SAT with Gradient Descent

Objective

The aim of this project is to investigate to what extent Stochastic Gradient Descent can be helpful in approximating solutions to the Boolean Satisfiability problem (SAT). This is well know to be an NP-complete problem in its discrete form, which can be stated as:

> Given a list of logical expressions defined over a set of variables, determine whether there is a truth assignment to these variables such that all the given expressions evaluate to **true**.

This problem can be framed as *maximizing the sum of the truth values of all the boolean expressions on the vertices of the boolean hypercube*, which brings to light its intimate tie with integer programming (which is thus also NP-complete).

Approach

Our approach consists of relaxing the constraint that the "truth values" of inputs and logical expressions be binary.Instead, we allow them to range from 0 to 1. We will call these *augmented inputs* and *augmented expressions*. Similarly, a circuit that evaluates augmented expressions using augmented inputs will be called an *augmented circuit*.

The relaxed problem can then be framed as the optimization of a continuous (but in general non-linear) utility function given by the sum of the truth values of a set of augmented expressions, given a set of augmented inputs. This would make it an instance of constrained optimization on the boolean hypercube.

It is more desirable and efficient, however, to solve an unconstrained optimization problem than a constrained one. For this reason, we choose a function that maps R^n onto the boolean hypercube. Similarly to what is done in logistic regression, we can use the sigmoid function for this purpose:

$$\sigma(x) = \frac{1}{1 + e^{-x}}$$

Also taking inspiration from logistic regression, we use cross-entropy as a loss function:

$$\mathcal{L}(\mathbf{y}, \hat{\mathbf{y}}) = -\sum_i y_i \, log(\hat{y}_i) + (1 - y_i) \, log(1 - \hat{y}_i)$$

In this case, all truth labels y_i are true, so this reduces to:

$$\mathcal{L}(\hat{\mathbf{y}}) = -\sum_i log(\hat{y}_i)$$

Hence, we can frame the relaxed problem as an *unconstrained* optimization:

Let $\mathbf{S} = \{s_1, s_2, \ldots, s_k\}$ be a set of augmented logical expressions defined on a set of augmented boolean inputs $\mathbf{e} = \{e_1, e_2, .. e_n\}$. Let $\mathbf{S}(\mathbf{e}) = \{s_1(\mathbf{e}), s_2(\mathbf{e}), \ldots, s_k(\mathbf{e})\}$, where $s_i(\mathbf{e})$ denotes the value of augemented expression s_i when evaluated with inputs $\mathbf{e}$. Define a vector $\mathbf{x} \in R^n : \mathbf{x} = \{x_1, x_2, \ldots, x_n\}$, and let $e_i = \sigma(x_i)$ for all i. Then the problem is to find

$$\mathbf{x}^* = \underset{\mathbf{x}}{argmin} \; \mathcal{L}(\mathbf{S}(\mathbf{e}))$$

Our approach will attempt to approximate $\mathbf{x}^*$ using stochastic gradient descent. In particular, we will construct a computation graph representing an augmented circuit that computes $\mathbf{S}(\mathbf{e})$. It will be augmented not only for allowing augmented boolean inputs, but also for supporting backpropagation, much like computation graphs used in Deep Learning. The difference is that, instead of optimizing parameters of a model, our aim is to optimize the inputs to obtain as large a truth value as possible.

However, since an augmented *OR* gate is represented by a nonlinear function, it will not be possible to use a vectorized implementation for this. Instead, we explicitly model the circuit as a network of Computation Nodes, each supporting forward propagation and backpropagation. The supported nodes are:

Node type	Description	Input type	Forward step	Backward step for input a
InputNode	Contains real numbers x_i	Unary	a	1
SigmoidGate	Converts x_i to e_i	Unary	$\sigma(a)$	$\sigma(a) * (1 - \sigma(a))$
AndGate	Logical AND gate	Binary	$a \cdot b$	b
OrGate	Logical OR gate	Binary	$a + b - a \cdot b$	$1 - b$
NotGate	Logical NOT gate	Unary	$1 - a$	-1

Node type	Description	Input type	Forward step	Backward step for input a
CostNode	Computes cost for a given expression	Unary	$-log(a)$	$\frac{1}{a}$

Usage

Input

The `main` method of the `ComputationGraph` class reads from `System.in` a number n and a series of logical expressions using variables labeled from 0 to n. The expressions must be formated as follows:

- The ith variable is denoted by `$i` (i.e. `$0`, `$1`, `$10`, etc.).
- The logical AND operation is denoted by `^` (i.e. `$3 ^ $5` represents 3 AND 4).
- The logical OR operation is denoted by `v` (i.e. `$0 v $1` represents 3 OR 4).
- The logical NOT operation is denoted by `~` (i.e. `~$3` represents NOT 3).

Expression example: `$10 v ~($11 ^ ~$12)`

For now, only these operations are supported, but they are sufficient to write any boolean function as a logical expression.

Hence, an example of an input to the program would be

```
6
$0 v ~($1 ^ ~$2)
$1 v ~($2 ^ ~$3)
$2 v ~($3 ^ ~$4)
$3 v ~($4 ^ ~$5)
```

Output

The program will output a list relating each augmented boolean variable to its truth value after a number of steps of gradient descent (which can be modified in `ComputationGraph.java`). An example of output is:

```
$0 = 0,696331
$1 = 0,898549
$2 = 0,999418
$3 = 0,998624
```

```
$4 = 0,945440
$5 = 0,954613
```

Notice how some outputs are much further to either 0 or 1 than others. One of the questions we must investigate is why this is - is it because they can be both true and false and still yield valid solutions, or is it an inherent limitation of the method?

Compiling and running

Compile using

```
javac ComputationGraph.java
```

Run using

```
java ComputationGraph < inputFile
```

The class `CNFParser` takes files in *DIMACS cnf format* (such as the ones available at https://www.cs.ubc.ca/~hoos/SATLIB/benchm.html) and converts them to the appropriate form used by the program. Hence, one can also use on a Unix machine

```
java ComputationGraph | java CNFParser cnfFileName
```