CS 367 Tutorial
18 August 2008
Week 5 (tutorial #3)
Carl Schultz

Material is taken from lecture notes (http://www.cs.auckland.ac.nz/compsci367s2c/lectures/index.html) and one of the course text books "Stuart J. Russell and Peter Norvig. Artificial Intelligence : A Modern Approach. Prentice Hall, Upper Saddle River, New Jersey, 1995."

NB: recommended text for this part of the course is "Tom M. Mitchell, Machine Learning McGraw-Hill, New York, 1997"

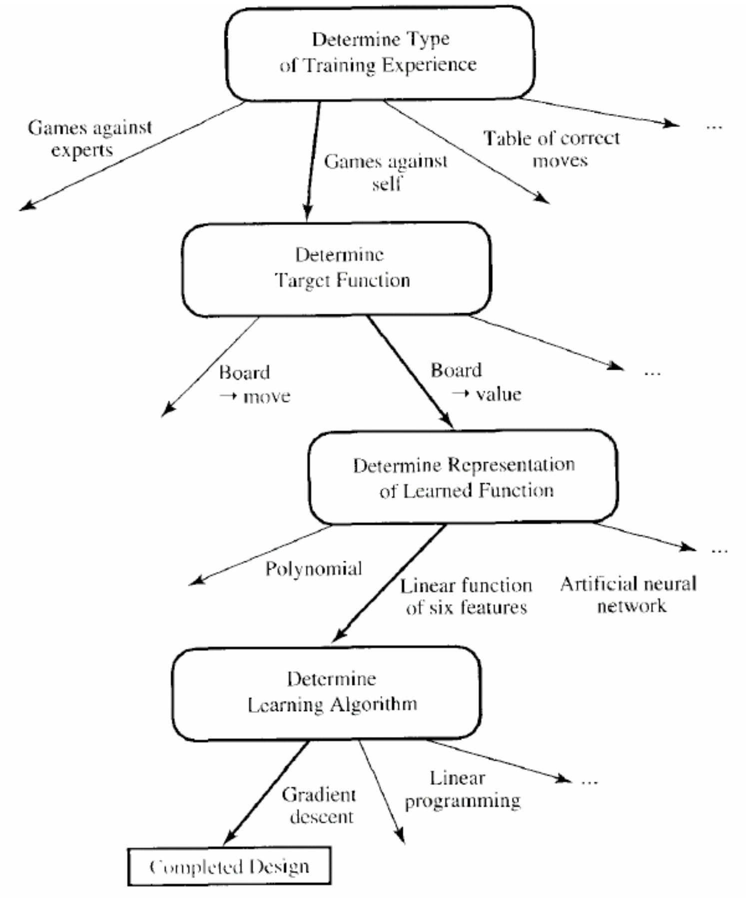
- what does learning mean?
    1) computer program takes some input, produces some output (i.e. it's a *mathematical function*)
    2) evaluates *quality* of its output
    3) based on evaluation, changes itself so that, in future, the quality of the output will hopefully be improved (changes mapping between input and output)

- Formally,
    - **T**: class of tasks that we want computer program to do
    - **P**: measure of performance for how well computer did
    - **E**: some experience program has with task

- e.g. Handwriting Recognition:
    - **T**: recognising and classifying handwritten words with images
    - **P**: percent of words correctly classified
    - **E**: a database of handwritten words with given classifications

    - a computer is said to **learn** from an experience **E** if its performance **P** improves at tasks in **T** after the experience **E**

- we can design learning systems - design choices, e.g.:



- need to determine program's **target function**
    - remember that a computer program is a function: input → output
    - e.g. ChooseMove: "chess board state" → "legal move"
    - e.g. Value: "chess board state" → "score of board state"
      higher scores mean more desirable state; so, consider all legal moves on current board to determine all possible successor board states, and then use Value to decide which successor board state is the best

- need to determine how to **represent** the target function
    - e.g. maybe as a collection of rules?
      ”IF 2 steps away from edge THEN score=40”

    - e.g. maybe a quadratic polynomial function of predefined board features
      ”y=number of black pieces”
      ”z=number of red pieces”
      ”b is a board state”
      ”w_0 and w_1 are weights / numerical coefficients”
      ”$f(b) = w_0 + w_1 y + w_2 z$”

Quick maths revision

- **polynomial**. expression with linear (+ and –) combination of terms (constants × variables) where exponent is non-negative integer (x^4 is okay, but $x^{3/4}$ or x^{-2} not polynomial), e.g. these are polynomial expressions:
  - $x^5 + 3$
  - $x^2 + 5x + 1$
  - $x^3 + 3x^0$
  - $f(x)=x^3 + 2$ …is a polynomial function
  - $0=x^3 + 2$ …is a polynomial equation
- **degree**. take a term, sum the exponents of the variables in that term
  - x^5 has degree 5
  - xy has degree 2
- **degree of a polynomial**. is the highest degree of any of the terms – polynomials with degree 1 to 5 are given special names
  - linear. has degree 1
  - quadratic. has degree 2
  - cubic. has degree 3
  - quartic. has degree 4
  - quintic. has degree 5

- **quadratic polynomial**. has degree 2, e.g.
  - x^2
  - $10x + 3 + x^2$
- *NB: in many cases, people say "quadratic" and mean that the highest* ***allowable*** *degree is 2 (not necessarily exactly 2), i.e. the degree might be less*

- recap.,
  - **T**ask: checkers
  - **E**xperience: games played against self
  - **P**erformance: games won in competition
  - target function is Value: "chess board state" → "score of board state"
    - V: B → real numbers
  - target function representation is polynomial equation
    - $V'(b) = w_0 + w_1x_1 + w_2x_2 + w_3x_3 + w_4x_4 + w_5x_5 + w_6x_6$

- now decide on a learning algorithm
  - training examples: <b, V_{train}(b)>
  - e.g. <(x1=3,x2=0,x3=1,x4=0,x5=0,x6=0), **+100**>

  - want to find values for $w_0 \ldots w_6$ so our function gets a **best fit** for the training data
  - so what changes as we learn? The values of the weights

- **error**
  - difference between expected output (from training data) and actual output (from our learning computer program)

$$\underbrace{V_{train}(b)}_{\text{expected output}} - \underbrace{V'(b)}_{\text{actual system output}}$$

- **overall error**
  - classical measure of error E: sum of squared errors

$$E \equiv \sum_{<b, V_{train}(b)> \in training-examples} (\underbrace{\boxed{V_{train}(b) - V'(b)}}_{\text{error}})^2$$

- **learning**
  - minimise the *squared error* iteratively for each training example
  - i.e. I'll give you (the computer) a training example, and you modify weights $w_0 \ldots w_6$ to minimise:

$$(V_{train}(b) - V'(b))^2$$

  - …and I'll give you another training example, and you modify your weights again,…, and we'll keep doing this until we are satisfied

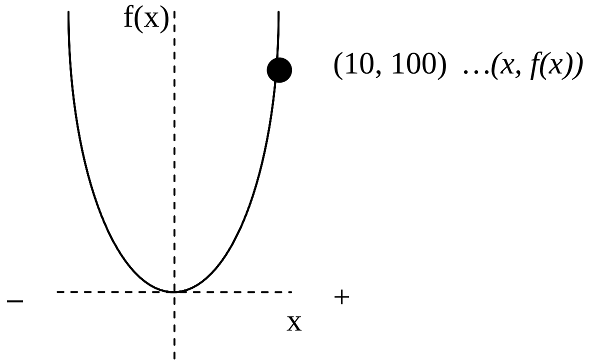
  - algorithm: **Least Mean Squares**
    - stochastic **gradient-descent**

Quick maths revision

- **gradient-descent**. finds the local minimum of a function – does this by moving in direction *negative* of gradient at the current point
    - **a** is current point
    - **b** is next point
    - f '(**a**) is gradient of function at point **a**
    - η is the size of the step that we'll take (must be +ve)

$$\mathbf{b} = \mathbf{a} - \eta \,.\, f'(\mathbf{a})$$

- $f(x) = x^2$



- f '(x) = 2x …derivative of our function used to get the gradient at our point
    - f '(10) = 20 *…formula says we move negative to gradient, so says move left along graph, which seems sensible*
    - **a** = 10
    - η = 0.1 *…something small*
    - **b** = 10 – 0.1 × 20 = 8 *…8 < 10, so we are moving towards the minimum*

- can work for functions with arbitrary number of variables
    - f(x,…, z)

- to do this, take partial derivatives of each variable in turn
    - **partial derivative**. differentiate function for one variable, and fix all other variables (treat as constants)
    - in terms of *learning*, this means we modify each weight in turn

- algorithm: **Least Mean Squares**
  - for each term *i* with weight w_i and variable x_i do the following:

$$\underbrace{w_i}_{\textbf{b}} \leftarrow \underbrace{w_i}_{\textbf{a}} + \underbrace{\eta}_{\substack{\eta \\ \text{(learning rate)}}} \underbrace{(V_{train}(b) - V'(b))x_i}_{\substack{\textbf{negative}\text{ rate-of-change of} \\ \text{error with respect to} \\ \text{weight } i \text{ (i.e. -ve gradient)}}}$$

- *NB: it turns out that the partial derivative of squared error is negative, and so it cancels out the negative sign in front of* η

- think about error
  - if the **error** is positive →
    - our actual value V'(b) is too small, …that is:
    - weight w_i is too small, so we'll **increase** it

    - NB: gradient-descent states that moving in the direction of the negative gradient will *decrease* the function output…this is what we're doing

  - if the **error** is negative →
    - our actual value V'(b) is too big, …that is:
    - weight w_i is too big, so we'll **decrease** it

    - NB: gradient-descent states that moving in the direction of the negative gradient will *decrease* the function output…this is what we're doing