Genetic Programming Discussion: Schema Theory

William H. Hsu
Department of Computing and Information Sciences, KSU

KSOL course page: http://snipurl.com/v9v3
Course web site: http://www.kddresearch.org/Courses/CIS730
Instructor home page: http://www.cis.ksu.edu/~bhsu

Reading for Next Class:

Chapter 20, Russell and Norvig
• Perceptron: Single Neuron Model
  – *aka* Linear Threshold Unit (LTU) or Linear Threshold Gate (LTG)
  – Net input to unit: defined as linear combination \( net = \sum_{i=0}^{n} w_i x_i \)
  – Output of unit: threshold (activation) function on net input (threshold \( \theta = w_0 \))

• Perceptron Networks
  – Neuron is modeled using a unit connected by weighted links \( w_i \) to other units
  – Multi-Layer Perceptron (MLP): next lecture

\[
\sum_{i=0}^{n} w_i x_i \rightarrow \begin{cases} 
1 & \text{if } \sum_{i=0}^{n} w_i x_i > 0 \\
-1 & \text{otherwise}
\end{cases}
\]

Vector notation: \( o(\mathbf{x}) = \text{sgn}(\mathbf{x}, \mathbf{w}) = \begin{cases} 
1 & \text{if } \mathbf{w} \cdot \mathbf{x} > 0 \\
-1 & \text{otherwise}
\end{cases} \)
**Gradient Descent:** Derivation of Delta/LMS (Widrow-Hoff) Rule

**Definition: Gradient**

\[
\nabla E[\vec{w}] = \begin{bmatrix}
\frac{\partial E}{\partial w_0}, & \frac{\partial E}{\partial w_1}, & \ldots, & \frac{\partial E}{\partial w_n}
\end{bmatrix}
\]

**Modified Gradient Descent Training Rule**

\[
\Delta \vec{w} = -r \nabla E[\vec{w}]
\]

\[
\Delta w_i = -r \frac{\partial E}{\partial w_i}
\]

\[
\frac{\partial E}{\partial w_i} = \frac{\partial}{\partial w_i} \left[ \frac{1}{2} \sum_{x \in D} (t(x) - o(x))^2 \right] = \frac{1}{2} \sum_{x \in D} \frac{\partial}{\partial w_i} (t(x) - o(x))^2
\]

\[
= \frac{1}{2} \sum_{x \in D} \left[ 2(t(x) - o(x)) \frac{\partial}{\partial w_i} (t(x) - o(x)) \right] = \sum_{x \in D} (t(x) - o(x)) \frac{\partial}{\partial w_i} (t(x) - \vec{w} \cdot \vec{x})
\]

\[
\frac{\partial E}{\partial w_i} = \sum_{x \in D} [(t(x) - o(x))(-x_i)]
\]
**Algorithm Gradient-Descent \((D, r)\)**

- Each training example is a pair of the form \(<x, t(x)>, \) where \(x\) is the vector of input values and \(t(x)\) is the output value. \(r\) is the learning rate (e.g., 0.05)
- Initialize all weights \(w_i\) to (small) random values
- **UNTIL** the termination condition is met, DO
  
  Initialize each \(\Delta w_i\) to zero
  
  FOR each \(<x, t(x)>\) in \(D\), DO
  
  Input the instance \(x\) to the unit and compute the output \(o\)
  
  FOR each linear unit weight \(w_i\), DO
  
  \[
  \Delta w_i \leftarrow \Delta w_i + r(t - o)x_i
  \]
  
  \[
  w_i \leftarrow w_i + \Delta w_i
  \]

- **RETURN** final \(w\)

**Mechanics of Delta Rule**

- Gradient is based on a derivative
- **Significance:** later, will use nonlinear activation functions (aka transfer functions, squashing functions)
Review: Derivation of Backprop

Recall: Gradient of Error Function

\[ \nabla E[\mathbf{w}] = \left[ \frac{\partial E}{\partial w_0}, \frac{\partial E}{\partial w_1}, \ldots, \frac{\partial E}{\partial w_n} \right] \]

Gradient of Sigmoid Activation Function

\[
\frac{\partial E}{\partial w_i} = \frac{\partial}{\partial w_i} \left[ \frac{1}{2} \sum_{\langle \hat{x}, t(x) \rangle \in D} (t(\hat{x}) - o(\hat{x}))^2 \right] = \frac{1}{2} \sum_{\langle \hat{x}, t(x) \rangle \in D} \left[ \frac{\partial}{\partial w_i} (t(\hat{x}) - o(\hat{x}))^2 \right]
\]

\[
= \frac{1}{2} \sum_{\langle \hat{x}, t(x) \rangle \in D} \left[ 2(t(\hat{x}) - o(\hat{x})) \frac{\partial}{\partial w_i} (t(\hat{x}) - o(\hat{x})) \right] = \sum_{\langle \hat{x}, t(x) \rangle \in D} \left[ (t(\hat{x}) - o(\hat{x})) \left( -\frac{\partial o(\hat{x})}{\partial w_i} \right) \right]
\]

But We Know:

\[
\frac{\partial o(\hat{x})}{\partial net(\hat{x})} = \frac{\partial o(net(\hat{x}))}{\partial net(\hat{x})} = o(\hat{x})(1-o(\hat{x}))
\]

\[
\frac{\partial net(\hat{x})}{\partial w_i} = \frac{\partial (\mathbf{w} \cdot \hat{x})}{\partial w_i} = x_i
\]

So:

\[
\frac{\partial E}{\partial w_i} = - \sum_{\langle \hat{x}, t(x) \rangle \in D} \left[ (t(\hat{x}) - o(\hat{x})) \cdot o(\hat{x})(1-o(\hat{x})) \cdot x_i \right]
\]
**Intuitive Idea:** Distribute *Blame* for Error to Previous Layers

**Algorithm** *Train-by-Backprop* $(D, r)$

- Each training example is a pair of the form $<x, t(x)>$, where $x$ is the vector of input values and $t(x)$ is the output value. $r$ is the learning rate (e.g., 0.05)
- Initialize all weights $w_{ij}$ to (small) random values
- UNTIL the termination condition is met, DO
  
  FOR each $<x, t(x)>$ in $D$, DO
    
    Input the instance $x$ to the unit and compute the output $o(x) = \sigma(\text{net}(x))$
    
    FOR each output unit $k$, DO
      
      $\delta_k = o_k(x)(1 - o_k(x))(t_k(x) - o_k(x))$

    FOR each hidden unit $j$, DO
      
      $\delta_j = h_j(x)(1 - h_j(x)) \sum_{k \in \text{outputs}} v_{jk} \delta_k$

    Update each $w = u_{ij} (a = h_i)$ or $w = v_{jk} (a = o_k)$

    $w_{\text{start-layer, end-layer}} \leftarrow w_{\text{start-layer, end-layer}} + \Delta w_{\text{start-layer, end-layer}}$
    
    $\Delta w_{\text{start-layer, end-layer}} \leftarrow r \Delta \delta_{\text{end-layer}} a_{\text{end-layer}}$

- RETURN final $u, v$
**Simple Genetic Algorithm (SGA)**

- Algorithm *Simple-Genetic-Algorithm* (*Fitness*, *Fitness-Threshold*, *p*, *r*, *m*)

  // *p*: population size; *r*: replacement rate (*aka* generation gap width), *m*: string size

  - *P* ← *p* random hypotheses   // initialize population
  - FOR each *h* in *P* DO *f*[*h*] ← *Fitness*(*h*)   // evaluate *Fitness*: hypothesis → R
  - WHILE (*Max*(*f*) < *Fitness-Threshold*) DO
    - 1. **Select**: Probabilistically select (1 - *r*) *p* members of *P* to add to *P*<sub>S</sub>
      
      \[ P(h_i) = \frac{f[h_i]}{\sum_{j=1}^{p} f[h_j]} \]
    - 2. **Crossover**:
      - Probabilistically select (*r* · *p*)/2 pairs of hypotheses from *P*
      - FOR each pair <*h*<sub>1</sub>, *h*<sub>2</sub>> DO
        
        \[ P_{s} += \text{Crossover}(<h_1, h_2>) \quad \text{// } P_{s}[t+1] = P_{s}[t] + <\text{offspring}_1, \text{offspring}_2> \]
    - 3. **Mutate**: Invert a randomly selected bit in *m* · *p* random members of *P*<sub>S</sub>
    - 4. **Update**: *P* ← *P*<sub>S</sub>
    - 5. **Evaluate**: FOR each *h* in *P* DO *f*[*h*] ← *Fitness*(*h*)

  - RETURN the hypothesis *h* in *P* that has maximum fitness *f*[*h*]
GA-Based Inductive Learning (GABIL)

- GABIL System [Dejong et al, 1993]
  - Given: concept learning problem and examples
  - Learn: disjunctive set of propositional rules
  - Goal: results competitive with those for current decision tree learning algorithms (e.g., C4.5)

- Fitness Function: $Fitness(h) = (Correct(h))^2$

- Representation
  - Rules: IF $a_1 = T \land a_2 = F$ THEN $c = T$; IF $a_2 = T$ THEN $c = F$
  - Bit string encoding: $a_1 [10] \cdot a_2 [01] \cdot c [1] \cdot a_1 [11] \cdot a_2 [10] \cdot c [0] = 10011$
    $11100$

- Genetic Operators
  - Want variable-length rule sets
  - Want only well-formed bit string hypotheses
CROSSOVER: VARIABLE-LENGTH BIT STRINGS

- Basic Representation
  - Start with
    
    \[
    \begin{array}{ccc}
    a_1 & a_2 & c \\
    h_1 & 1 & 01 & 1 & 11 & 10 & 0 \\
    h_2 & 0 & 1 & 0 & 10 & 01 & 0 \\
    \end{array}
    \]
  - Idea: allow crossover to produce variable-length offspring

- Procedure
  - 1. Choose crossover points for \( h_1 \), e.g., after bits 1, 8
  - 2. Now restrict crossover points in \( h_2 \) to those that produce bitstrings with well-defined semantics, e.g., \(<1, 3>, <1, 8>, <6, 8>\)

- Example
  - Suppose we choose \(<1, 3>\)
  - Result
    
    \[
    \begin{array}{ccc}
    h_3 & 11 & 10 & 0 \\
    h_4 & 00 & 01 & 1 & 11 & 11 & 0 & 10 & 01 & 0 \\
    \end{array}
    \]
**GABI Extensions**

- **New Genetic Operators**
  - Applied probabilistically
  - 1. **AddAlternative**: generalize constraint on \( a_i \) by changing a 0 to a 1
  - 2. **DropCondition**: generalize constraint on \( a_i \) by changing every 0 to a 1

- **New Field**
  - Add fields to bit string to decide whether to allow the above operators
    
    \[
    \begin{array}{cccccccc}
    a_1 & a_2 & c & a_1 & a_2 & c & AA & DC \\
    01 & 11 & 0 & 10 & 01 & 0 & 1 & 0
    \end{array}
    \]
  - So now the learning strategy also evolves!
  - *aka genetic wrapper*
GABIL Results

- Classification Accuracy
  - Compared to symbolic rule/tree learning methods
    - C4.5 [Quinlan, 1993]
    - ID5R
    - AQ14 [Michalski, 1986]
  - Performance of GABIL comparable
    - Average performance on a set of 12 synthetic problems: 92.1% test accuracy
    - Symbolic learning methods ranged from 91.2% to 96.6%

- Effect of Generalization Operators
  - Result above is for GABIL without AA and DC
  - Average test set accuracy on 12 synthetic problems with AA and DC: 95.2%
Building Blocks (Schemas)

- Problem
  - How to characterize evolution of population in GA?

- Goal
  - Identify basic building block of GAs
  - Describe family of individuals

- Definition: Schema
  - String containing 0, 1, * ("don’t care")
  - Typical schema: 10**0*
  - Instances of above schema: 101101, 100000, …

- Solution Approach
  - Characterize population by number of instances representing each possible schema

\[ m(s, t) = \text{number of instances of schema } s \text{ in population at time } t \]
Selection and Building Blocks

- **Restricted Case: Selection Only**
  - \( \bar{f}(t) \equiv \text{average fitness of population at time } t \)
  - \( m(s, t) \equiv \text{number of instances of schema } s \text{ in population at time } t \)
  - \( \hat{u}(s,t) \equiv \text{average fitness of instances of schema } s \text{ at time } t \)

- **Quantities of Interest**
  - Probability of selecting \( h \) in one selection step
    \[
    P(h) = \frac{f(h)}{\sum_{i=1}^{n} f(h_i)}
    \]
  - Probability of selecting an instance of \( s \) in one selection step
    \[
    E[m(s, t+1)] = \frac{\hat{u}(s, t)}{\bar{f}(t)} \cdot m(s, t)
    \]
  - Expected number of instances of \( s \) after \( n \) selections
Schema Theorem

\[ E[m(s, t + 1)] \geq \frac{\hat{u}(s, t)}{f(t)} \cdot m(s, t) \cdot \left(1 - p_c \frac{d_s}{l-1}\right) \cdot (1 - p_m)^{o(s)} \]

- \( m(s, t) \) \equiv number of instances of schema \( s \) in population at time \( t \)
- \( f(t) \) \equiv average fitness of population at time \( t \)
- \( \hat{u}(s, t) \) \equiv average fitness of instances of schema \( s \) at time \( t \)
- \( p_c \) \equiv probability of single point crossover operator
- \( p_m \) \equiv probability of mutation operator
- \( l \) \equiv length of individual bit strings
- \( o(s) \) \equiv number of defined (non “*”) bits in \( s \)
- \( d(s) \) \equiv distance between rightmost, leftmost defined bits in \( s \)

Intuitive Meaning

- “The expected number of instances of a schema in the population tends toward its relative fitness”

A fundamental theorem of GA analysis and design
**Terminology**

- **Evolutionary Computation (EC): Models Based on Natural Selection**

- **Genetic Algorithm (GA) Concepts**
  - **Individual**: single entity of model (corresponds to hypothesis)
  - **Population**: collection of entities in competition for survival
  - **Generation**: single application of selection and crossover operations
  - **Schema aka building block**: descriptor of GA population (e.g., $10^{**0**}$)
  - **Schema theorem**: representation of schema proportional to its relative fitness

- **Simple Genetic Algorithm (SGA) Steps**
  - **Selection**
    - Proportionate reproduction (*aka* roulette wheel): $P(\text{individual}) \propto f(\text{individual})$
    - Tournament: let individuals compete in pairs or tuples; eliminate unfit ones
  - **Crossover**
    - Single-point: $11101001000 \times 00001010101 \rightarrow \{ 11101010101, 00001001000 \}$
    - Two-point: $11101001000 \times 00001010101 \rightarrow \{ 11001011000, 00101000101 \}$
    - Uniform: $11101001000 \times 00001010101 \rightarrow \{ 10001000100, 01101011001 \}$
  - **Mutation**: single-point (“bit flip”), multi-point
Lecture Outline

- Readings / Viewings
  - View GP videos 1-3
    - GP1 – *Genetic Programming: The Video*
    - GP2 – *Genetic Programming: The Next Generation*
    - GP3 – *Genetic Programming: Human-Competitive*
  - Suggested: Chapters 1-5, Koza

- Previously
  - Genetic and evolutionary computation (GEC)
  - Generational vs. steady-state GAs; relation to simulated annealing, MCMC
  - Schema theory and GA engineering overview

- Today: GP Discussions
  - Code bloat and potential mitigants: types, OOP, parsimony, optimization, reuse
  - Genetic programming vs. human programming: similarities, differences

Thursday: Course Review
GP Flow Graph

Flowchart for Genetic Programming

1. **Gen = 0**
   - Create Initial Random Population
   - Termination Criteria Satisfied?
     - Yes → Result
     - No → Execute Fitness of Each Individual in Population
     - Individuals = 0
     - **Gen = Gen + 1**

2. Select Genetic Operation Probabilistically
   - Reproduction
     - Select One Individual Based on Fitness
     - Perform Reproduction
     - Copy into New Population
     - Individuals = Individuals + 1
   - Mutation
     - Select One Individual Based on Fitness
     - Perform Mutation
     - Insert Mutant into New Population
     - Individuals = Individuals + 1
   - Crossover
     - Select Two Individuals Based on Fitness
     - Perform Crossover
     - Insert Offspring into New Population
     - Individuals = Individuals + 1
Structural Crossover

Crossover Operation with Different Parents

Parents

Children

Crossover Operation with Identical Parents

Parents

Children

Adapted from The Genetic Programming Notebook © 2002 Jaime J. Fernandez
http://www.geneticprogramming.com
Structural Mutation

Mutation

Original Individual

Mutated Individuals
GENETIC PROGRAMMING: THE NEXT GENERATION
(Synopsis and Discussion)

- Automatically-Defined Functions (ADFs)
  - *aka* macros, anonymous inline functions, subroutines
  - Basic method of software reuse

- Questions for Discussion
  - *What are advantages, disadvantages of learning anonymous functions?*
  - *How are GP ADFs similar to and different from human-produced functions?*

- Exploiting Advantages
  - Reuse
  - Innovation

- Mitigating Disadvantages
  - Potential lack of meaning – semantic clarity issue (and topic of debate)
  - Redundancy

Accelerated bloat – scalability issue
Code Bloat [1]: Problem Definition

- **Definition**
  - Increase in program size not commensurate with increase in functionality (possibly as function of problem size)
  - Compare: structural criteria for overfitting, overtraining

- **Scalability Issue**
  - Large GPs will have this problem
  - Discussion: *When do we expect large GPs?*
  - Machine learning: large, complex data sets
  - Optimization, control, decision making / DSS: complex problem

- **What Does It Look Like?**

- **What Can We Do About It?**
  - ADFs
  - Advanced reuse techniques from software engineering: e.g., design patterns
  - Functional, object-oriented design; theory of types
  - Controlling size: parsimony (MDL-like), optimization (cf. compilers)
Code Bloat [2]: Mitigants

- Automatically Defined Functions

- Types
  - Ensure
    - Compatibility of functions created
    - Soundness of functions themselves
  - Define: abstract data types (ADTs) – object-oriented programming
  - Behavioral subtyping – still “future work” in GP
  - Generics (cf. C++ templates)
  - Polymorphism

- Advanced Reuse Techniques
  - Design patterns
  - Workflow models
  - Inheritance, reusable classes
Code Bloat [3]:
More Mitigants

- Parsimony (cf. Minimum Description Length)
  - Penalize code bloat
  - Inverse fitness = loss + cost of code (evaluation)
  - May include terminals

- Target Language Optimization
  - Rewriting of constants
  - Memoization
  - Loop unrolling
  - Loop-invariant code motion
16 Criteria for Automatic Program Synthesis by Computational Intelligence

1. **Specification**: starts with *what needs to be done*
2. **Procedural representation**: tells us how to do it
3. **Algorithm implementation**: produces a computer program
4. **Automatic determination of program size**
5. **Code reuse**
6. **Parametric reuse**
7. **Internal storage**
8. **Iteration** (while / for), recursion
9. **Self-organization of hierarchies**
10. **Automatic determination of architecture**
11. **Wide range of programming constructs**
12. **Well-defined**
13. **Problem independent**
Genetic Programming 3 (Synopsis and Discussion [2])

- 16 Criteria for Automatic Program Synthesis ...
  - 14. Generalization: wide applicability
  - 15. Scalability
  - 16. Human-competitiveness

- Current Bugbears: Generalization, Scalability

- Discussion: Human Competitiveness?
More Food for Thought and Research Resources

- Discussion: Future of GP
- Current Applications
- Conferences
  - GECCO: ICGA + ICEC + GP
  - GEC
  - EuroGP
- Journals
  - Evolutionary Computation Journal (ECJ)
  - Genetic Programming and Evolvable Machines (GPEM)