Mesterséges Intellingencia

házifeladat dokumentáció

Bella Jakab (JOF6SB), Juhász Bálint (CGQ956)

Feladat leírása:

A mesterséges intelligencia számos alterületén, főleg a gépi tanulásnál gyakran merül fel feladatként első- illetve magasabb rendű deriváltak kiszámítása (például optimalizációs feladatok megoldása során). Ez egyszerűbb esetekben papíron elvégezhető, bonyolult függvényeknél gyakran numerikus módszerek alkalmazására kerül sor, amelyek azonban a kerekítési hibákon keresztül jelentősen ronthatják a pontosságot. Használhatók továbbá szimbolikus módszerek, amelyek a deriválási szabályok ismételt alkalmazásával a kiszámítják a formula deriváltját; az eredményként kapott formulát implementálva a gépi pontosságnak megfelelő eredményhez juthatunk. Itt azonban problémát jelenthet bonyolult függvények formulával való leírása, valamint a formulák esetenként hatalmas mérete. Az automatikus deriválás ezzel szemben közvetlenül az implementáció (kód) szintjén végzi el a függvények deriválását. A C++ nyelvi elemeinek kreatív használatával elérhető, hogy a függvények deriváltját a compiler fordítási időben meghatározza és a binárisba már ez kerüljön (template metaprogramozás). A feladat során előre-módú automatikus deriválást kell megvalósítani template metaprogramozással; pluszpont jár a hátra-módú módszer implementációjáért. A feladatra a C++-t ismerő hallgatók jelentkezését várom, akik tisztában vannak az osztálysablonok fogalmával és használatával.

Feladat megoldásának menete:

Először arra gondoltunk, hogy a feladat egy compile time deriváló megvalósítása, azaz a teljes műveletet fordítási idő alatt kell kiszámolni. Erre már létezett implementáció, melyet a következőképpen lehet használni:

// 5+3\*sin(x+3.1415)

using fv =

plus<

Constant<5>,

multiplies<

Constant<3>,

sin\_t<

plus<

Constant<31415, -4>,

var\_x

>

>

>

>;  
constexpr double derivative = fv::value(0);

Miután egyeztettünk a konzulenssel, kiderült, hogy ennyire nem kell compile time-nak lennie a függvénynek, inkább csak magát a kifejezés fáját állítsa elő ez alatt az idő alatt.

Szerencsére a meglévő módszert könnyen át lehet ültetni egy félig runtime megoldásba, amely már megfelel a kiírásnak.

Ehhez fel kellett használni pár forrást az előre módú deriválásról.

A programot úgy alkottuk meg, hogy könnyű legyen a használata. Ehhez nyújt segítséget az expression template megoldás. Ha felhasználók nem szeretnének bonyolult és hosszú típust konstruálni, majd ugyanannak még a konstruktorát is meghívni, akkor szükség van egy átlátható megoldásra:

auto fx = x + 5;

Szerettük volna ha ez a kód működne, nem akartuk túlbonyolítani.

Konzultáció után kiderült, hogy egy nehezítést mégis bele kell raknunk, mivel végtelen számú változóval kell működnie a programnak. Emiatt a változókat nem definiálhattuk előre, kissé több kell, hogy látszódjon a program belső felépítéséből, ezt a kompromisszumot sajnos tartanunk kellett.

Most, hogy tudjuk, hogy nézzen ki a programunk interfésze, meg kell valósítanunk a belső működést is.

Először vegyük a program alapját a konstansokat és az ismeretleneket.

template <int ID = 0>

struct symbol {

using value = unknown<ID>;

};

Az ismeretleneket az itt látható módon lehet létrehozni. Minden ismeretlennek van egy saját ID-ja ami azt határozza meg, hogy ő melyik változó az egyenletben. Így tudunk később parciálisan deriválni.

Amikor kiadjuk az ’x+5’ parancsot, természetesen egy ’operator+’ hívódik meg, de mivel az ’x’ nem POD, a fordító hibát fog adni. Ezt a helyzetet használjuk ki, overload-oljuk az ’operator+’-t az ’x’ típusával és visszaadunk valami számunkra kedves csomagoló struktúrát. Innen már könnyű. Az ’operator+’-t overload-oljuk a csomagolóstruktúrára és többet nem is kell vele törődnünk.

Lássuk, hogy működik ez a gyakorlatban:

template<class SubExpr>

add<constant, SubExpr> operator+(const constant c, const SubExpr se)

{

return add<constant, SubExpr>{c, se};

}

Egy olyan operátort definiáltunk, ami egy konstanshoz jobb oldalról egy csomagoló osztályt ad. Az így létrejövő típus egy olyan csomagoló osztály lesz aminek a bal ágán egy konstans áll, a jobb ágán pedig szintén egy csomagoló osztály. Ezen a ponton sérül a compile time kikötés.

Jól látható, hogy egy fa struktúrát hozunk létre a kifejezésből. Amennyiben a fa gyökerére meghívjuk a deriváló függvényt, annak végig kell szaladnia az összes ágán és visszaadni a derivált eredményét.

template <class OperandA, class OperandB>

struct add: public binary\_operator<OperandA, OperandB> {

using bin\_op\_inst = binary\_operator<OperandA, OperandB>;

add(const OperandA a, const OperandB b): bin\_op\_inst(a, b) {}

double diff(int id, double x) const {

return bin\_op\_inst::a.diff(id, x) + bin\_op\_inst::b.diff(id, x);

}

};

Itt történik a logika. Az ’add’ osztály miután létrejön, eltároljuk a változóit és ha majd egyszer meghívjuk a kifejezésre a diff függvényt (amennyiben minden részkifejezésre implementálva van a függvény, nincs probléma), a ’diff’ metódus fog lefutni a deriválás szabályai szerint.

Egy kis kitérő, hogy miért sérül a compile time kikötés:

Ha már elkezdünk expression template-et használni, rögtön sérül, hiszen az operátornak nem POD-ot adunk hanem egy osztály példányát amit nem tud fordítási időben meghatározni a fordító. (Az első kódrészlet működik csak így)

De tegyük fel, hogy valahogyan megoldottuk, hogy egy osztályt egy függvény fordítási időben átvegyen ha nincs benne változó. Ekkor újabb akadályba ütközünk, mégpedig abba, hogy template paraméternek nem lehet megadni lebegőpontos számot. Ha szeretnénk 3.14-et a ctd::constant osztályon belül template paraméterként tárolni, a fordító hibát adna.

Most, hogy készen vagyunk az összes aritmetikai osztállyal és overload-olt operátorral, megírhatjuk még a függvényeket.

Példaként nézzük meg a szinuszfüggvény kezelőjét!

template <class SubExpr>

**struct** func\_sin {

**const** SubExpr arg;

func\_sin(**const** SubExpr arg): arg{arg} {}

**double** value(**int** id, **double** x) **const** {

**return** std::sin(arg.value(id, x));

}

**double** diff(**int** id, **double** x) **const** {

**return** std::cos(arg.value(id, x)) \* arg.diff(id, x);

}

std::string printValue(**int** id) **const** {

std::stringstream ss;

ss << "sin(" << arg.printValue(id) << ")";

**return** ss.str();

}

std::string printDiff(**int** id) **const** {

std::stringstream ss;

ss << "cos(" << arg.printValue(id) << ")\*(" << arg.printDiff(id) << ")";

**return** ss.str();

}

};

Az egyetlen tagváltozó az “arg”, ami a függvény “hasában” található kifejezés. A használt struktúra miatt vissza kell tudjuk adni a függvény (sin(arg)) értékét és deriváltját, ezek természetesen újabb kifejezések is lehetnek.

A value függvény egyszerűen visszaadja a tárolt kifejezés értékét, a megadott sorszámú változó szerint. A továbbiakban nem írjuk le külön – minden esetben megadható, hogy éppen melyik változó szerinti értéket kérjük le.  
A diff függvény a deriváltat adja vissza, hiszen: sin(arg)’ = cos(arg)\*(arg’)  
Ezeknek a függvényeknek megvalósítottuk egy kiíró verzióját is, ami C++ stringgel tér vissza. Ez a formázásnál jön jól.

Ennyi azonban nem elég, a struktúrát ki kellett, hogy egészítsük néhány függvénnyel, hogy ne kaphassunk hibát, és mindig a specifikációnak megfelelően működjön a program.

A használt megoldás miatt gondoskodnunk kell pl. arról, hogy ha int-tel hívják meg a függvényt, azt előbb doble-be kasztoljuk.

template <

typename SubExpr,

typename output\_type = func\_sin<SubExpr>

>

output\_type sin(**const** SubExpr a){

**return** func\_sin<SubExpr>{a};

}

func\_sin<constant> sin(**double** a){

**return** func\_sin<constant>{constant{a}};

}

**auto** sin(**int** a)

-> decltype(sin(static\_cast<**double**>(a)))

{

**return** sin(static\_cast<**double**>(a));

}

Lényegében a kifejezésekre és konstansokra meghívott szinuszokat átirányítjuk, hogy minden esetben a mi func\_sin osztályunkat (struktúránkat) használják.

A többi függvényre (cos, log) hasonlóan működik.

A műveletek megvalósítása:

Alapnak a binary\_operator struktúrát használjuk, melyet külön nem részletezünk – egyszerűen könnyebb rá építeni.

Példaként nézzük meg alaposabban az összeadást; a többi művelet is hasonlóan működik.

// add

**template** <

**class** OperandA,

**class** OperandB

>

**struct** add: **public** binary\_operator<OperandA, OperandB>, **public** negatable<add<OperandA, OperandB>> {

**using** bin\_op\_inst = binary\_operator<OperandA, OperandB>;

add(**const** OperandA a, **const** OperandB b):

bin\_op\_inst(a, b)

{}

**double** value(**int** id, **double** x) **const** {

**return** bin\_op\_inst::a.value(id, x) + bin\_op\_inst::b.value(id, x);

}

**double** diff(**int** id, **double** x) **const** {

**return** bin\_op\_inst::a.diff(id, x) + bin\_op\_inst::b.diff(id, x);

}

}

Az összeadás felbontható két tagváltozós műveletekre (alapesetben két, tovább nem bontható tagot adunk össze), így az összeadó struktúra a binary\_operator-ra épül, melynek két adattagja a és b.

Itt is meg kell adnunk az összeadás “értékét” és deriváltját. Egyszerű összeadás esetén ez a tagváltozók (kifejezések) összege illetve deriváltja.

A diff függvény pl. a szorzásnál máshogy működik (a’\*b + b’\*a), ott így néz ki:

**double** diff(**int** id, **double** x) **const** {

**return** bin\_op\_inst::a.diff(id, x) \* bin\_op\_inst::b.value(id, x)

+ bin\_op\_inst::a.value(id, x) \* bin\_op\_inst::b.diff(id, x);

}

Természetesen itt is megvalósítjuk a kiíráshoz szükséges függvényeket is:

std::string printValue(**int** id) **const** {

std::stringstream ss;

ss << "(" << bin\_op\_inst::a.print(id) << ")+(" << bin\_op\_inst::b.print(id) << ")";

**return** ss.str();

}

std::string printDiff(**int** id) **const** {

std::stringstream ss;

ss << "(" << bin\_op\_inst::a.printDiff(id) << ")+(" << bin\_op\_inst::b.printDiff(id) << ")";

**return** ss.str();

}

Hasonlóan működik a többi műveletre (szorzás, osztás, exponenciális) is.

Ez azonban természetesen nem elég, hiszen az alapértelmezett műveleteket felül kell írni, hogy az itt megvalósított függvényeket használják. Ennek részletes kifejtése rengeteg ismétlődést tartalmazna, ezért itt is csak példaként az összeadást és a kivonást mutatjuk be.

Mindenek előtt a negatív előjelet visszavezettük a -1-gyel való szorzásra:

// -expr

**template** <**typename** ChildType>

multiply<constant, ChildType> negatable<ChildType>::**operator**-() **const**

{

**return** multiply<constant, ChildType>{-1, **static\_cast**<**const** ChildType&>(\***this**)};

}

Ezután felül kell definiálni az operátorokat, hogy minden esetben a mi összeadófüggvényünk fusson le.

A három lehetőség: kifejezés + kifejezés, konstans + kifejezés, kifejezés + konstans.  
Mivel a konstansok az összeadás mindkét oldalán állhatnak, ténylegesen meg kell valósítani mindkét verziót. A többi műveletnél a két verzió “belseje” különbözni fog.

// expr + expr

**template**<

**class** SubExprA,

**class** SubExprB,

**class** add\_inst = add<SubExprA, SubExprB>,

**typename** std::enable\_if<

std::is\_pod<SubExprA>::value == **false** && std::is\_pod<SubExprB>::value == **false**

>::type\* = **nullptr**

>

add\_inst **operator**+(**const** SubExprA sea, **const** SubExprB seb){

**return** add\_inst{sea, seb};

}

// const + expr

**template**<

**class** SubExpr,

**class** add\_inst = add<constant, SubExpr>

>

add\_inst **operator**+(**const** constant c, **const** SubExpr se)

{

**return** add\_inst{c, se};

}

// expr + const

**template**<

**class** SubExpr,

**class** add\_inst = add<SubExpr, constant>

>

add\_inst **operator**+(**const** SubExpr se, **const** constant c)

{

**return** add\_inst{se, c};

}