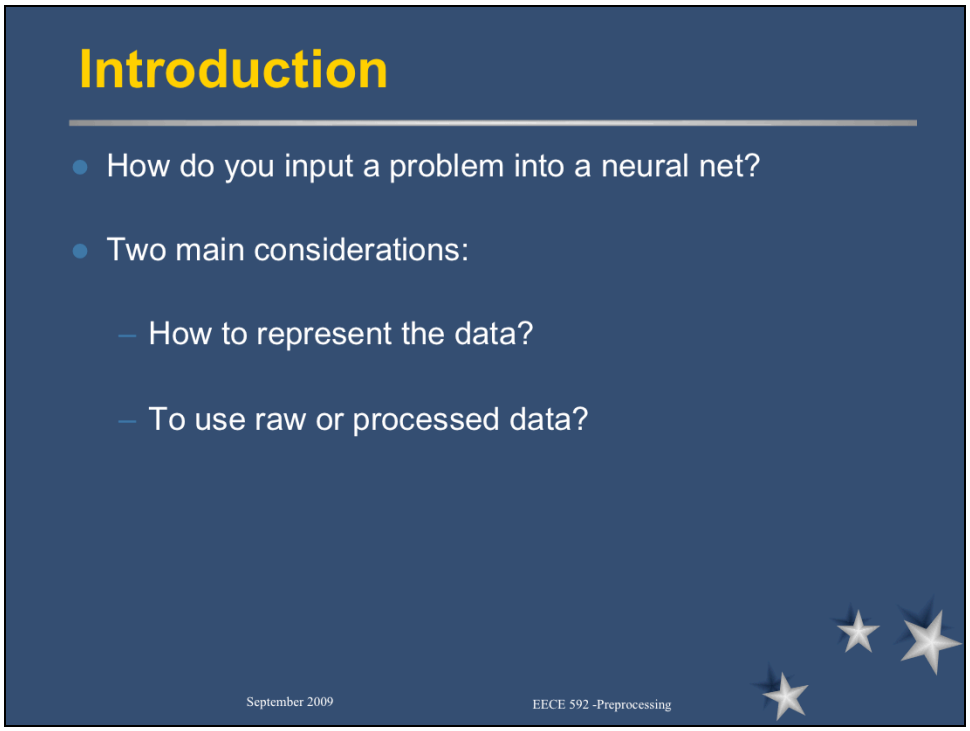
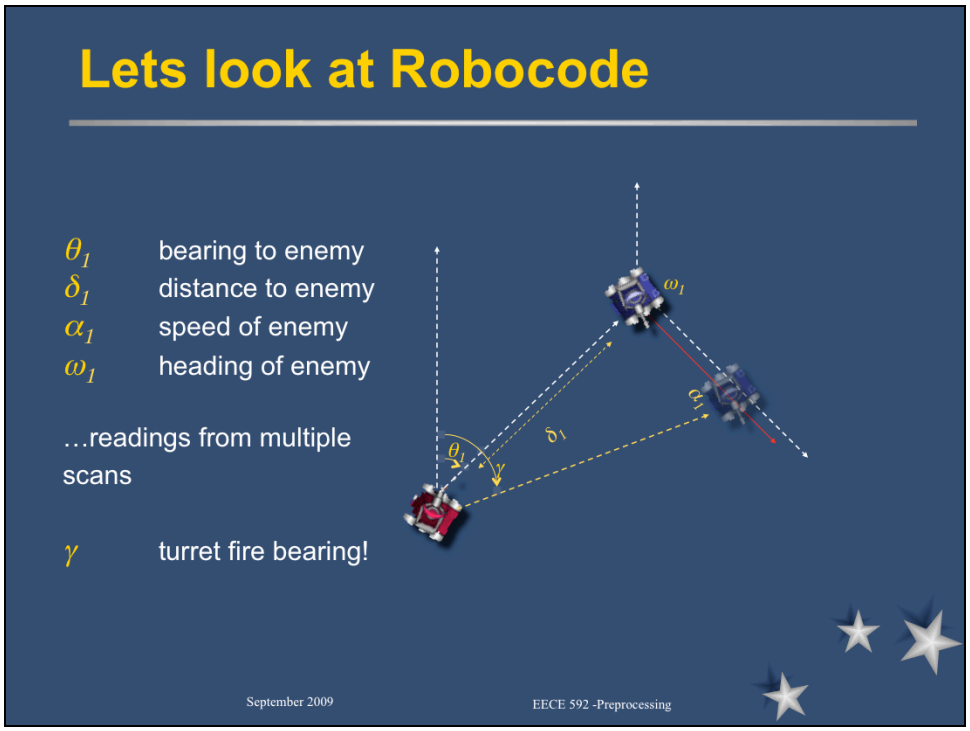

Pre-processing



A neural network is theoretically able to handle raw data. However, without some attempt to present the data in a suitable form, it is unlikely that the training process or the resulting generalization will be its most efficient or most effective.

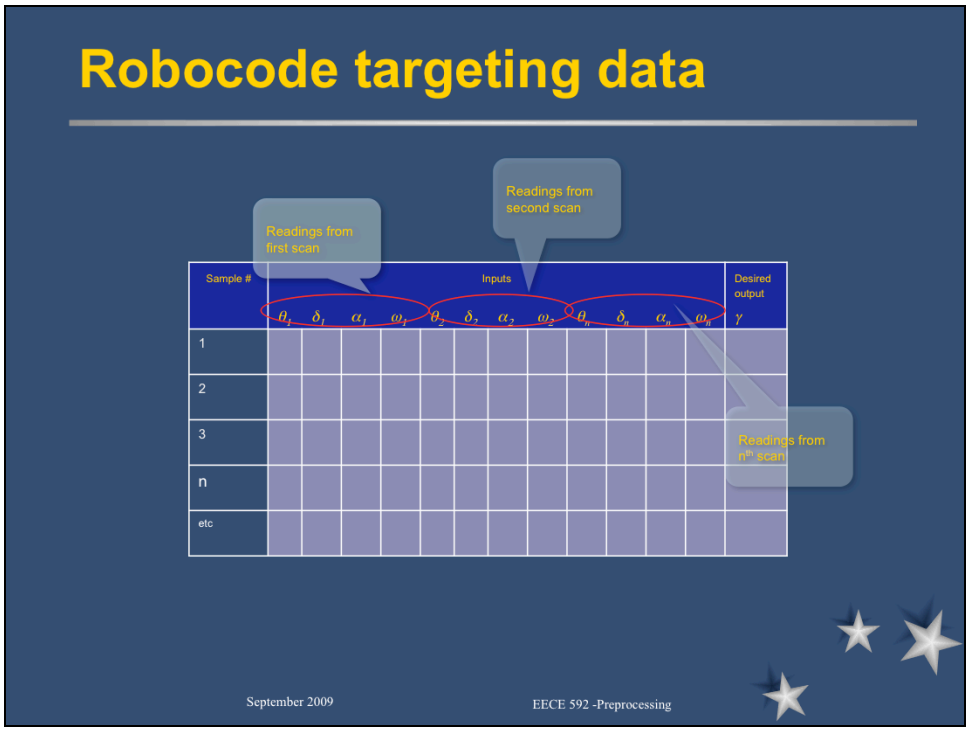
Consider, for example, data taken from a chemical plant. Temperature and pressure values vary greatly in magnitude. If used for learning, this difference in magnitude will have to be accommodated by the weights that develop. This can cause problems since in this case, the difference in magnitudes does not reflect on the relative importance of the values themselves.

There are two basic means by which the input data can be operated upon. One is to choose a suitable representation for the input values. For example should the input values be presented as real values or coded using boolean values? The other is to pre-process the input data in order to extract or emphasize the distribution's most salient features.



Lets look at Robocode

The robocode environment generates a wealth of information. The project assignment suggests that RL is used to help your tank learn to improve its behaviour. For this, you will need to define the "state" of the Robocode environment at any given moment. Lets say RL is being used to figure out whether to attack a tank or not. Here we see the information that might be useful.



Maybe we should scan the same tank multiple times. Perhaps this will help to determine whether the tank is a good candidate to attack or not.

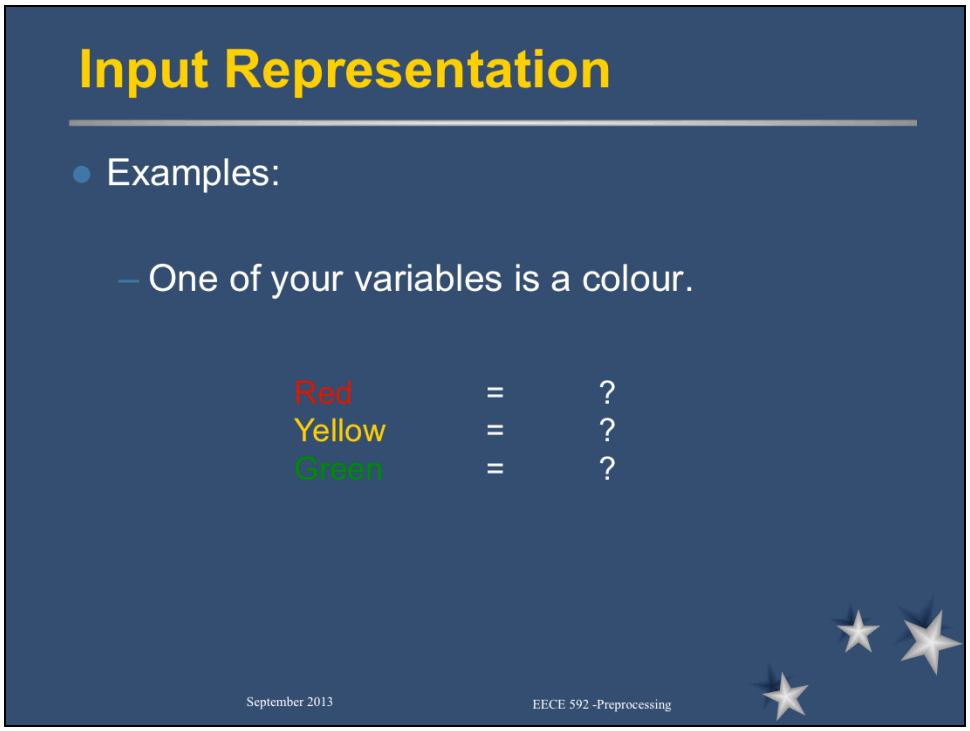
This table helps to show the data that you might collect. The column on the right is the target value to be predicted. The input columns represent multiple sets of readings from multiple scans.

It seems that you wouldn't have to do anything special with distances. But what about bearings. A bearing of 0.0 is actually almost the same as 359 degrees. Is this ok?

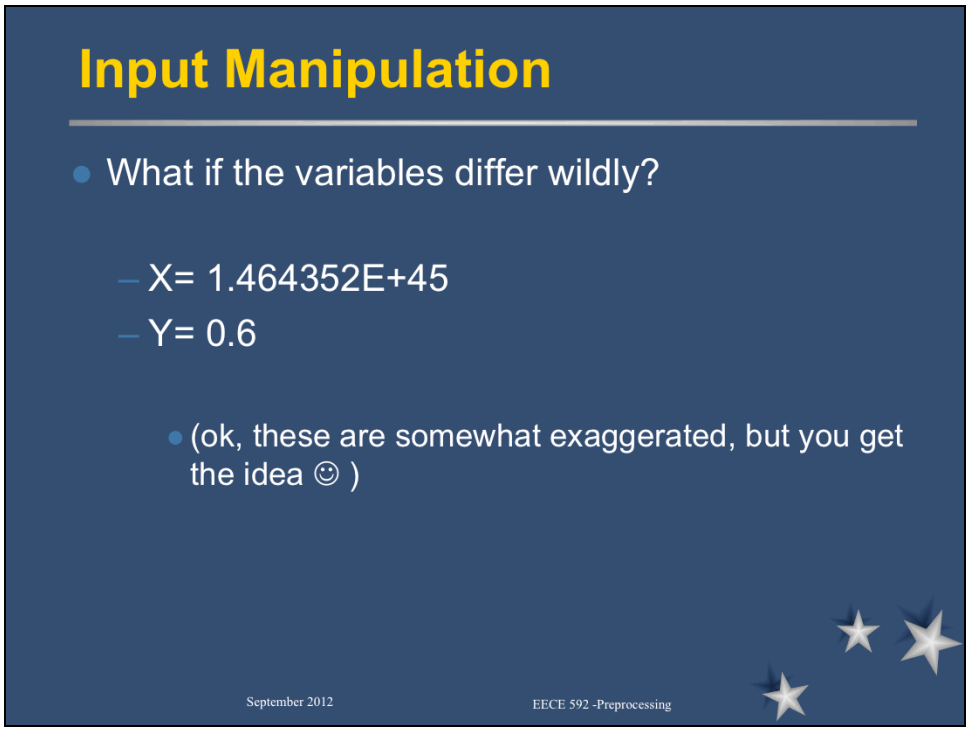
Would it help if the data was mathematically manipulated in some way first?

If we are to use a neural network to approximate the value function, how well is it likely to generalize?

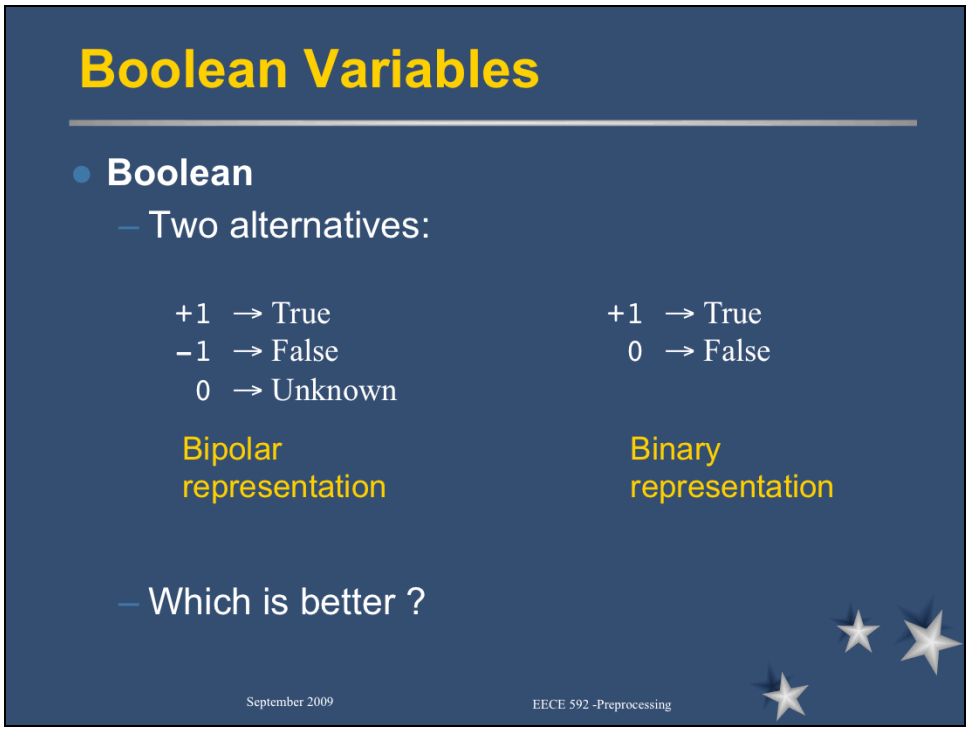
The following slides provides a general introduction to data pre-processing and will help answer some of these questions.



Only numeric values can be input to a neural net. So in this case, how should these colours be represented? Would it be ok to assign values arbitrarily e.g. red = 1, yellow = 2 and green = 3? In fact it wouldn't. This representation would impart some degree of ordinality to the problem which is clearly not present in the natural representation. I.e. green does not really have a "greater" value than red…..except perhaps if these were colours from a traffic light.



There's no reason why variables in the same input pattern could not differ by many orders of magnitude. However knowing that it may significantly slow down a learning algorithm suggests that perhaps these inputs should be normalised before use.

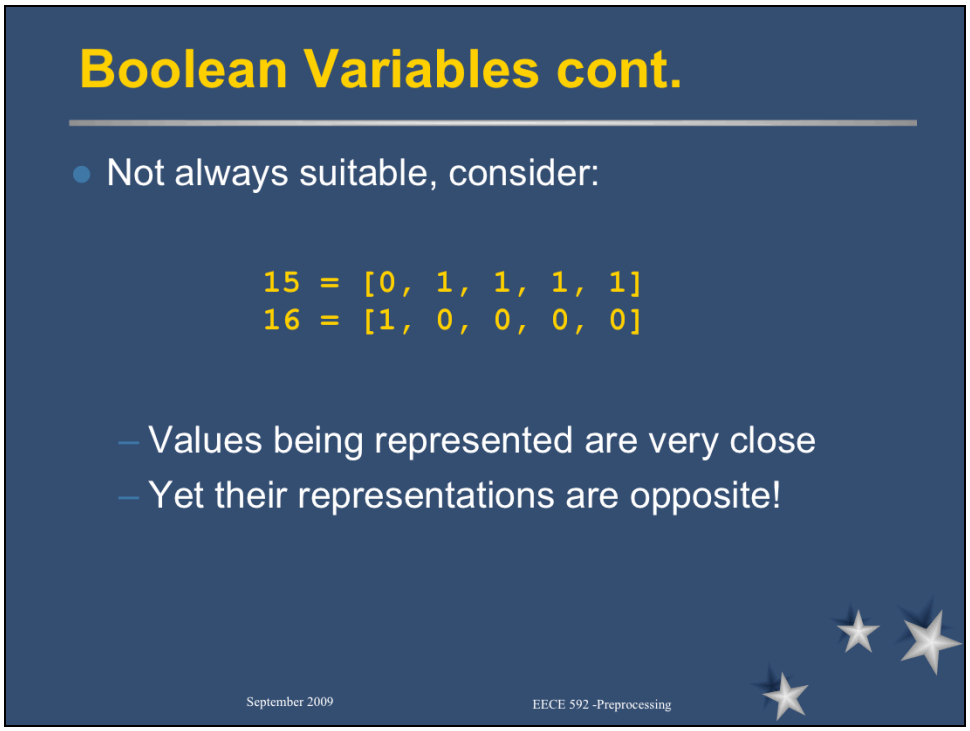


The inputs to a neural net will be in the form of numbers, namely integers or floating points. The meaning and use of such numbers will depend on the particular problem that they are meant to describe.

One of the simplest forms of representation is the boolean. For example in a diagnosis system, an input could signal the presence or absence of a particular symptom. It turns out that when used with neural nets, there are two ways that boolean variables may be presented: the so called, bipolar or binary representations.

The choice of {+1, -1, 0} or {0, 1} models can affect learning algorithms.

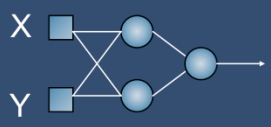
It turns out that, in general, {+1, -1, 0} models learn faster. This is the case with backpropagation training for example because the error used to update the weights, is a function of the input value. When the input value is 0, so is the error. This second alternative, not only provides a 'richer' representation, but should reduce the learning time.



In the above example, although the value to be represented changes by only a small fraction, the boolean representation of it is very different. So in the problem space, the patterns are very close, yet the representations are completely different! This is one reason why nets find learning the parity problem difficult.

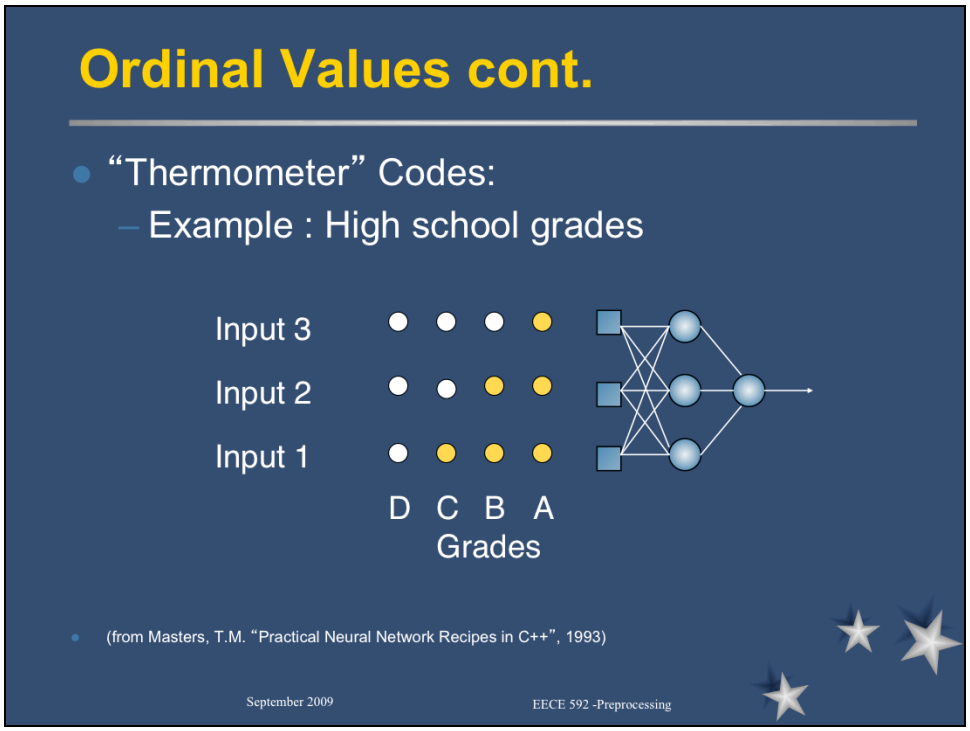
Ordinal Variables

- Representing continuous (integers or float) variables
- Easiest:
  - One variable maps to one neural net input



Typically, continuous or ordinal variables (integers or floating points) represent the magnitude of some quantity such as length, speed or temperature. Values measured on an ordinal scale possess a unique ordering. In most cases, an ordinal parameter can be presented as a single input to a net. This is the simplest way of representing ordinal valued functions, however not necessarily the most effective.

Other more sophisticated schemes exist for representing ordinal values. Typically they involve expressing the value not using a single neuron, but a group. One advantage of increasing the number of neurons that represent a value is that it can facilitate easier learning.



In this case, many inputs (or outputs) are used to code a single ordinal value. The basic idea is:

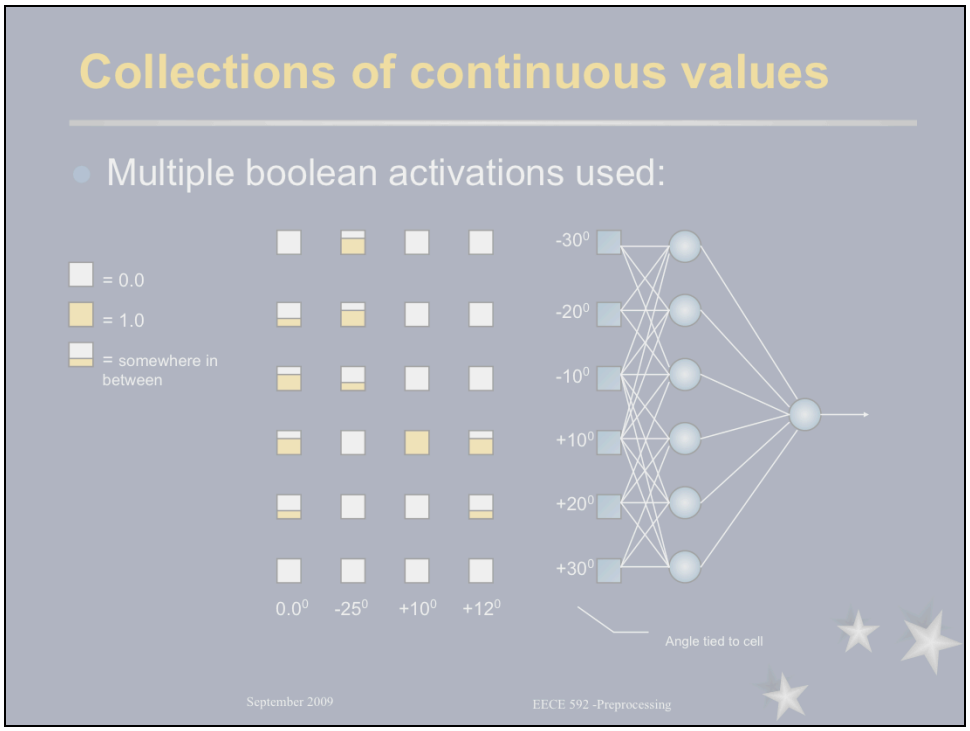
$$[u_1, u_2, \ldots, u_k] = \text{True iff } x \geq [\theta_1, \theta_2, \ldots, \theta_k]$$

{Here "iff" means if and only if}

The inputs (or outputs) are still boolean, however, as the magnitude of the ordinal being represented increases, the number of 'true' activations also increases, much like a thermometer as temperature rises.

Any arbitrary precision can be obtained by varying the number of discrete cells (k) used. Each activation effectively quantizes the value.

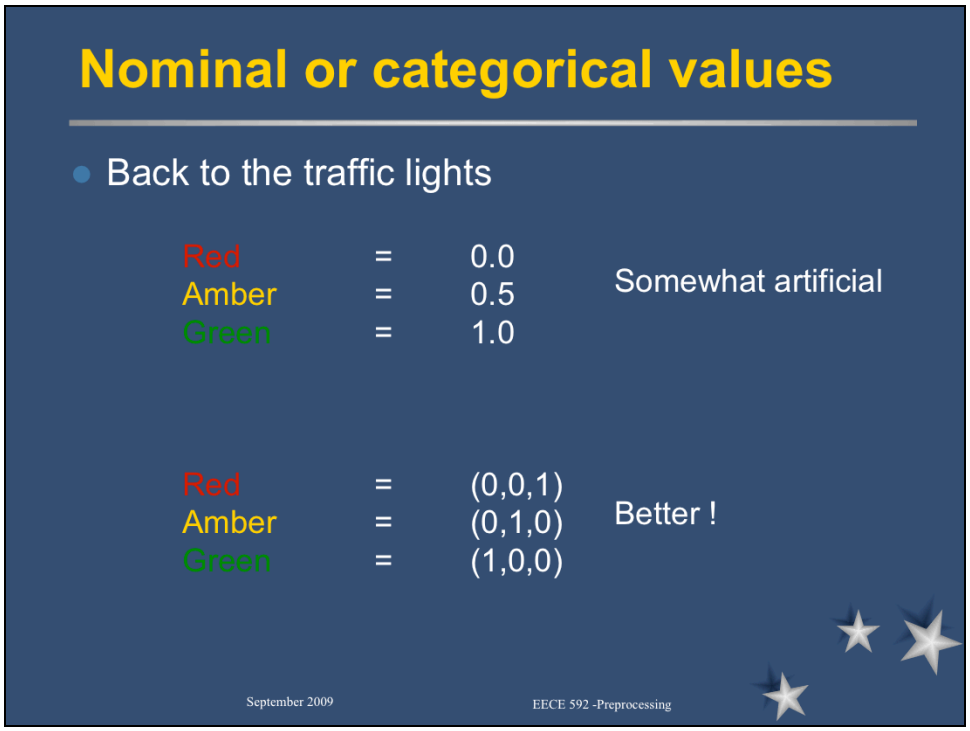
Note that the interval need not be even. It might be beneficial to cluster cells around some desired value. (e.g. normal distribution)



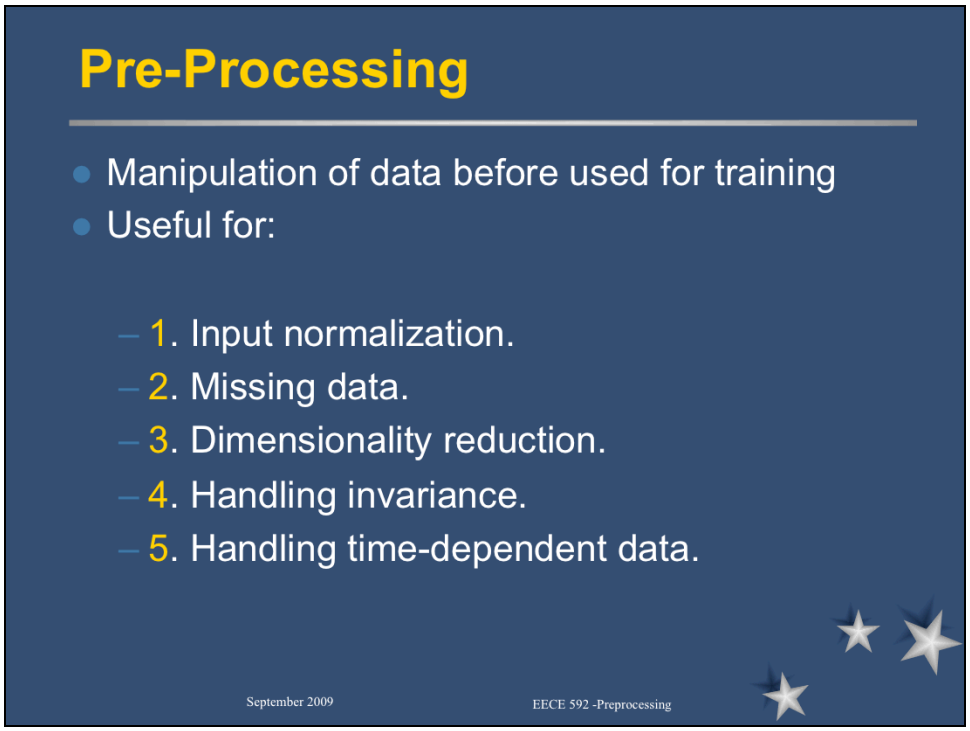
Ordinals can also be represented using a collection of continuous valued activations. This is similar to the thermometer codes except that the different components of the representation are actually tied to differing values in the number space being represented and the activations are no longer boolean but continuous.

One reason why approaches such a thermometer codes or collections of continuous cells work, is that they simply increase the number of learning sites within the neural net (I.e. the number of weights).

Also, the representation used can be designed to better reflect the actual nature of the problem. For example in a distribution in which a certain variable is clustered about a close range of values, to help the neural net differentiate, it may be advantageous to use a large number of inputs to represent that region of input space.

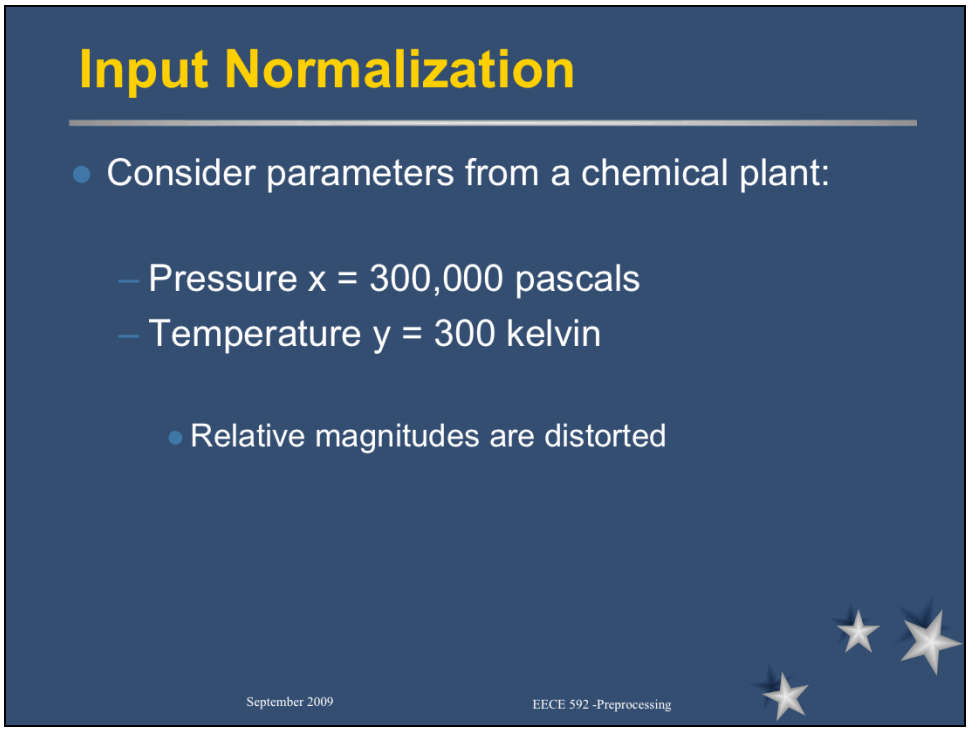


The use of ordinal values for coding input data is not always the most logical. For example, one way of encoding input data to represent the value of a traffic light may be to say 0.0 is green, 0.5 is amber and 1.0 is red. However, this is placing an unnecessary asymmetry on the problem, simply because the magnitude of each 'colour' is different. Learning would tend to accommodate such a variance even if it was artificial. In this case it would be better to have three boolean inputs; e.g. (1 0 0) for green, (0 1 0) for amber and (0 0 1) for red. Thus each input represents a category of input.



Here the term *pre-processing* means to perform some (mathematical) manipulation on the input data before using it for training. The following are several reasons why pre-processing may not just be advantageous, but also necessary:

1. Input normalization.
2. Missing data.
3. Dimensionality reduction.
4. Handling invariance.
5. Handling time-dependent data.



In some applications, it may be advantageous to normalize the input values. Consider an application that takes as its input two measures, the temperature and pressure in a chemical plant. The relative magnitudes of these parameters may be so far removed that the relative importance of the measures is distorted.

Again, a non-linear multi-layer perceptron would be expected to cope but the final weights would themselves differ greatly in magnitude.

By applying simple linear normalization, input values can be re-scaled so that they are all of similar magnitude.

Input normalization

- Simple normalization can be applied:

$$\tilde{x}_i^n = \frac{x_i^n - \bar{x}_i}{\sigma_i}$$

  - This leads to variables with zero mean and unit variance

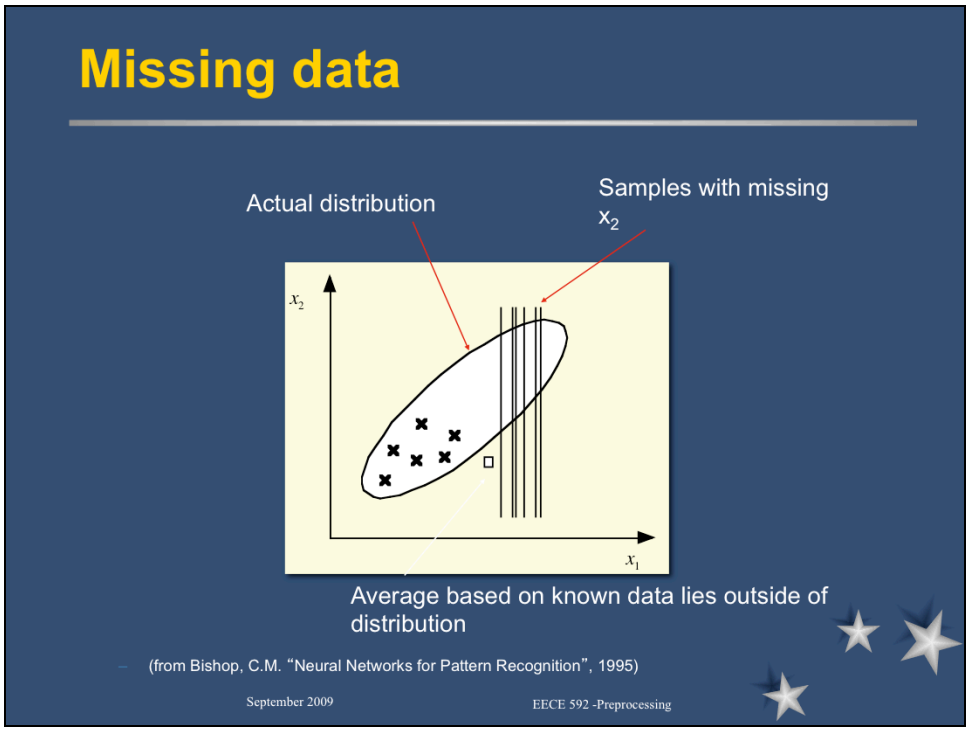
- Why is this useful ???

Mean: $\bar{x}_i = \frac{1}{N}\sum_{n=1}^{N} x_i^n$

Variance: $\sigma_i^2 = \frac{1}{N-1}\sum_{n=1}^{N}(x_i^n - \bar{x}_i)^2$

Normalized variables: $\tilde{x}_i^n = \frac{x_i^n - \bar{x}_i}{\sigma_i}$

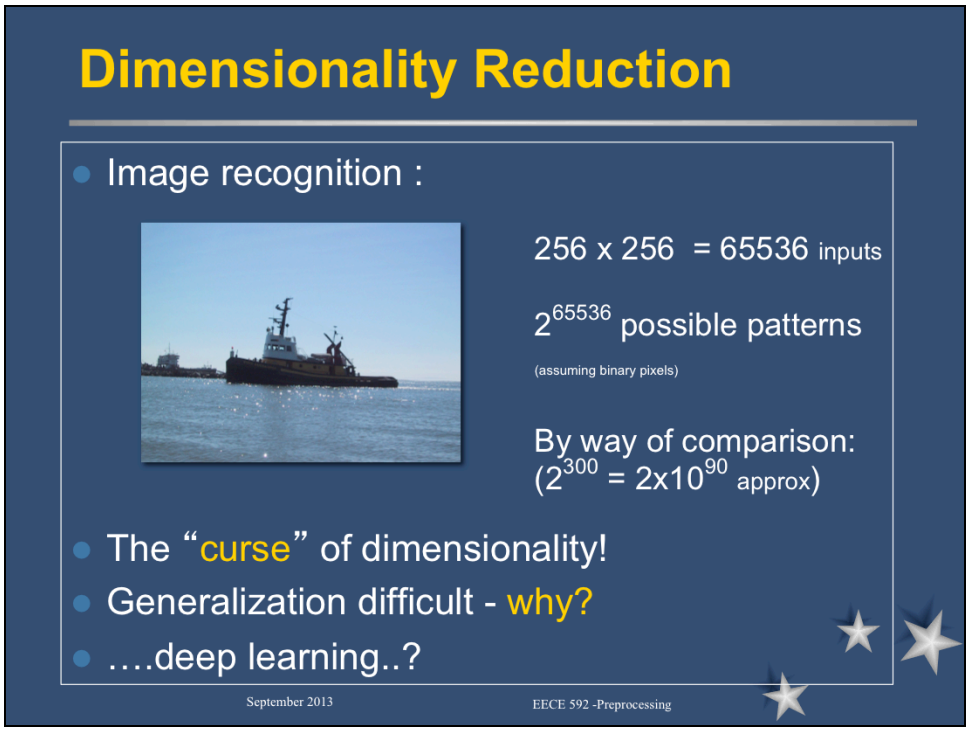
The transformed input variables $\tilde{x}_i^n$ have zero mean and unit variance over the training set. Subsequently, the weights that develop after training will likely also be of unit magnitude. Before learning commences, all weights in a multi-layer perceptron are assigned random initial values. Knowing the range of their final values is helpful when selecting an approach for this initialization, i.e. the initial values can be made the same order of magnitude as their final values. This linear re-scaling can be applied independently of target values, although typically the target values should themselves be re-scaled.



In practical applications, it may be necessary to deal with incomplete input patterns. I.e. input vectors with one or more variables missing. When there are a large number of patterns compared to the missing data, then it is often satisfactory to discard the incomplete patterns. This cannot however be done if the input set is small or if the removal is likely to alter the data distribution. In this case, the missing data must be 'filled in'.

One way to do this is to simply obtain the average value of the missing variable over all other patterns. However, this can generate poor results as the diagram below illustrates. The empty square (the average value of x_2 over the remaining patterns) would not fall in the distribution when used to substitute the missing data shown by the vertical bars.

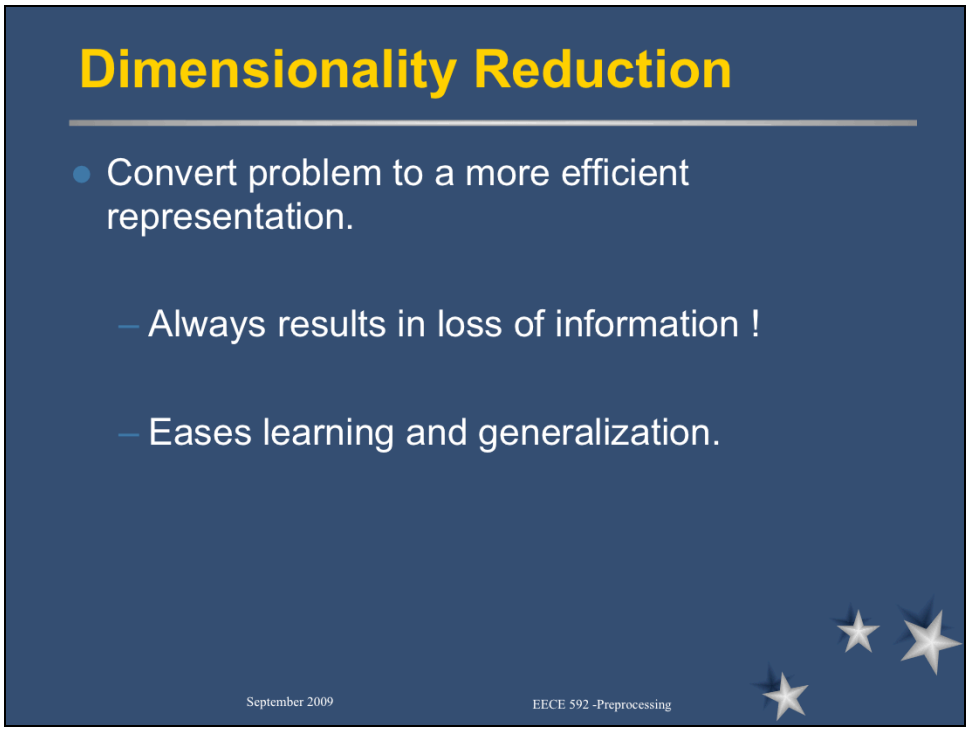
A better solution involves modelling the input distribution itself and then filling missing variables with data drawn from this distribution.



A multi-layer perceptron with non-linear activations can in principle perform any arbitrary mapping of inputs to outputs. In practice however, it is usually beneficial to perform some kind of preprocessing.

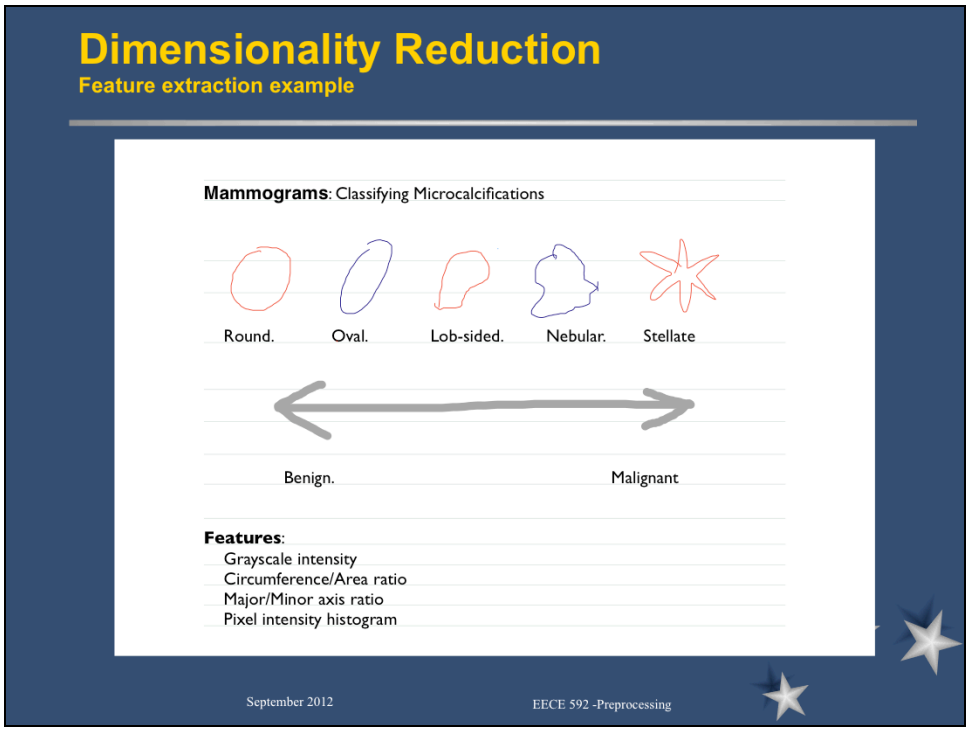
For example, in a recognition problem involving a 256 x 256-pixel input, the neural net would have 65536 inputs. The units in the first layer of the network would each have 65536 weights. Typically, as the numbers of adaptive patterns grow, so it becomes more and more difficult to satisfactorily constrain the problem. Generalization will suffer as a result. (Aside: with binary inputs, this problem represents a total of 2^{65536} possible different patterns! To attain suitable generalization, one would expect to use a representative sample that itself might be excessively large.)

Note that recent (2009-2013) breakthroughs in deep learning have made it possible for neural networks to perform recognition with these amounts of data!



Dimensionality reduction involves suitably reducing the number of inputs and doing so can significantly improve both the learning and generalization. In the above example, a simple form of dimensionality reduction would be to use pixel averaging. This of course would only work so long as the details necessary for successful categorization remained larger than the size of the averaged pixels. As such, dimensionality reduction always incurs a loss of information. The goal is to not lose useful information.

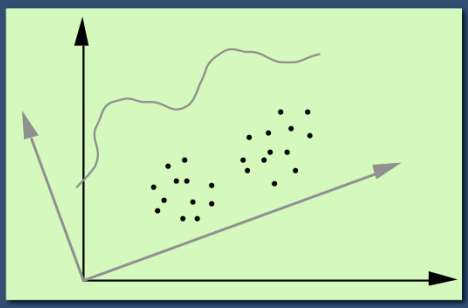
In an image recognition application, dimensionality reduction typically takes the form of feature extraction. Instead of using the raw 256 x 256 input, the image is first preprocessed to extract features appropriate to the task. If done correctly, a reduction in dimensionality doesn't always mean a loss in information.



For example, in classifying micro-calcifications in mammograms, it might be better to use features that relate to the circumference/area ratio, major axis/ minor axis ratio and gray scale intensity rather than the raw images themselves.

Principal Component Analysis (PCA)

- PCA is a method for dimensionality reduction
- PCA attempts to preserve as much relevant information as possible

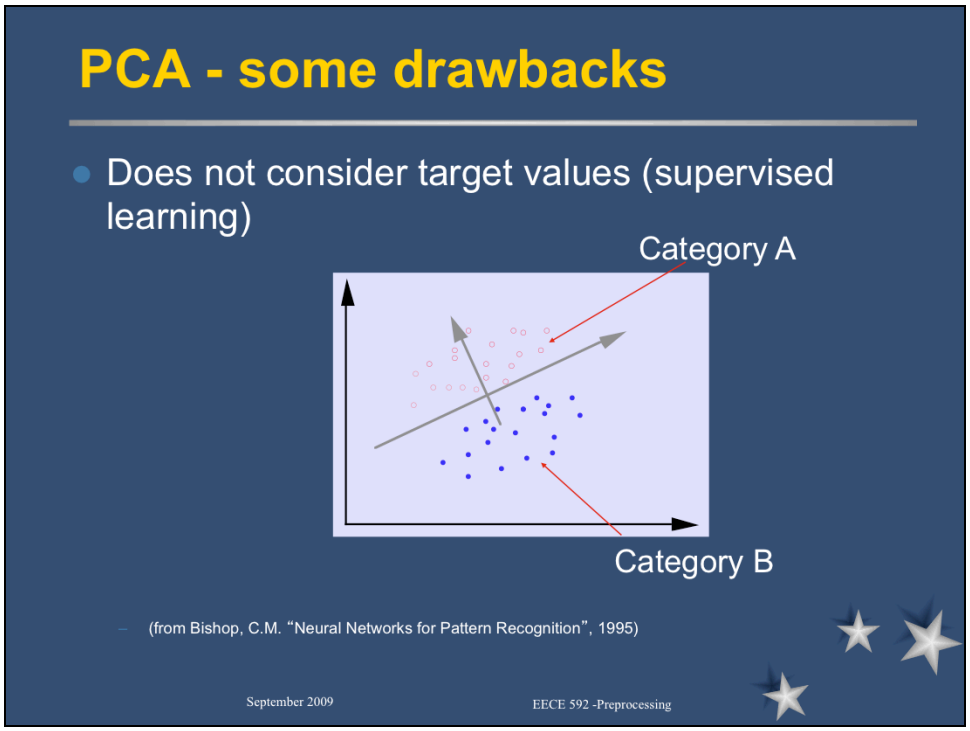


– (from Bishop, C.M. "Neural Networks for Pattern Recognition", 1995)

PCA is a method of dimensionality reduction based upon finding the vectors (principal components) along which the greatest variance of a distribution is observed. Dimensionality reduction is almost always accompanied by a loss of information. PCA works because it attempts to preserve as much of the relevant information as possible.

The diagram shows two distributions that are more easily separated using the first principal component than the original two Cartesian axes. Principle components are always mutually orthogonal. The data points are now represented by a single value being the projection onto the principle axis rather than two x,y coordinates.

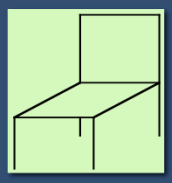
With higher dimensionality space it may be necessary to extract the first few principle components.



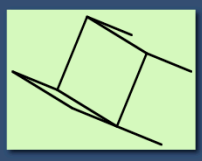
A drawback with PCA is that it considers only the input distribution and not any related output or target values. As such, the approach is regarded to be sub-optimal. The diagram below shows an extreme example in which the first principle component loses all discriminatory powers for the distributions belonging to two different classes. All data points fall within the same range of projection. Instead, the second component would provide the best discriminatory ability.

Handling Invariances

- Invariances could be due to translation, scale, rotation, perspective or many other factors.
- E.g. 3-D object recognition:

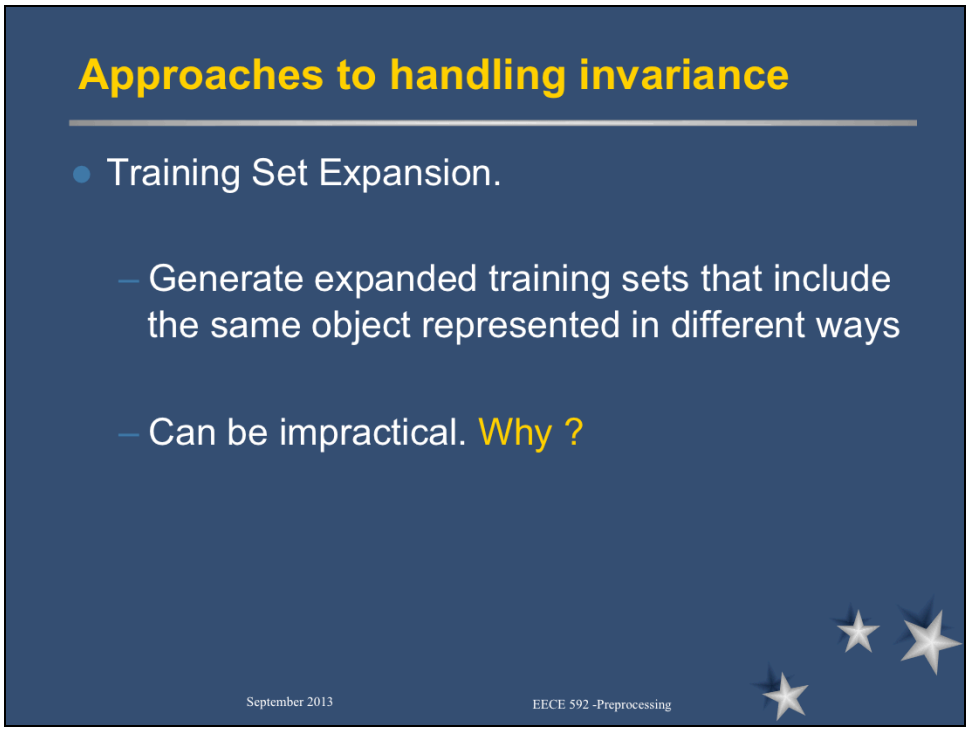


A chair



Still a chair !

In object recognition problems it is often the case that classification should be invariant to any changes due to translation, rotation or scale that the image may have undergone. For example in character recognition problem, the character may not always be the same size or centred in a known position in the image.



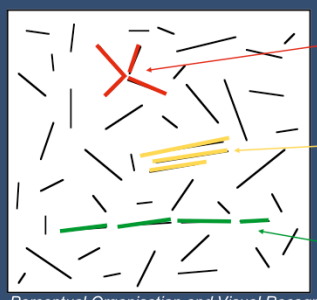
There are a number of ways to address the problem of invariance:

By subjecting the pattern to all expected invariance, it is possible to generate an expanded training set that includes the same object presented in different ways. Theoretically it is possible to train a net with such an expanded training set, in the hope that generalization will accommodate the invariance. However, such an approach is often not practical for all but the simplest of invariance. For example, when attempting to recognise three-dimensional objects in a two-dimensional image, the training set must attempt to address rotation of the object in three dimensions. It's obvious that such an approach will lead to impracticably large training sets. Furthermore, its difficult to get access to large amounts of labelled data.

Deep learning note: DL actually does need very large numbers of unlabelled images as part of its training.

Approaches to handling invariance

- Use invariant input data
- E.g. 3-D object recognition
  - Extract features that are 3-D invariant



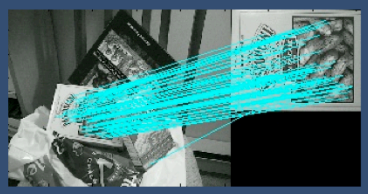
- from: Lowe D.G. (1985). *Perceptual Organisation and Visual Recognition*., Kluwer Academic Publishers.

A third approach to handling invariance involves the choice of a pre-processing function that itself accommodates the necessary invariance. Any network trained with the features of this function will of course exhibit invariant classification. For example, for three-dimensional object recognition, certain object features remain invariant to rotation in three dimensions. E.g. parallel lines observed on the object will remain parallel throughout most of a rotation. This approach was used by Lowe, D.G. for a (non-neural net based) object recognition system.

A more mathematical approach to generating invariant features involves finding the *moments* of the original input data.

SIFT

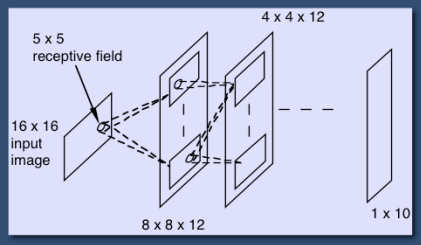
- Scale Invariant Feature Transform
  - Lowe D.G. (CS @ UBC) 1999
  - Goal: Invariance to:
    - Perspective
    - Affine distortion
    - Rotation
    - Scale
    - Illumination



See http://www.cs.ubc.ca/~lowe/keypoints/ for details or search for "SIFT" in Google. The algorithm is quite popular.

Approaches to handling invariance

- Weight Sharing (a.k.a. convolution)
  - Good for translation invariance
  - Used by Yann LeCun *et al*. for character recognition



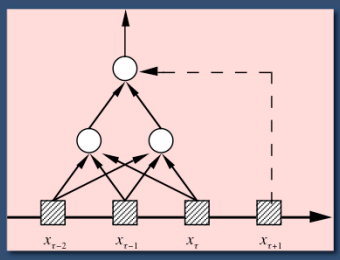
  - And more recently in conjunction with deep learning.

A second approach is based on 'weight sharing'. Consider the above character recognition problem. Take a trained net capable of recognising a character centred at a known position. Then physically repeat that network to cover all other possible centres in the image. This will result in a larger structure that by definition handles translation invariance. Such an approach was used by Le Cun *et al*. for just such an application. Actually, Le Cun was also able to handle small invariances due to size, rotation and style as would be expected from different individuals. However, it could not, for example, recognise characters written upside down.

Note: Neural nets that use weight sharing are also now-a-days known as convolution neural nets and have been quite prominent in the application of deep learning to image recognition.

Time Series Prediction

- Prediction of a future value based upon a series
  - (e.g. stock market prediction).



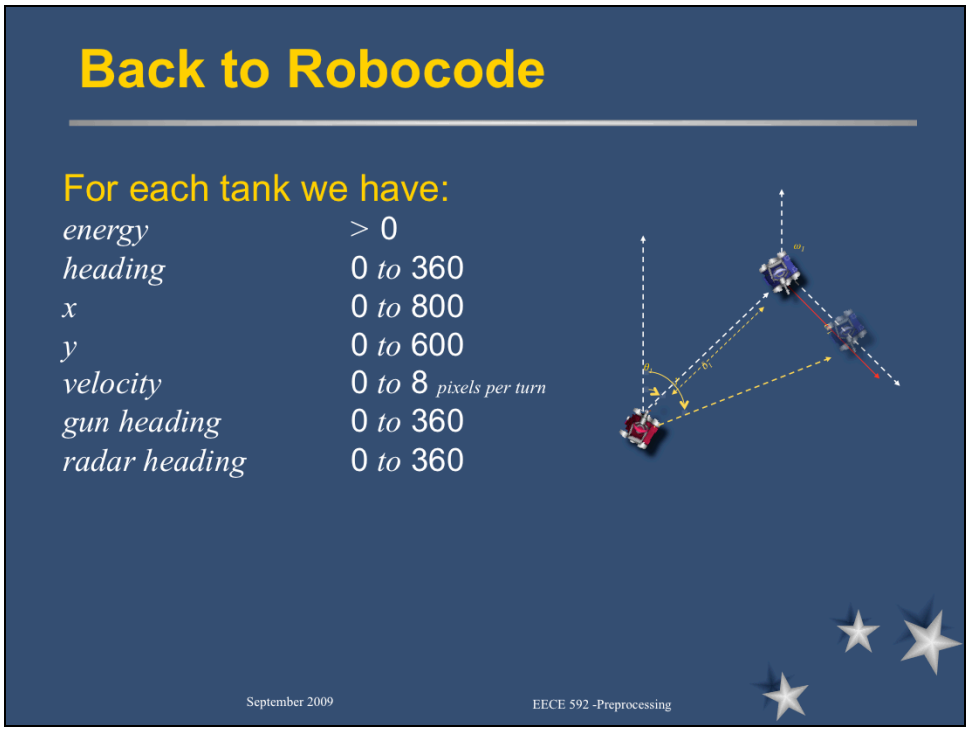
- (from Bishop, C.M. "Neural Networks for Pattern Recognition", 1995)

Given data which varies as a function of time, e.g. , $x(\tau)$ it is possible to use a neural net to attempt to predict the value of x some time into the future.

When using feed-forward multi-layer perceptrons, one of the easiest ways to achieve this is to sample the data at regular intervals over a certain period. This generates a set of values

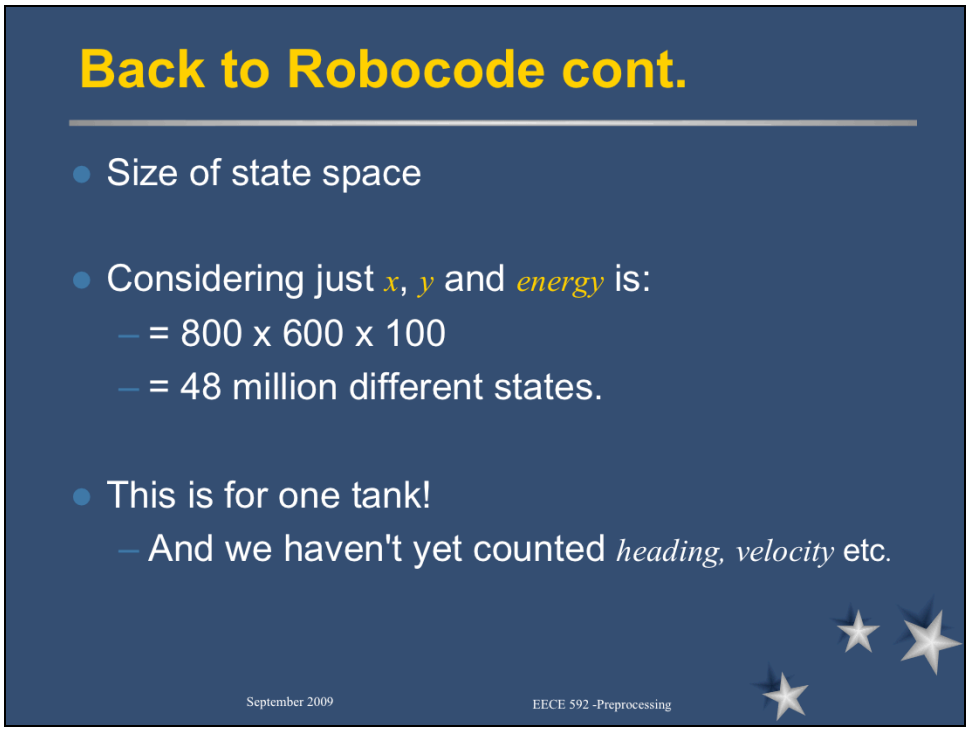
$$x_{\tau-2} \quad x_{\tau-1} \quad x_{\tau} \quad x_{\tau+1}$$

This data can be mapped onto the inputs of a neural net as follows and used to generate a prediction for $x_{\tau+1}$



Back to Robocode

For the assignment you will have to think about how to represent the state of the Robocode environment.



In fact the state space size for one scan of one tank is

(*energy* x *heading* x *x* x *y* x *velocity* x *gun heading* x *radar heading*)

100 x 360 x 800 x 600 x 8 x 360 x 360 = $1.79 * 10^{16}$

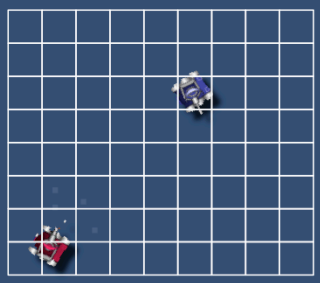
- Even when using a neural net to approximate the value function, you will likely have to reduce this state space. There are many strategies:

(1) Convert positional data to be relative to your tank. E.g. distance / bearing from your tank. This might help.

(2) Quantize the data. E.g. x / y becomes quadrants (a bit extreme). Energy quantized to 10 different levels. (Thermometer codes might help).

Robocode – tile coding

- Partition x, y into a set of n non-overlapping tiles.



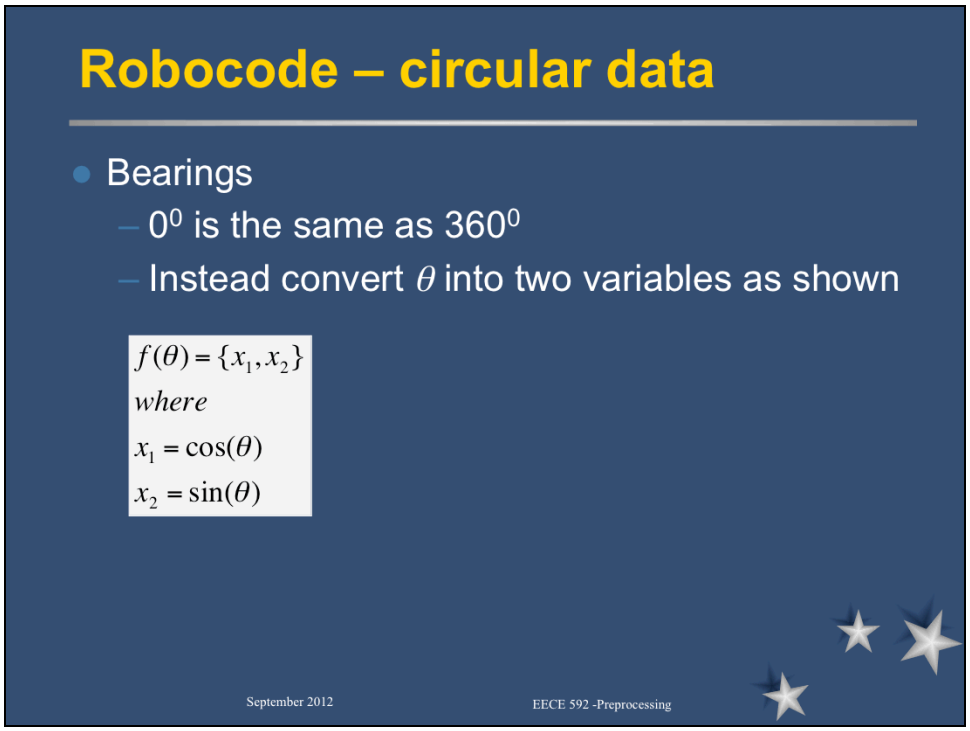
$$\vec{s} = \{x, y\}$$
$$\vec{\phi}(s) = \{\phi_s(1), \phi_s(2), ..., \phi_s(n)\}$$
$$where$$
$$\phi_s(n) \in \{0,1\}$$

- Radial Basis Functions are a natural extension:

$$\phi_s(n) \in [0,1]$$

Where $\phi(s)$ is a function of the state vector s={x,y} and is itself a vector of variables composed of each of the n tile coding functions $\phi_s(n)$. Each tile generates a boolean value. The shape of the tiles need not be regular but can take any shape. E.g. elongated tiles maybe better at generalizing along one dimension over the other.

Note: You can also use overlapping "tiles" – although they are no longer tiles. This is known as coarse coding. Radial basis functions are a natural generalization of coarse coding techniques such that the components of the transfer function are no longer boolean, but continuous and a function of the distance of the input vector from the "centre" of the tile.



By representing circular values such as bearing using sine and cosine, we now have a function that is smooth and continuous, even at the point the value re-cycles. Thus eliminating the issue with 0^0 and 360^0.