“Machine Learning Languge” Workshop Jugendforum BW Informatik

Rainer Gemulla Roland Leißa Simon Forbat
Universität Mannheim
{rgemulla,leissa,sforbat}@uni-mannheim.de

22.2.2024

Ziel dieses Workshops ist es, eine voll funktionale DSL (*domain-specific language*) namens *MLL* für das maschinelle Lernen zu erstellen. Die Sprache soll es ermöglichen, Rechengraphen auszudrücken, Berechnungen durchzuführen, Gradienten auszurechnen, sowie Optimierungen durchzuführen. Am Ende des Workshops werden wir diese Sprache verwenden, um ein einfaches Modell des maschinellen Lernens zu trainieren.

1 Erwärmung

MLL verwendet Rechengraphen, in denen jede Operation genau einen reellen Wert ausgibt (`double`). Folgende Klassen sind bereits (teilweise) vorgegeben:

- **DAG.** Ein Rechengraph.
- **Op.** Ein Knoten (Operation) im Rechengraph.
- **Lit.** Ein Literal, d.h. ein Knoten, der einen konstanten Wert zurückliefert.
- **Var.** Eine Variable, d.h. ein Knoten, der den Wert eine Variable zurückliefert.
- **UnOp.** Eine unäre Operation (ein Eingang).
- **BinOp.** Eine binäre Operation (zwei Eingänge).
- **Add.** Die binäre Additionsoperation.

a) Führe folgenden Code aus:

```
var dag = new DAG();
Op out = dag.lit(1);
System.out.println( out.dot() );
Util.saveDotPng(out.dot(), "1a");
```

Was passiert in jeder Zeile? (Die letzten beiden Zeilen sind nützlich, wir werden sie aus Platzgründen aber im Folgenden nicht explizit angeben.)

Hinweis: Du kannst für diese Aufgaben auch mit einem Jupyter Notebook arbeiten; das ist vermutlich etwas bequemer. Ersetze dafür die letzte Zeile durch

```
Util.viewDot(out.dot());
```

b) Führe folgenden Code aus:

```
var dag = new DAG();
out = dag.lit(1).add(dag.lit(2));
```

Was passiert in jedem Schritt?

c) Der Code aus der vorherigen Aufgabe ist eine *Kurzschreibweise* für:

```
var dag = new DAG();
out = Add.c(Lit.c(dag, 5), Lit.c(dag, 3));
```

Vergewissere dich, dass beide das gleiche Ergebnis zurückliefern. Warum ist das so? Wie wird die Kurzschreibweise im Quellcode umgesetzt?

d) Um einen Rechengraphen auszuwerten, verwenden wir eine `env`*ironment*, welches jedem Knoten einen Wert zuordnet. Das sieht wie folgt aus:

```
var env = new HashMap<Op, Double>();
System.out.println( dag.eval(env) );
```

Teste, ob die obigen Rechengraphen korrekt evaluiert werden. Was steht am Ende in `env`?

e) Schaue dir die Methode `Op.eval` an. Wie wird hier vorgegangen? Was ist die Rolle der Methode `Op.eval_`? Schaue dir die Implementierung dieser Methode in `Add`, `Lit` und `Var` an.

f) Erstelle einen Rechengraphen für den Ausdruck:

$$(x + 5) + (2 + 3)$$

Achte auf korrekte Klammerung. Wie viele Knoten hat der Rechengraph?

g) Führe den folgenden Code aus:

```
System.out.println( dag.eval(1) );
System.out.println( dag.eval(2) );
System.out.println( dag.eval(3) );
```

Was wird hier berechnet? Wie ist diese Berechnung intern umgesetzt?

h) Wir können Rechengraphen mit freien Variablen (wie oben `x`) als Funktionen auffassen. Du kannst eine Funktion wie folgt graphisch ausgeben:

```
var x = Util.getx(-5, 5, 100);
var y = Util.gety(x, out);
Util.savePlot(x, y, "1h");
```

Schaue dir an, was hier intern passiert.

Hinweis: In Jupyter Notebook, ersetze die letzte Zeile durch

```
Util.viewPlot(x, y, "1h");
```

2 Vorwärts

Du solltest jetzt ein gutes Verständnis der Sprache MLL haben. Momentan unterstützt die Sprache allerdings nur Literale, Variablen und Additionen.

a) Füge zumindest die folgenden Operationen hinzu, indem du jeweils eine entsprechende Klasse sowie eine Kurzschreibweise anlegst.

(i) `Neg`. Negation $-x$.

(ii) `Exp`. Exponentation $\exp(x)$

(iii) `Log`. Natürlicher Logarithmus $\log(x)$

(iv) `Sub`. Substraktion $x - y$

(v) `Div`. Division x/y

(vi) `Mul`. Multiplikation xy

(vii) `Pow`. Potenz x^y

Folge dem vorgegebenen Muster, um eine Operation zu implementieren, d.h. erbe von `UnOp` oder `BinOp` und implementiere `c` zum Erstellen (nicht den Konstruktor; warum erfahrt ihr unten) und `eval_` zum Auswerten.

Die Methoden `diff` und `llvm_` haben wir noch nicht besprochen. "Implementiert" sie vorerst wie folgt:

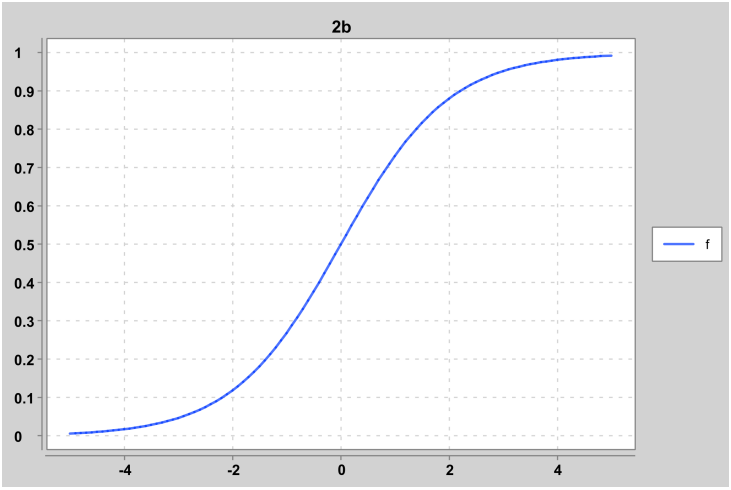
```
@Override protected String
llvm_(HashMap<Op, String> map, Writer writer) throws IOException {
  throw new UnsupportedOperationException();
}

@Override protected Op diff(int inputIdx) {
  throw new UnsupportedOperationException();
}
```

b) Zeichne die *logistische Funktion* σ mit deinem Programm:

$$\sigma(x) = \frac{1}{1 + \exp(-x)}.$$

Wenn du alles richtig gemacht hast, erhältst du das folgende Ergebnis:



c) Schau dir den Rechengraph von $(x + x) + (1 + 1)$ an. Was fällt dir auf? Warum verwenden wir Rechengraphen und nicht Rechenbäume?

d) Wir verwenden pro `Op` eine *statische* Methode `c`(*reate*), um neue Operatoren zu erstellen. Das ist ein sogenannter *smart constructor*. Seine Verwendung erlaubt es uns, den Rechengraphen während seiner Erstellung zu optimieren. Genau das tun wir jetzt. Implementiere die folgende Regel, indem du die entsprechenden `c`-Methoden anpasst:

> Wenn alle Eingaben eines Operators Literale sind, soll direkt das Ergebnis berechnet und als Literal zurückgegeben werden.

Wenn du alles richtig gemacht hast, werden Teilausdrücke ohne Variablen direkt ausgewertet. Also z.B. sollte der DAG für

$$\frac{1}{1+\exp(-0)}$$

aus einem einzigen Knoten (Literal mit Wert 0.5) bestehen.

e) Implementiere weitere sinnvolle Regeln. Zum Beispiel:

- Ersetze $x + 0$ sowie $x - 0$ durch x.
- Ersetze $x + x$ durch $2x$.
- Ersetzt $x - x$ durch 0.

Beachte dabei auch, dass einige Operationen kommutativ sind.

f) Überlege: Kannst du mit den von dir geschriebenen Regeln den Rechengraphen vollständig optimieren? Warum oder warum nicht? Schaue dir dazu unter anderem nochmal den Rechengraphen für

$$(x + 5) + (2 + 3)$$

aus Aufgabe 1 an, nachdem du deine Umschreiberegeln implementiert hast.

3 Rückwärts

MLL ist jetzt hinreichend mächtig. Als nächsten Schritt werden wir Operationen hinzufügen, die es uns erlauben, Ableitungen zu berechnen. Unser Ziel ist es, die Ableitungen der Ausgabe bezüglich jeder der Eingaben (Variablen) zu berechnen. Dazu verwenden wir *Backpropagation*.

a) Betrachte die Funktion

$$f(x) = (x + x) + 5$$

mit Ableitung

$$f'(x) = \frac{\partial}{\partial x} f(x) = 2.$$

Wir können die Ableitung in MLL wie folgt berechnen:

```
var dag = new DAG();
Op out = dag.x().add(dag.x()).add(dag.lit(5)); // f(x)
Grad dout = out.backwards(); // f'(x)
Util.saveDotPng(dout.dot(), "3a-backwards");
```

Führe diesen Code aus und schaue dir den entstehenden Rechengraphen an. Wie wird die Ableitung im Rechengraph repräsentiert?

Hinweis: Wenn du eine Umschreiberegel eingeführt hast, die $x + x$ durch $2x$ ersetzen, dann erhältst du hier vermutlich einen Fehler. Wenn dem so ist, deaktiviere diese Umschreiberegeln für einen Moment.

b) Betrachte die Funktion

$$f(x, y) = (x + x) + (y + y + y) + 5.$$

Implementiere diese Funktion in MLL und berechne die partiellen Ableitungen wie oben. Was ist der Wert der beiden Ableitungen $\frac{\partial}{\partial x} f(x, y)$ sowie $\frac{\partial}{\partial y} f(x, y)$?

c) MLL enthält eine Implementation von Backpropagation. Damit diese Implementation funktioniert, muss jeder Operator eine Funktion `diff` implementieren. Lies' dir die Dokumentation dieser Funktion durch und schaue dir deren Implementierung in `Add#diff` an. Was passiert hier?

Hinweis: Die partiellen Ableitungen sind:

$$\begin{aligned} add(x, y) &= x + y \\ \frac{\partial}{\partial x} add(x, y) &= 1 \\ \frac{\partial}{\partial y} add(x, y) &= 1. \end{aligned}$$

d) Du kannst Ableitungen auch auswerten und visualieren. Hier sind ein paar Beispiele:

```
var dag = new DAG();
Op out = dag.x().mul(dag.x()); // f(x)
Grad dout = out.backwards(); // f'(x)
Util.saveDotPng(dout.dot(), "3d-backwards");

System.out.println(dout.eval(5)); // should give 25
System.out.println(dout.result()); // should give 25
System.out.println(dout.grad("x")); // should give 10

var env = new HashMap<Op,Double>();
System.out.println(dout.eval(env, 5)); // should give 25
System.out.println(env.get(dout.input(0))); // should give 25
System.out.println(env.get(dout.input(1))); // should give 10

Util.saveDotPng(dout.dot(env), "3d-backwards-env");

var x = Util.getx(-10,  10, 100);
double[] y1 = Util.gety(x, dout);
double[] y2 = Util.getTangent(x, 5., dout);
Util.savePlot(x, y1, y2, "3d");
```

Führe diese Beispiele aus und vollziehe nach, was die jeweiligen Ausgaben sind und warum.

e) Implementiere die `diff`-Funktion für den Operator `Mul`. Nutze dabei die partiellen Ableitungen

$$\begin{aligned} mul(x, y) &= x * y \\ \frac{\partial}{\partial x} mul(x, y) &= y \\ \frac{\partial}{\partial y} mul(x, y) &= x. \end{aligned}$$

Teste deinen Code, indem du einige Funktionen mit MLL ableitest. Zum Beispiel:

$$\begin{aligned} f_1(x) &= 2 * x & f_1'(x) &= 2 \\ f_2(x) &= x * x & f_2'(x) &= 2x \\ f_3(x) &= x * x * x * x * x & f_3'(x) &= 5x^4 \end{aligned}$$

Wie ist sind die Ableitungen jeweils im Rechengraphen repräsentiert?

Hinweis: Du kannst die Umschreiberegel $x + x$ nach $2x$ jetzt wieder aktivieren, musst für diese Aufgabe aber ggf. die Umschreiberegel $x * x$ nach x^2 deaktivieren.

f) Implementiere die `diff`-Funktion für weitere Operatoren. Bestimme und implementiere dazu jeweils die jeweiligen Ableitungen:

$$\frac{\partial}{\partial x} unop(x)$$
$$\frac{\partial}{\partial x} binop(x, y)$$
$$\frac{\partial}{\partial y} binop(x, y)$$

Du solltest anschließend z.B. die Ableitung der folgenden Funktion berechnen und auswerten können (aus Aufgabe 2):

$$\sigma(x) = \frac{1}{1 + \exp(-x)}.$$

Hinweis: Wenn dir die (partiellen) Ableitungen einiger Funktionen nicht bekannt sind bzw. du sie nicht nachschlagen kannst (z.B. auf Wikipedia), frag' uns!

g) (*) Schaue dir die Implementation von Backpropagation in `Op#backwards` an. Kannst du beschreiben, wie der Algorithmus funktioniert?

4 Schneller

Wir können uns die Rechengraph aus den vorherigen Aufgaben als ein Programm vorstellen, das beschreibt, wie aus Eingaben (Variablen) bestimmte Ausgaben (Ergebnisse) berechnet werden. Um die Berechnung für eine bestimmte Belegung der Variablen durchzuführen, *interpretieren* wir dieses Programm, d.h. wir traversieren den Rechengraph und führen die einzelnen Operationen aus.

In Aufgabe 3 haben wir eine Transformation des Rechengraphen durchgeführt, nämlich vom ursprünglichen Rechengraph ("vorwärts") zu einen Rechengraph der den Gradienten berechnet ("rückwärts"). Das ist ein Beispiel einer *Programmtransformation*.

In dieser Aufgabe führen wir eine andere Programmtransformation durch: Wir implementieren einen Compiler, der einen MLL-Programm (d.h. den Rechengraph) in die populäre Compilerzwischendarstellung LLVM übersetzt.[1] Mithilfe der LLVM-Toolchain können wir dieses erzeugte LLVM-Programm automatisch weiter optimieren (lassen) und schließlich nativen Code erzeugen (lassen), den deine CPU direkt ausführen kann. Das ist dann deutlich schneller!

a) Schaue dir nochmal den Ausdruck

$$(x + 5) + (2 + 3)$$

an. Erstelle den Rechengraph `out` und generiere ein LLVM-Programm mittels:

```
Util.llvm(out, "4a");
```

(i) Schaue dir das erzeugte LLVM-Programm an und versuche es zu verstehen.

(ii) Welche LLVM-Anweisungen entsprechen den Operationen des ursprünglichen Rechengraphen?

[1] siehe https://llvm.org/docs/LangRef.html

(iii) Wie kommuniziert das LLVM-Programm seine Ein- und Ausgaben?

b) Nun wollen wir das erzeugte LLVM-Programm in nativen Maschinencode übersetzen und mit dem C-Programm `main.c` *binden*, welches die Eingabe(n) von der Kommandozeile liest, die erzeugte `mll`-Funktion aufruft, und die Ausgaben ausgibt. Rufe dazu den Compiler Clang auf:

```
Util.clang(out, "4a");
```

Das kompilierte Programm liegt nun unter `out/bin/4a`. Du kannst es entweder von der Kommandozeile aufrufen (nur Linux)

```
./4a 5
```

oder direkt von Java (alle Systeme)

```
Util.runBinary("4a", "5")
```

Teste diese Schritt. Wenn du möchtest, schaue dir anschließend das Programm `main.c` an und versuche zu verstehen, wie es funktioniert.

c) Mach' dich mit `Op.llvm` and `Add.llvm_` vertraut. Hier wirst du auf Muster stoßen, die du bereits kennst.

(i) Implementiere `Mul.llvm_`. Kopiere dazu einfach die Implementierung von `Add.\ llvm_` und versuche anhand der LLVM-Referenz (oder durch gutes Raten) herauszufinden, in welche LLVM-Anweisung du die MLL-Multiplikation übersetzen musst.

(ii) Erstelle nun ein Test-MLL-Programm für den Ausdruck $3xy$,

(iii) berechne den Gradienten,

(iv) erzeuge anschließend das entsprechende LLVM-Programm `4c.ll` und

(v) erzeuge eine ausführbare Datei `4c`.

Teste nun das frisch kompilierte Programm mit:

```
./4c -b 3 5
```

bzw.

```
Util.runCommand("4c", "-b", "3", "5")
```

Der Schalter `-b` sagt, dass das Programm nicht nur *ein* Ergebnis berechnet sondern auch den Gradienten. Wenn du alles richtig gemacht hast, solltest du folgenden Output sehen:

```
mll(3.000000, 5.000000)
=> (45.000000, 15.000000, 9.000000)
```

d) Implementiere nun die fehlenden `llvm_`-Methoden für weitere Operatoren. Schaue dazu in der LLVM-Referenz nach, welche LLVM-Anweisungen du dazu am besten verwendest. Einige MLL-Operatoren lassen sich nicht direkt durch LLVM-Anweisungen darstellen; diese werden über sogenannte *intrinsische* Funktionen modelliert.[2]

Teste deine Implementierung, indem du die Programme aus den vorherigen Ausgaben kompilierst und mit den Ergebnissen des Interpreters vergleichst.

[2] Diese besprechen wir vor Ort.

5 Finale

Wichtig: Bevor du mit der Bearbeitung der Aufgabe beginnen kannst, kopiere die Klasse `shared/mll/out/java/LogReg.java` in den Ordner `shared/mll/src/mll/`.

Die Klasse `LogReg` implementiert eine einfaches Verfahren für logistische Regression unter Verwendung von MLL.

a) Führe die Klasse aus und schaue dir das Ergebnis (gelernte Gewichte und Vorhersagen) an. Ist der erhaltene Klassifikator brauchbar? Schaue dir dazu auch die Daten und Visualisierungen der Daten an.

b) Experimentiere mit verschiedenen Daten, Anzahl Epochen und Lernraten. Du findest einige einfache Beispieldatensätze unter `mll/out/data`.

c) Schaue dir die generierten Rechengraphen und den Quelltext an. Versuche zu verstehen, was hier passiert.

d) Manchmal schlägt das Verfahren fehl, d.h. die Ergebnisse sind `NaN`. Kannst du herausfinden, warum und wann das passiert?