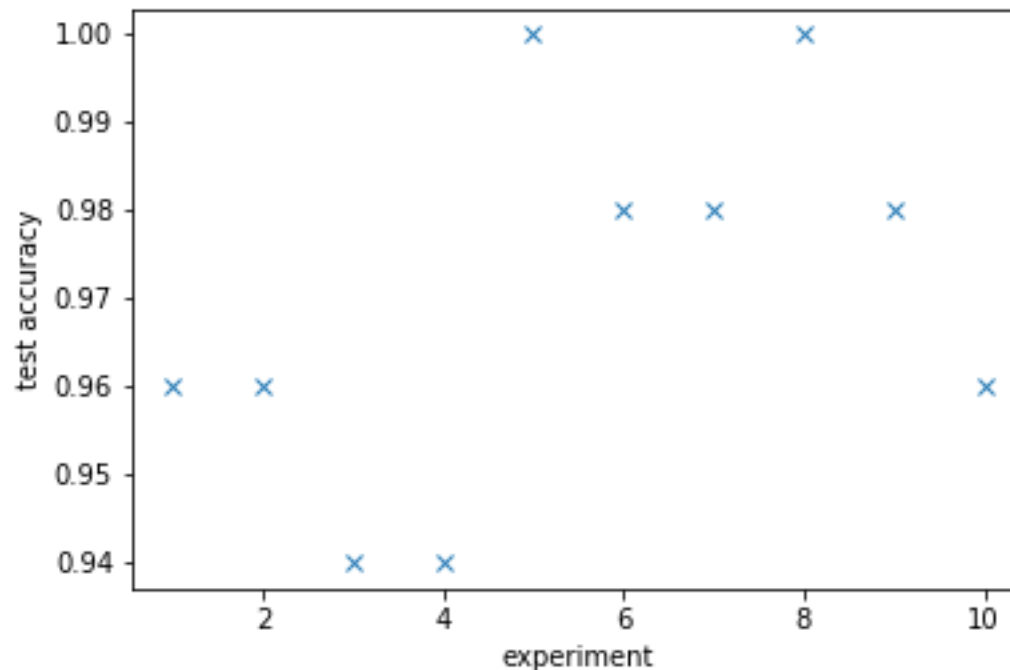


# Example 1a: Random subsampling

150 data points

In each experiment, 50 points for testing and 100 for training

Example: 5 hidden neurons, 10 experiments

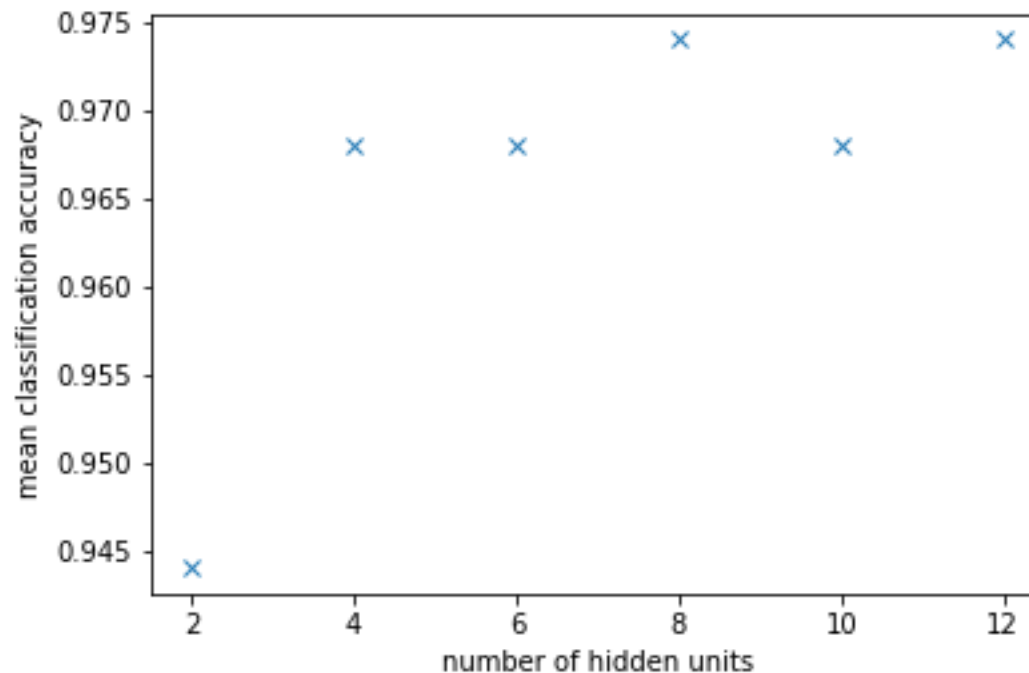


Mean accuracy = 97.0%



## Example 1b

For different number of hidden units, misclassification errors in 10 experiments



Optimum number of hidden units = 12

Accuracy = 97.4%



# Example 2a: Cross validation

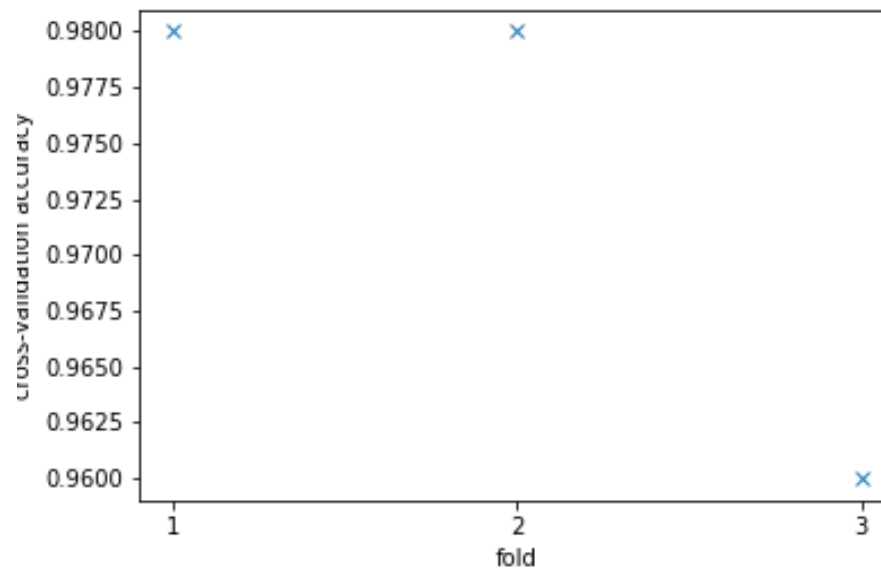
150 data points



3-fold cross validation: 50 data points in one fold.

Two folds are used for training and the remaining fold for testing

Example: hidden number of units = 5

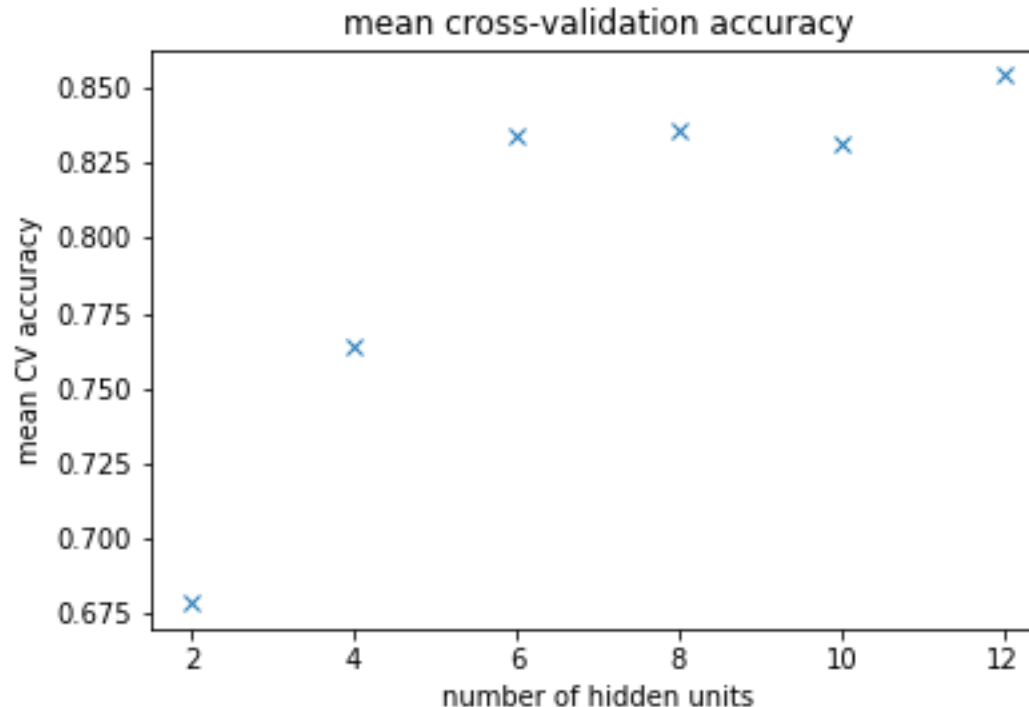


3-fold cross-validation (CV) accuracy = 97.33%



## Example 2b

Mean CV error for 10 experiments for different number of hidden units:



Optimum number of hidden neurons = 12

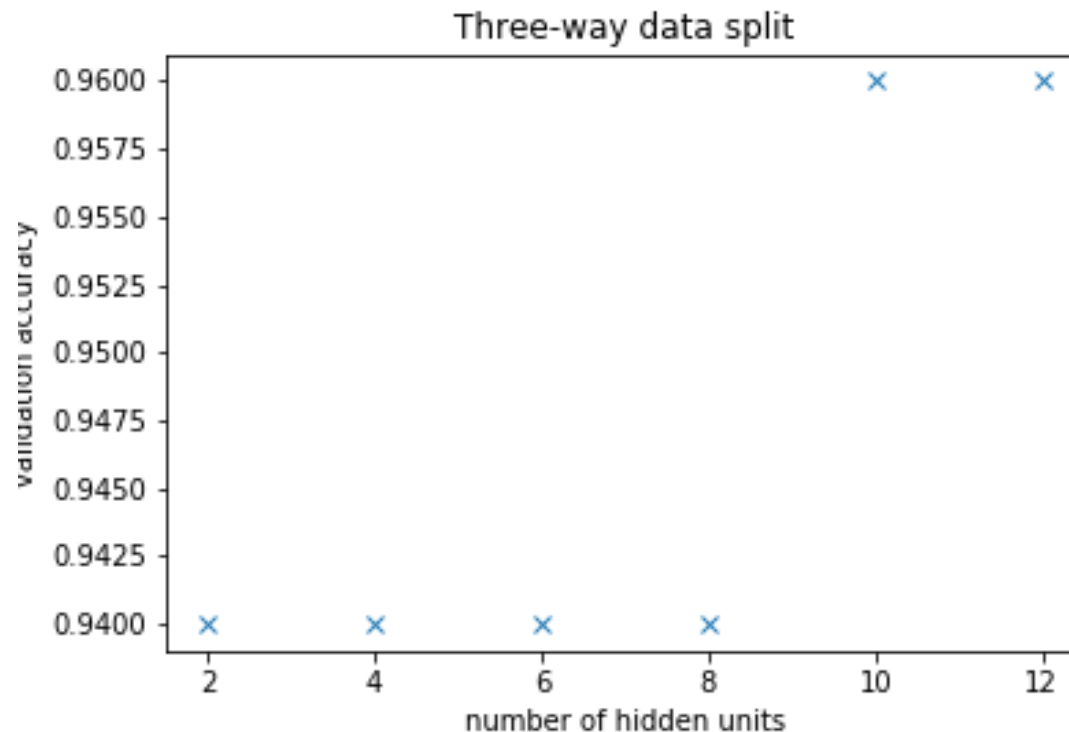
Cross-validation accuracy = 85.4%



# Example 3a: Three-way data splits

150 data points

50 data points each for training, for validation, and for testing.



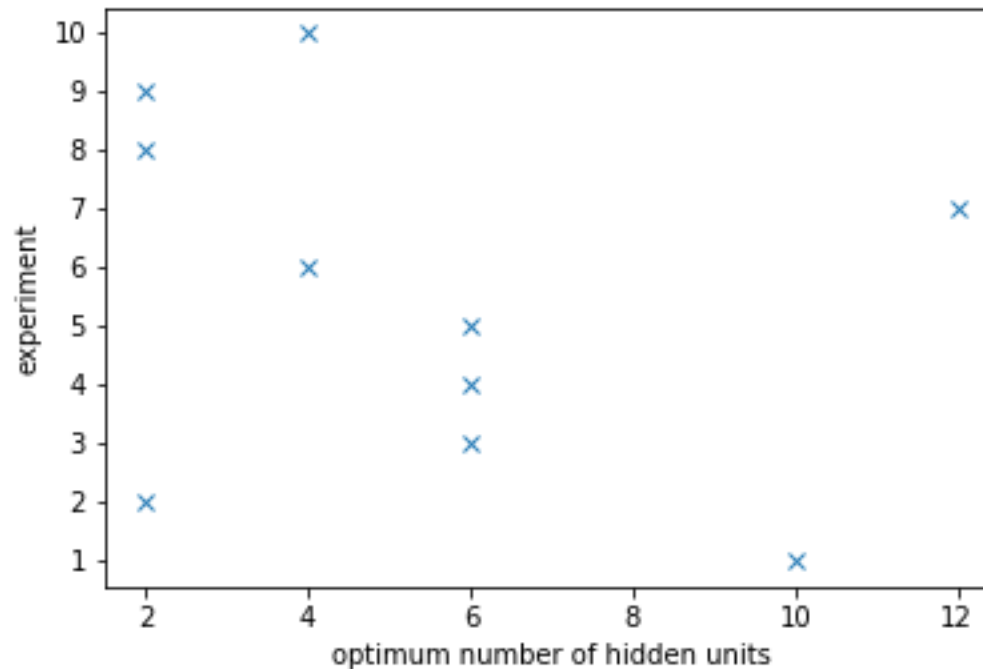
Optimum number of hidden neurons = 10

Accuracy = 96%



## Example 3b

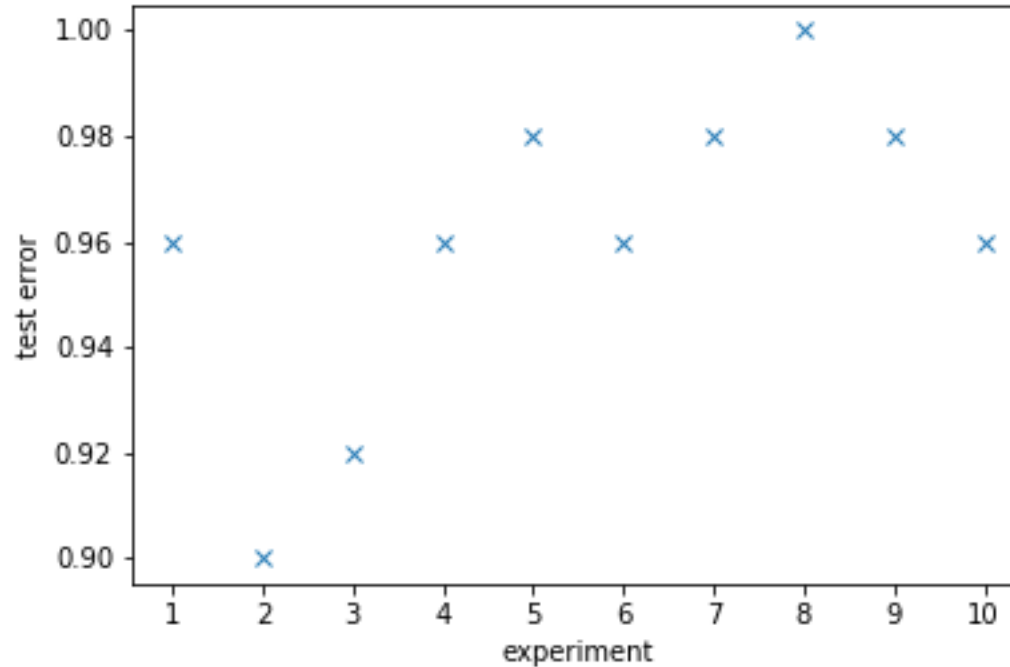
For 10 experiments:



Optimum number of hidden neurons = 2



## Example 3b



Accuracy = 96%  
(mean corresponding to 2 hidden units)



# Model Complexity

**Complex models:** models with many adjustable weights and biases will

- more likely to be able to solve the required task,
- more likely to memorize the training data without solving the task.

**Simple models:** The simpler the model that can learn from the training data is more likely to generalize over the entire sample space. May not learn the problem well.

This is the fundamental trade-off:

- Too simple — cannot do the task, e.g. 5 hidden neurons. (*underfitting*)
- Too complex — cannot generalize from small and noisy datasets well, e.g. 20 hidden neurons. (*overfitting*)





# Overfitting

*Overfitting* is one of the problems that occur during training of neural networks.

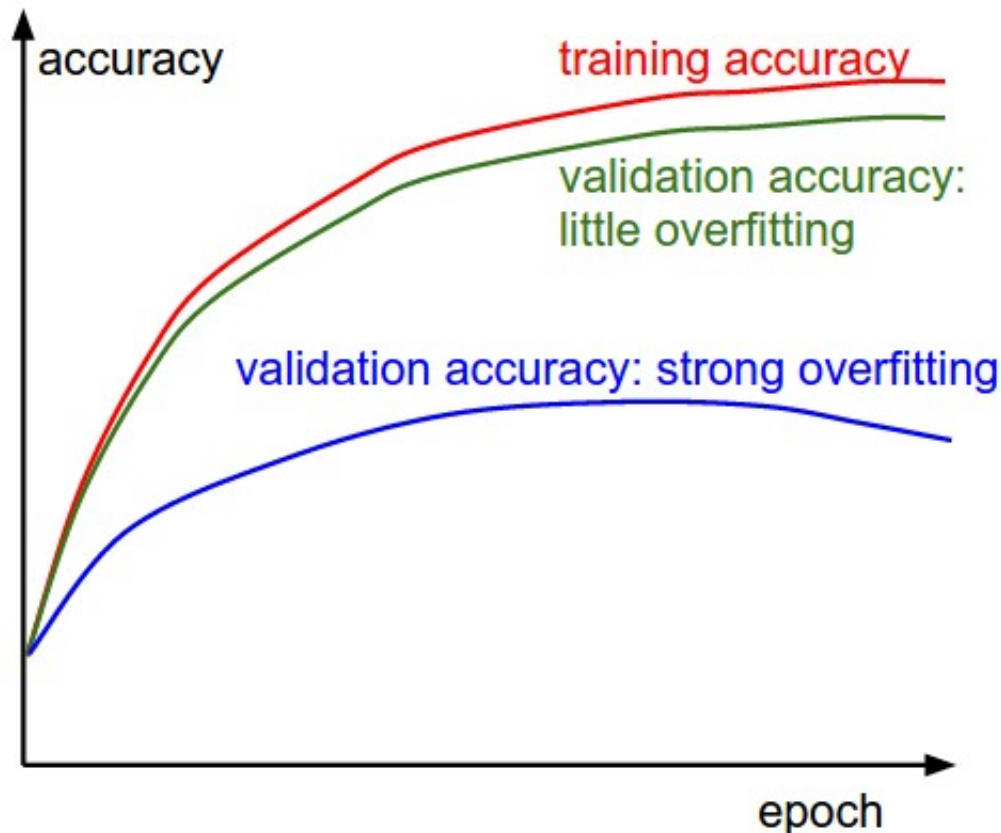
The training error of the network is driven to a very small value at the expense of the test error. The network learns to respond correctly to the training inputs but is unable to generalize to produce correct outputs to novel inputs.

Overfitting occurs when the amount of training data is insufficient or the network has too many parameters to learn.



# Overfitting

Overfitting occurs when the number weights is too large and the network learn training patterns too much.

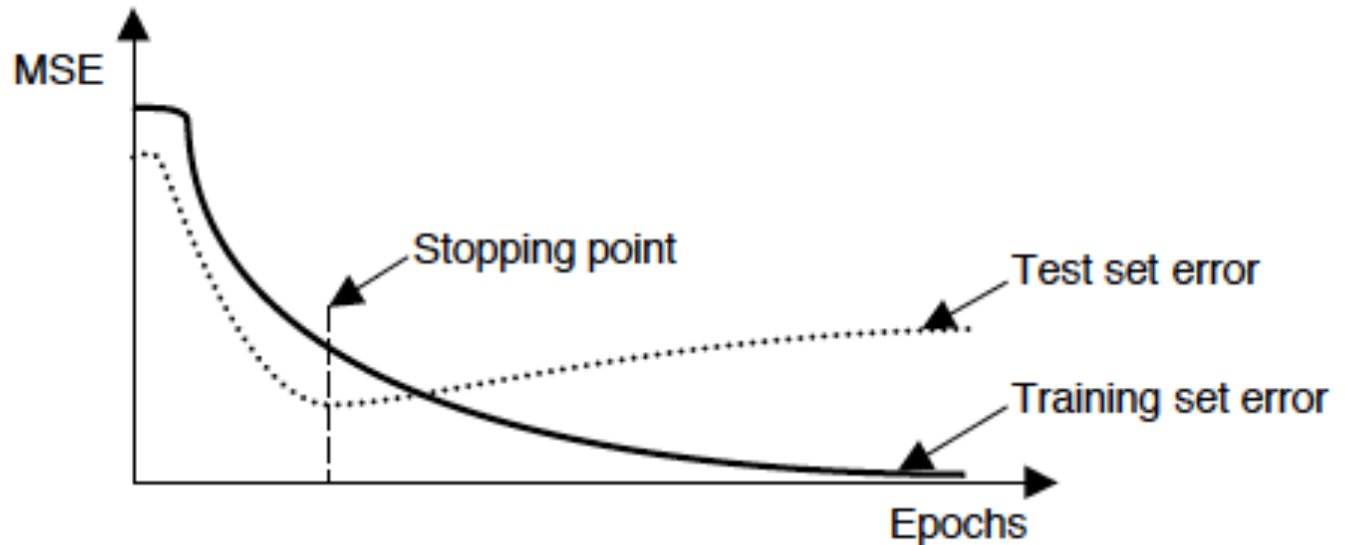


# Methods to overcome overfitting

1. Early stopping
2. Regularization of weights
3. Dropouts



# Early stopping



Training of the network is to be stopped when the test error starts increasing.

Early stopping can be used in test/validation by stopping when the validation error is minimum.



# Regularization of weights

During overfitting, some weights attain large values to reduce training error, jeopardizing its ability to generalize. In order to avoid this, a *penalty* term (*regularization* term) is added to the cost function.

For a network with weights  $\mathbf{W} = \{w_{ij}\}$  and bias  $\mathbf{b}$ , the penalized (or regularized) cost function  $J_1(\mathbf{W}, \mathbf{b})$  is defined as

$$J_1 = J + \beta_1 \sum_{i,j} |w_{ij}| + \beta_2 \sum_{i,j} (w_{ij})^2$$

where  $J(\mathbf{W}, \mathbf{b})$  is the standard cost function (i.e., m.s.e. or cross-entropy),

$$L^1 - norm = \sum_{i,j} |w_{ij}|$$

$$L^2 - norm = \sum_{i,j} (w_{ij})^2$$

And  $\beta_1$  and  $\beta_2$  are known as  $L^1$  and  $L^2$  regularization (penalty) constants, respectively. These penalties discourage weights from attaining large values



# Regularization of weights

Regularization is usually not applied on bias terms.  $L^2$  regularization is most popular on weights.

$$J_1 = J + \beta_2 \sum_{ij} (w_{ij})^2$$

Gradient of the regularized cost wrt weights:

$$\nabla_W J_1 = \nabla_W J + \beta_2 \frac{\partial (\sum_{ij} w_{ij}^2)}{\partial W} \quad (\text{A})$$



# Regularization of weights

$$L^2 - norm = \sum_{ij} (w_{ij})^2 = w_{11}^2 + w_{12}^2 + \dots w_{Kn}^2$$

$$\frac{\partial (\sum_{ij} (w_{ij})^2)}{\partial \mathbf{W}} = \begin{pmatrix} \frac{\partial \sum (w_{ij})^2}{\partial w_{11}} & \frac{\partial \sum (w_{ij})^2}{\partial w_{12}} & \dots & \frac{\partial \sum (w_{ij})^2}{\partial w_{1k}} \\ \frac{\partial \sum (w_{ij})^2}{\partial w_{21}} & \frac{\partial \sum (w_{ij})^2}{\partial w_{22}} & \dots & \frac{\partial \sum (w_{ij})^2}{\partial w_{2k}} \\ \vdots & \vdots & \vdots & \vdots \\ \frac{\partial \sum (w_{ij})^2}{\partial w_{n1}} & \frac{\partial \sum (w_{ij})^2}{\partial w_{n2}} & \dots & \frac{\partial \sum (w_{ij})^2}{\partial w_{nK}} \end{pmatrix} = 2\mathbf{W}$$



# Regularization of weights

Substituting in (A):

$$\nabla_{\mathbf{W}} J_1 = \nabla_{\mathbf{W}} J + \beta_2 \frac{\partial (\sum_{ij} w_{ij}^2)}{\partial \mathbf{W}} = \nabla_{\mathbf{W}} J + 2\beta_2 \mathbf{W}$$

For gradient decent learning that uses regularized cost function:

$$\mathbf{W} \leftarrow \mathbf{W} - \alpha \nabla_{\mathbf{W}} J_1$$

Substituting above:

$$\mathbf{W} \leftarrow \mathbf{W} - \alpha (\nabla_{\mathbf{W}} J + \beta \mathbf{W})$$

where  $\beta = 2\beta_2$

$\beta$  is known as the *weight decay parameter*.

That is for  $L^2$  regularization, the weight matrix is weighted by decay parameter and added to the gradient term.





# Classification of MNIST images

<http://yann.lecun.com/exdb/mnist/>

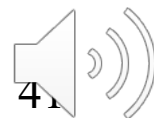


Each image is 28x28 size.

Intensities are in the range [0, 255].

Training set: 60,000

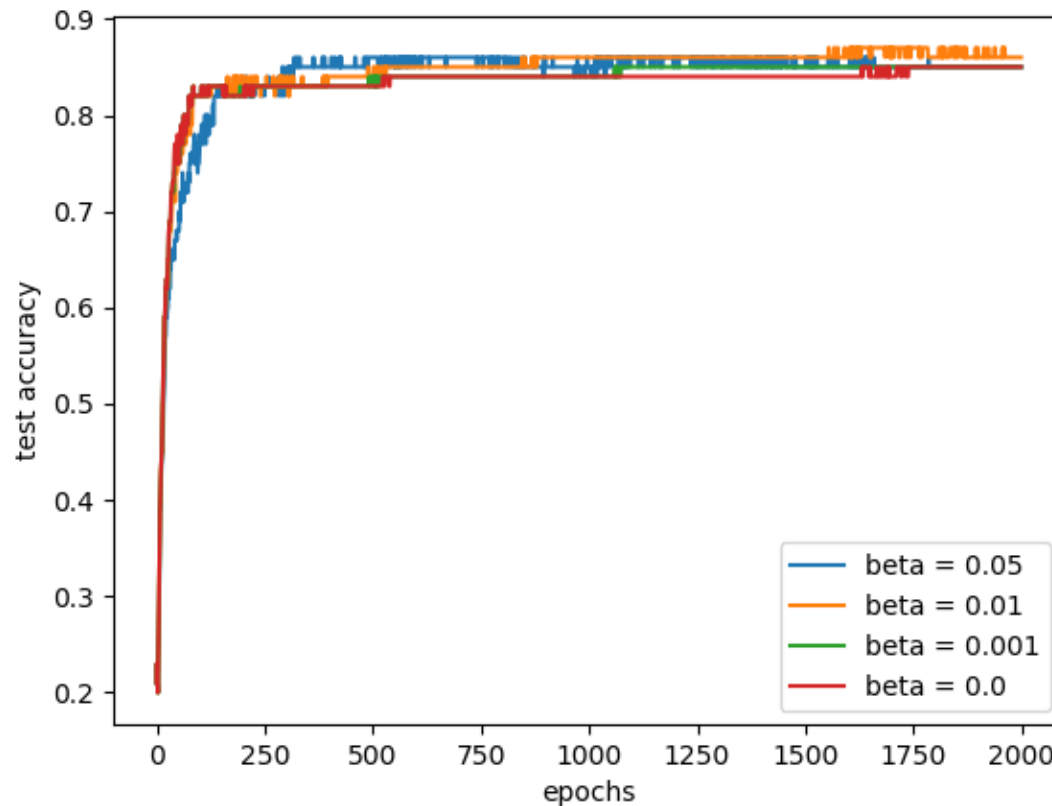
Test set: 10,000



## Example 4: Training with different weight regularization

600 training patterns and 100 test patterns

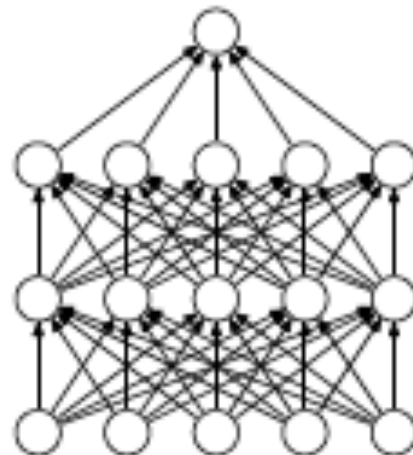
DNN with [784, 625, 100, 10] architecture.



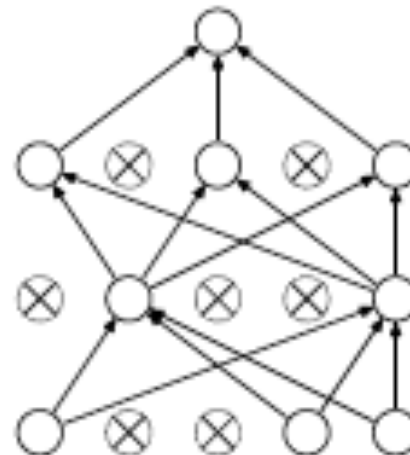
# Dropouts

Deep neural networks with a large number of parameters (weights, biases, etc.) are powerful learning machines. Overfitting can be avoided by training only a fraction of weights in each iteration.

The key idea of ‘dropouts’ is to randomly drop neurons (along with their connections) from the networks during training. This prevents neurons from co-adapting and thereby reduces overfitting.

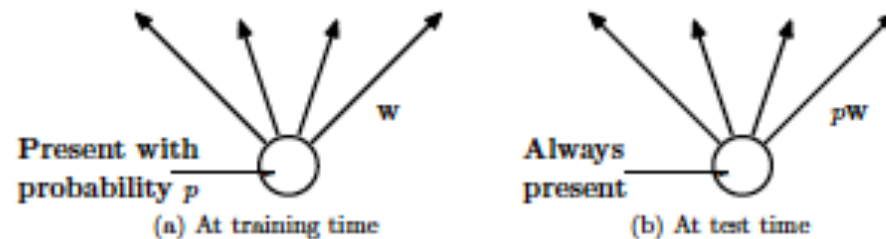


(a) Standard Neural Net



(b) After applying dropout.

# Dropouts



The units (neurons) are presented with a probability  $p$  and presented to the next layer with weight  $W$  to the next layer at the training time.

At test time, the weights are always present and presented to the network with weights multiplied by  $p$ . The output at the test time is same as the expected output at the training time.

Applying dropouts result in a ‘thinned network’ that consists of only neurons that survived.



# Dropouts

Dropout neural networks can be trained in a similar manner to standard networks. The only difference is that in each mini-batch training, we train on a thinned network dropping out units. The gradients of each parameter are averaged over the training cases in each mini-batch



# Example 5: Training with dropouts

