Demonstracijski okvir za diferencijabilno programiranje

Novak, Jakov

Undergraduate thesis / Završni rad

2024

Degree Grantor / Ustanova koja je dodijelila akademski / stručni stupanj: **University of Zagreb, Faculty of Electrical Engineering and Computing / Sveučilište u Zagrebu, Fakultet elektrotehnike i računarstva**

Permanent link / Trajna poveznica: https://urn.nsk.hr/urn:nbn:hr:168:739472

Rights / Prava: In copyright / Zaštićeno autorskim pravom.

Download date / Datum preuzimanja: **2025-03-20**

Repository / Repozitorij:

FER Repository - University of Zagreb Faculty of Electrical Engineering and Computing repozitory

SVEUČILIŠTE U ZAGREBU
FAKULTET ELEKTROTEHNIKE I RAČUNARSTVA

ZAVRŠNI RAD br. 1655

DEMONSTRACIJSKI OKVIR ZA DIFERENCIJABILNO PROGRAMIRANJE

Jakov Novak

Zagreb, lipanj 2024.

SVEUČILIŠTE U ZAGREBU
FAKULTET ELEKTROTEHNIKE I RAČUNARSTVA

ZAVRŠNI RAD br. 1655

DEMONSTRACIJSKI OKVIR ZA DIFERENCIJABILNO PROGRAMIRANJE

Jakov Novak

Zagreb, lipanj 2024.

SVEUČILIŠTE U ZAGREBU
FAKULTET ELEKTROTEHNIKE I RAČUNARSTVA

Zagreb, 4. ožujka 2024.

ZAVRŠNI ZADATAK br. 1655

Pristupnik:	**Jakov Novak (0036543546)**
Studij:	Elektrotehnika i informacijska tehnologija i Računarstvo
Modul:	Računarstvo
Mentor:	prof. dr. sc. Siniša Šegvić

Zadatak: **Demonstracijski okvir za diferencijabilno programiranje**

Opis zadatka:

Diferencijabilno programiranje proučava formalno izražavanje parametriziranih algoritama koji omogućavaju automatsko računanje parcijalnih derivacija izlaza s obzirom na ulaze i parametre. Diferencijabilni programi mogu učiti iz podataka odnosno prilagoditi se željenoj domeni gradijentnom optimizacijom parametara. Ova programska paradigma postaje nezaobilazni sastojak mnogih modernih pristupa strojnom učenju i računarskoj znanosti. U okviru rada, potrebno je upoznati temeljne postavke programskog oblikovanja, matematičke analize i linearne algebre. Proučiti i ukratko opisati postojeće pristupe za provođenje automatskog diferenciranja. Osmisliti demonstracijsku platformu i predložiti izvedbu u programsko jeziku C++. Predložiti pravce za budući rad. Radu priložiti izvorni i izvršni kod razvijenih postupaka, ispitne slijedove i rezultate, uz potrebna objašnjenja i dokumentaciju. Citirati korištenu literaturu i navesti dobivenu pomoć.

Rok za predaju rada: 14. lipnja 2024.

Zahvaljujem profesoru Šegviću na suradnji tijekom pisanja rada te obitelji, prijateljima i kolegama na podršci tijekom studija.

Sadržaj

1. Uvod

Većina klasičnih postupaka u dubokom učenju se svode na optimizaciju izlaza neuralne mreže s obzirom na neku funkciju gubitka. Kod takvih postupaka je ključan izračun gradijenta te funkcije kako bi se podesile značajke mreže:

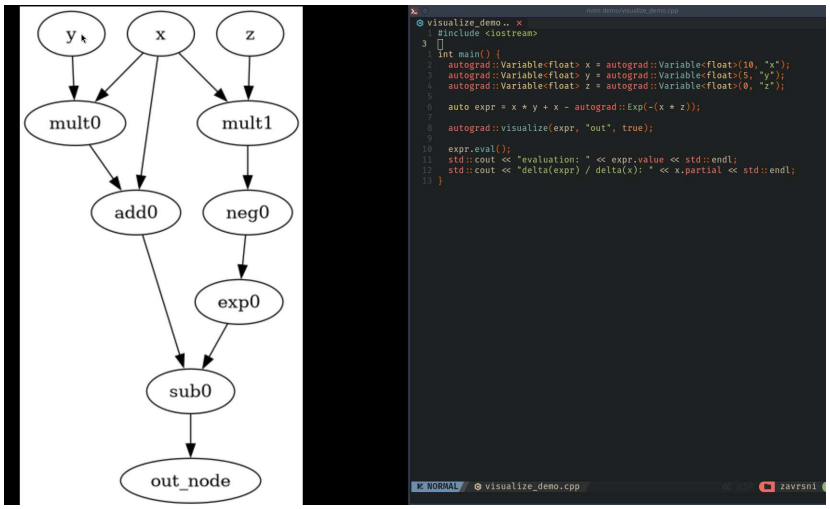
$$\nabla(f \circ L)(\text{ulaz}) \tag{1.1}$$

Postoje različiti pristupi u korištenju računala za rješavanje tog problema. Neki od glavnih su: simboličko diferenciranje, numeričke metode i automatsko diferenciranje. Glavni nedostatci prvih spomenutih pristupa su prevelika složenost dobivenih izraza (netraktabilnost) [1] i nepreciznost brojeva s pomičnim zarezom [2]. Ovaj rad je fokusiran na proučavanju metoda automatskog diferenciranja te implementacije jednostavnog demonstracijskog okvira u programskom jeziku C++ uz pomoć biblioteke OpenCL za matrične operacije.

2. Korišteni algoritmi i matematički aparat

2.1. Metode automatskog diferenciranja

Glavna ideja automatskog diferenciranja jest da dobivenu matematičku funkciju možemo prikazati kao aciklički digraf kojemu su čvorovi varijable, konstante ili međurezultati: $G = (V, E).V = \text{Var} \cup F$, gdje je Var skup varijabli, a F skup primitivnih funkcija koje se lako mogu derivirati. Ulaz neke funkcije $f \in F$ je definiran kao: $\{x \in V : (x, f) \in E\}$



Slika 2.1. Primjer grafa funkcije: $f(x, y, z) = x * y + x - \exp(-x * z)$

Općenito, za neku funkciju: $f : \mathbb{R}^n \to \mathbb{R}^m$ vrijedi da će njezin graf imati n čvorova varijabli, odnosno n izvora te m izlaznih čvorova, tj. m ponora. Dodatno možemo pojednostaviti njezin graf ako funkciju zamislimo kao $f : \mathbb{V}^n \to \mathbb{V}^m$ te zatim predstavimo sve varijable kao vektore. Ta ideja će nam uvelike pojednostaviti samu implementaciju autograd programa jer će apstrahirati prijelaz s univarijatnih na multivarijatne funkcije.
Sa slike 2.1. možemo već naslutiti koja je glavna ideja algoritma; Glavna ideja jest da

obiđemo zadani graf te u svakom čvoru odredimo derivaciju po pravilu ulančavanja:

$$\frac{\partial(f_1 \circ (f_2 \circ \cdots (f_{n-1} \circ f_n)))}{\partial x} = \frac{\partial f_1}{\partial f_2}\frac{\partial f_2}{\partial f_3} \cdots \frac{\partial f_{n-1}}{\partial f_n}\frac{\partial f_n}{\partial x} \tag{2.1}$$

Čvorove u grafu možemo na slijedeći način predstaviti u objektnoj paradigmi (programski jezik *Python*) [3]:

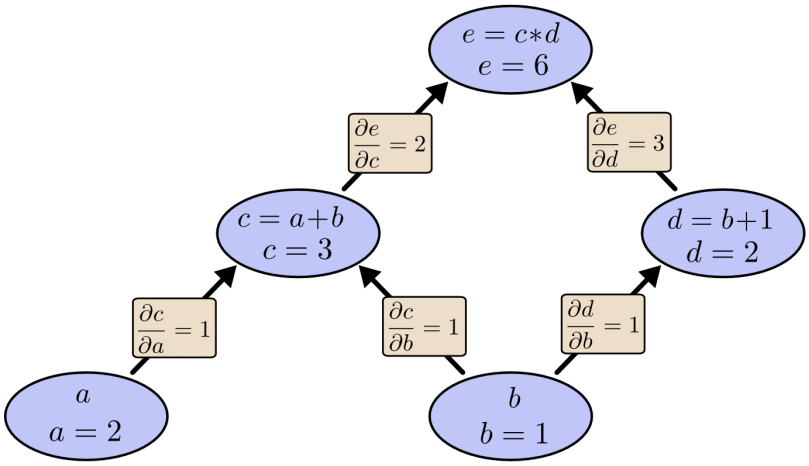
```
class ValueAndPartial:
    def __init__(self, value, partial):
        self.value = value
        self.partial = partial

    def toList(self):
        return [self.value, self.partial]

class Expression:
    def __add__(self, other):
        return Plus(self, other)

    def __mul__(self, other):
        return Multiply(self, other)
```

2.1.1. Forward mode

Ovdje postoje dva pristupa: prvi, takozvani *forward mode accumulation*, jest da počnemo od čvorova ulaza prema čvorovima izlaza, po formuli:

$$\frac{\partial(f_{i-1} \circ f_i)}{\partial x} = \frac{\partial f_{i-1}}{\partial x}\frac{\partial f_i}{\partial f_{i-1}} \tag{2.2}$$

Vrijednost f_{i-1} je u početku jednaka 1, a kasnije se propagira kroz slojeve grafa 2.2. kao vrijednost koju u literaturi često nazivaju *seed value* [5]. Ovaj pristup je pogodan ukoliko je broj ulaza daleko manji od broja izlaza, odnosno vrijedi $f : \mathbb{R}^n \to \mathbb{R}^m . n << m$. Složenost ovog koda ovisi o strukturi grafa, no vidljivo je da prolazak kroz graf ponavljamo n puta te je stoga poželjno da je n mnogo manji od m. Ovaj pristup nije idealan za strojno



Slika 2.2. Unaprijedni, *forward mode accumulation* pristup [4]

učenje pošto je često ulazna domena mnogo veća od izlazne u toj primjeni.

Listing 2..1: Primjer implementacije unaprijedne metode u programskom jeziku *Python* [3]

```
class Variable(Expression):
    def __init__(self, value):
        self.value = value

    def evaluateAndDerive(self, variable):
        partial = 1 if self == variable else 0
        return ValueAndPartial(self.value, partial)

class Plus(Expression):
    def __init__(self, expressionA, expressionB):
        self.expressionA = expressionA
        self.expressionB = expressionB

    def evaluateAndDerive(self, variable):
        valueA, partialA =
            self.expressionA.evaluateAndDerive(variable).toList()
        valueB, partialB =
            self.expressionB.evaluateAndDerive(variable).toList()
        return ValueAndPartial(valueA + valueB, partialA + partialB)
```

```
class Multiply(Expression):
    def __init__(self, expressionA, expressionB):
        self.expressionA = expressionA
        self.expressionB = expressionB

    def evaluateAndDerive(self, variable):
        valueA, partialA =
            self.expressionA.evaluateAndDerive(variable).toList()
        valueB, partialB =
            self.expressionB.evaluateAndDerive(variable).toList()
        return ValueAndPartial(valueA * valueB, valueB * partialA + valueA *
            partialB)
```

2.1.2. Reverse mode

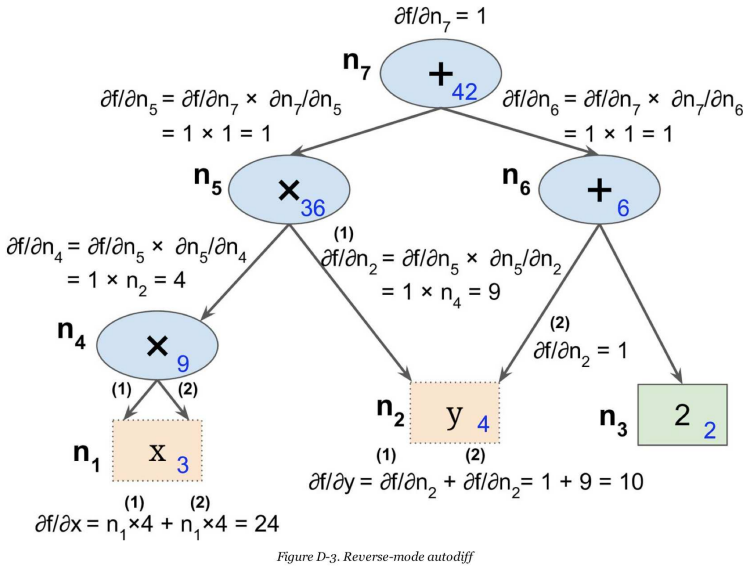
Drugi pristup problemu automatskog diferenciranja jest takozvani *reverse mode accumulation*, tj. prolazak grafa unazad:

$$\frac{\partial(f_i \circ f_{i+1})}{\partial x} = \frac{\partial f_{i+1}}{\partial f_i}\frac{\partial f_i}{\partial x} \tag{2.3}$$

U ovom pristupu se *seed*-vrijednost šalje unazad kroz graf jednadžbe od ponora prema izvorima 2.3. Iz toga slijedi da će za neku funkciju $f : \mathbb{R}^n \to \mathbb{R}^m . m << n$ algoritam biti najefikasniji zato što će se m puta pozvati funkcija prolaska kroz graf. Poseban slučaj ovog pristupa problemu automatskog diferenciranja jest temeljni princip koji se koristi za unazadnu propagaciju (*eng. backpropagation*) u strojnom učenju.
Zanimljivo je da su oba algoritma optimalna samo za jednu određenu domenu primjene. To je zato što je problem izračuna Jakobijana za proizvoljnu funkciju $f : \mathbb{R}^n \to \mathbb{R}^m$ s minimalnim brojem računskih operacija problem koji je NP-potpun. Problem je poznat i kao *optimal Jacobian problem* [7].

Listing 2..2: Primjer implementacije unazadne metode u programskom jeziku *Python* [3]

```
class Variable(Expression):
    def __init__(self, value):
```



Slika 2.3. Unazadni, *reverse mode accumulation* pristup [6]

```
        self.value = value
        self.partial = 0

    def evaluate(self):
        pass

    def derive(self, seed):
        self.partial += seed

class Plus(Expression):
    def __init__(self, expressionA, expressionB):
        self.expressionA = expressionA
        self.expressionB = expressionB
        self.value = None

    def evaluate(self):
        self.expressionA.evaluate()
        self.expressionB.evaluate()
        self.value = self.expressionA.value + self.expressionB.value
```

```
    def derive(self, seed):
        self.expressionA.derive(seed)
        self.expressionB.derive(seed)

class Multiply(Expression):
    def __init__(self, expressionA, expressionB):
        self.expressionA = expressionA
        self.expressionB = expressionB
        self.value = None

    def evaluate(self):
        self.expressionA.evaluate()
        self.expressionB.evaluate()
        self.value = self.expressionA.value * self.expressionB.value

    def derive(self, seed):
        self.expressionA.derive(self.expressionB.value * seed)
        self.expressionB.derive(self.expressionA.value * seed)
```

2.2. Dualni brojevi

Dualni brojevi su još jedan potenncijalni pristup za lako rješavanje prve derivacije jednadžbe. Iako nećemo koristiti ovaj pristup u implementaciji unazadnog (*backward mode*) diferenciranja, valjalo bi ga spomenuti kao jednog od najintuitivnijih pristupa za rješavanje *autodiffa* kod unaprijednog (*forward mode*) diferenciranja. O čemu se radi? Zamislimo da bilo koji realni broj možemo zapisati kao dualni broj x [8]:

$$\begin{aligned} x &= a + b\varepsilon \\ \varepsilon^2 &= 0, \varepsilon \neq 0 \\ a, b &\in \mathbb{R} \end{aligned} \tag{2.4}$$

U formuli 2.4 broj a predstavlja realni broj, početnu vrijednost neke varijable ili izraza, dok b predstavlja diferencijal kojeg propagiramo od varijabli do rezultata određene funkcije. Ovakav pristup je memorijski jednostavniji jer ne zahtjeva slaganje složenih grafova, već računamo s sve kao i s "normalnim" brojevima. To vrijedi zato što se svaka analitička funkcija lako može proširiti u domenu dualnih brojeva preko svojeg Taylorovog reda [8]:

$$f(a + b\varepsilon) = \sum_{n=0}^{\infty} \frac{f^{(n)}(a) b^n \varepsilon^n}{n!} = f(a) + bf'(a)\varepsilon \tag{2.5}$$

Jednostavna implementacija ove ideje za funkcije zbrajanja i množenja izleda ovako u programskom jeziku *Python*:

Listing 2..3: Primjer korištenja dualnih brojeva za unaprijednu metodu *autodiffa* [3]

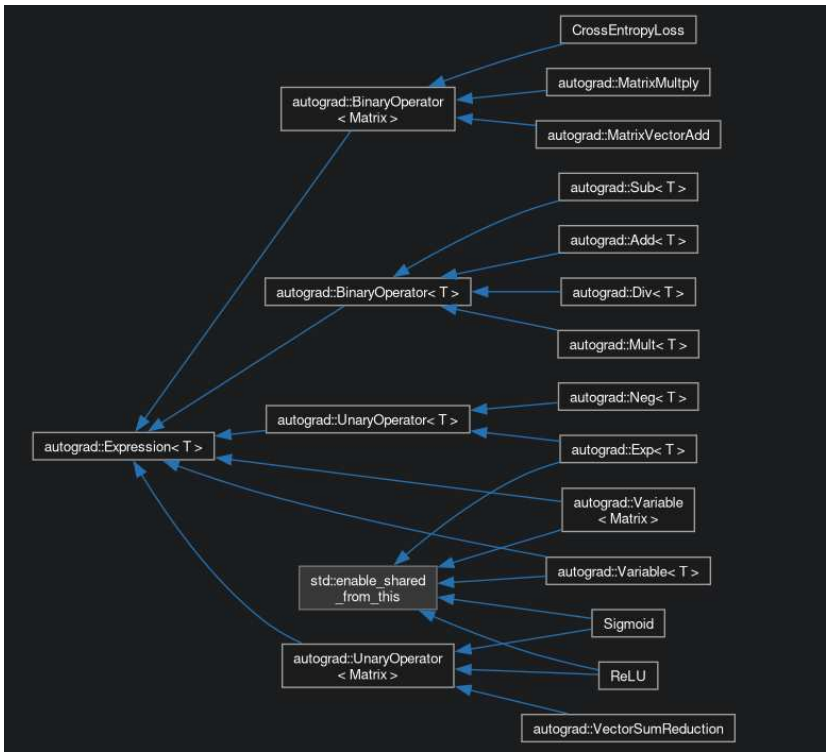
```
class Dual:
    def __init__(self, realPart, infinitesimalPart=0):
        self.realPart = realPart
        self.infinitesimalPart = infinitesimalPart


    def __add__(self, other):
        return Dual(
            self.realPart + other.realPart,
            self.infinitesimalPart + other.infinitesimalPart
        )
```

```
    def __mul__(self, other):
        return Dual(
            self.realPart * other.realPart,
            other.realPart * self.infinitesimalPart + self.realPart *
                other.infinitesimalPart
        )

# Primjer: Gradijent funkcije: z = x * (x + y) + y * y at (x, y) = (2, 3)
def f(x, y):
    return x * (x + y) + y * y
x = Dual(2)
y = Dual(3)
epsilon = Dual(0, 1)
a = f(x + epsilon, y)
b = f(x, y + epsilon)
print("parcijalno z po x =", a.infinitesimalPart) # Output: 7
print("parcijalno z po y =", b.infinitesimalPart) # Output: 8
```

2.3. Pristupi implementaciji grafa

2.3.1. Nadjačavanje operatora

Jedan način implementacije automatskog diferenciranja jest nadjačavanje operatora. To je način kojeg ćemo koristiti u ovom radu te koji je već bio pokazan u primjerima. Glavna ideja jest da imamo sučelje preko kojeg modeliramo izraz koji omogućava klijentu da zove metode *derive()* i *evaluate()*. Sve ostale operacije se realiziraju uz pomoć rekurzije, bilo unaprijedne ili unazadne, te nasljeđivanjem i implementacijom pravila ulančavanja za pojedine funkcije. Klijent takav sustav može na jednostavan način koristiti ako smo nadjačali operatore za osnovne računske operacije (eng. *operator overloading*). Ukoliko je to slučaj, klijent jednostavno može konstruirati varijable te zatim proizvoljne izrazi pisati bez da je svjestan pozadinskih operacija. Na takav način funkcionira Pytorchev *torch.autograd*, koji je glavni modul zaslužan za automatsko diferenciranje [9]. Na slici 2.4. je prikazan klasni dijagram takve implementacije.



Slika 2.4. Klasni dijagram implementacije automatskog diferenciranja uz nadjačavanje operatora u objektnoj paradigmi

2.3.2. Transformacija izvornog koda

Drugi, efikasniji način jest umetanje međukoda, kojeg optimirajući prevoditelj može dodatno obraditi kasnije, u naš kod, odnosno klijentov kod. Glavna ideja jest da imamo okvir koji matematičke izraze izravno prevodi u međukod za evaluaciju (zamjena za metodu *evaluate()*) te međukod za derivaciju (zamjena za metodu *derive()*). Ukoliko imamo takav okvir, klijent može lako preko njega prevesti matematičke izraze te njihov izračun umetnuti u svoj kod. To se opet može ostvariti preko objektnog modela. Primjer razvojnog okvira koji koristi ovakav pristup jest TensorFlow i Zygote.jl [5]. Zygote.jl koristi llvm međukod kako bi izravno, na temelju neke ulazne funkcije, izgenerirao naredbe u strojno neovisnom kodu niske razine koji se kasnije lako može optimirati:

Listing 2..4: Primjer korištenja Zygote.jl za generiranje llvm međukoda za funkciju: $a^2 + a * b$ [5]

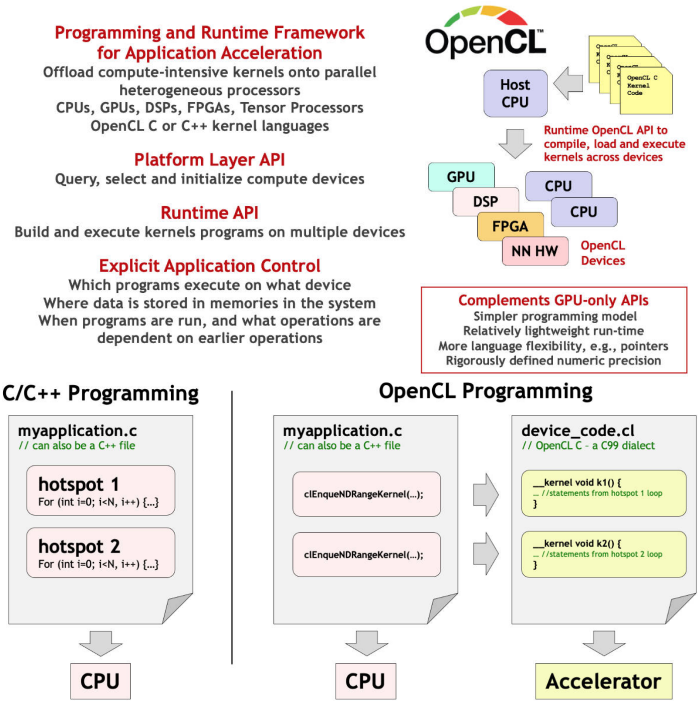
```
function P1(a::Float64, b::Float64) :: Float64
    return a^2 + a*b
end

@code_llvm P1(0.1, 1.0)
; a*a
%2 = fmul double %0, %0
; a*b
%3 = fmul double %0, %1
; a*a + a*b
%4 = fadd double %2, %3
ret double %4

@code_llvm gradient(P1, 0.1, 1.0)
; gradient = [2*a + b, a]
; a + a
 %3 = fadd double %1, %1
; 2*a + b
 %4 = fadd double %3, %2
 %.sroa.0.0..sroa_idx = getelementptr inbounds [2 x double], [2 x double]*
```

```
%0, i64 0, i64 0
; Res[0] = 2*a + b
store double %4, double* %.sroa.0.0..sroa_idx, align 8
 %.sroa.2.0..sroa_idx4 = getelementptr inbounds [2 x double], [2 x double]*
%0, i64 0, i64 1
; Res[1] = a
store double %1, double* %.sroa.2.0..sroa_idx4, align 8
ret void
```

2.4. OpenCL i C++

Do sad smo prošli kroz ideju automatskog diferenciranja, zašto i kako ga implementirati, ipak nešto nam još fali. Zadnji dio slagalice je podsustav za linearnu algebru, pošto su varijable u kontekstu strojnog učenja često vektori, odnosno matrice. Taj dio sustava ćemo u ovom radu implementirati uz pomoć OpenCL-a, zajedno s ručno napisanim jezgrama za računanje na grafičkoj kartici. Ključno pitanje je zapravo što je OpenCL i zašto ga koristimo?

OpenCL je standard otvorenog koda za paralelno programiranje heterogenih računalnih sustava. To konkretno znači da u okviru OpenCL-a pišemo kod u obliku takozvanih *jezgri* (eng. *kernel*) koji se pokreće više puta na našem akceleratoru (grafička, fpga pločica, višeprocesorski sustav, itd.) te na taj način postižemo grubozrnati paralelizam na podatkovnoj razini. Takav pristup je efikasan ukoliko imamo velik broj podataka koji se mogu međusobno neovisno obraditi. Ako to nije slučaj, gubimo vrijeme na prijenos podataka ili na međuovisnosti. Operacije s matricama su idealan obrazac upotrebe ovog okvira te ćemo ga stoga koristiti za prijenos podataka na grafičku karticu, izračun na grafičkoj te zatim prijenos na glavno računalo 2.5.



Slika 2.5. Grafički prikaz organizacije OpenCL sustava [10]

3. Glavni dio

3.1. Autograd_core implementacija

Sad, kad znamo teorijsku podlogu, možemo krenuti u implementaciju `autograd` sustava. Ovaj sustav će se temeljiti na principu unatražnog prolaska 2.1.2. Najprije implementiramo razred `autograd::Expression` koji definira neke okvirne metode za `cacheiranje` već izračunatih vrijedosti te dodatne metode za vizualizaciju grafova funkcije 2.1. Nakon toga možemo implementirati osnovne funkcionalnosti naslijeđivanjem razreda te nadjačavanjem metoda `autograd::Expression::eval` te `autograd::Expression::derive`. Jedini implementacijski detalj koji je možda malo nejasan iz koda jest korištenje pametnih pokazivača. Potreba za time dolazi zato što ne računamo direktno derivaciju funkcije već gradimo novi izraz tipa `autograd::Expression` kojeg zatim evaluiramo (kako bi mogli računati derivacije višeg reda) pa je stoga potrebno dinamički alocirati / oslobađati memoriju za vrijeme izvođenja programa. Pametni pokazivači, konkretno `std::shared_ptr<autograd::Expression>` su idealni za ovu primjenu. Ovakva implementacija zajedno s nadjačavanjem operatora 2.3.1. može bez problema računati gradijente funkcija više varijabli, ali samo kad su te varijable skalari. Za potrebe dubokog učenja trebamo proširiti tu funkcionalnost kako bi nam varijable mogle biti matrice ili vektori kojima prikazujemo težine neuralnih mreža i njihove ulaze. U idućem poglavlju ćemo se baviti tom problematikom.

Listing 3..1: Jezgrena funkcionalnost autograda

```
/**
 * @brief struktura koja predstavlja apstraktni matematički izraz ili
    varijablu. Generalno ju možemo zamisliti kao čvor u grafu funkcije
 *
 * @tparam T tip varijabli, najčešće float ili Matrix
```

```
 */
template <typename T>
struct Expression {
  /**
   * @brief izračunata vrijednost izraza
   */
  T value;
  /**
   * @brief zastavica kojom pamtimo je li izraz već izračunat
   */
  bool evaluated = false;
  /**
   * @brief zastavica kojom pamtimo treba li računati parcijalnu derivaciju
       za taj (pod)izraz
   */
  bool requires_grad = true;

  /**
   * @brief virtualna metoda kojom se definira evaluiranje izraza
   */
  virtual void eval() = 0;
  /**
   * @brief metoda koja provjerava jesmo li već odredili graf derivacije te
       ovisno o tome zove _derive ili vraća graf u obliku Expression<T>
   *
   * @param seed vrijednost koja se propagira unazad kao parcijalna
       derivacija
   * @param out_map referenca na mapu parcijalnih derivacija za sve
       varijable s requires_grad = true
   */
  void derive(std::shared_ptr<Expression<T>> seed,
      std::unordered_map<std::string, std::shared_ptr<Expression<T>>>
      &out_map) {
    if(!this->requires_grad)
```

```
      return;

    this->_derive(seed, out_map);
  }
  /**
   * @brief funkcija koja rekurzivno gradi graf parcijalnih derivacija
   *
   * @param seed vrijednost koja se propagira unazad kao parcijalna
       derivacija
   * @param out_map referenca na mapu parcijalnih derivacija za sve
       varijable s requires_grad = true
   */
  virtual void _derive(std::shared_ptr<Expression<T>> seed,
      std::unordered_map<std::string, std::shared_ptr<Expression<T>>>
      &out_map) = 0;
  virtual std::optional<std::shared_ptr<const Variable<T>>>
      find_variable(const std::string &name) const = 0;

public:
  /**
   * @brief funkcija koja vraća vrijednost trenutnog izraza
   *
   * @return referenca na this->value
   */
  T &getValue() {
    if(evaluated)
      return this->value;

    this->eval();
    this->evaluated = true;
    return this->value;
  }
  /**
   * @brief funkcija koja računa gradijent trenutnog izraza
```

```
 *
 * @return mapa gradijenta gdje su ključevi imena varijabli, a
     vrijednosti std::shared_ptr<Expression<T>>
 */
std::unordered_map<std::string, std::shared_ptr<Expression<T>>> grad() {
  auto out_map = std::unordered_map<std::string,
      std::shared_ptr<Expression<T>>>();
  this->derive(std::make_shared<Variable<T>>(1, "const 1", false),
      out_map);

  return out_map;
}

/**
 * @brief operator koji traži varijable po imenu
 *
 * @param name ime varijable
 * @return std::shared_ptr na varijablu
 * @throws std::logic_error ako varijabla nije nađena u izrazu
 */
std::shared_ptr<const Variable<T>> operator[](const std::string &name) {
  auto var = find_variable(name);
  if(!var.has_value())
    throw std::logic_error("variable doesn't exist!");

  return var.value();
}

/**
 * @brief funkcija koja sluzi dodavanju trenutnog cvora u graf
 * @see visualize.hpp
 *
 * @param graph pokazivac na graf
 * @param prev cvor koji se nalazi na izlazu trenutnog
```

```
   */
  virtual void addSubgraph(Agraph_t *graph, Agnode_t *prev) const = 0;
};

/**
 * @brief klasa koja implementira jednu varijablu
 */
template <typename T>
struct Variable : Expression<T>, public
    std::enable_shared_from_this<Variable<T>> {
  std::string name;

  /**
   * @brief konstruktor koji prima početnu vrijednost, ime varijable i
       zastavicu requires_grad
   *
   * @param value početna vrijednost
   * @param name ime varijable
   * @param requires_grad zastavica koja je istinita ako trebamo računati
       parcijalnu derivaciju po toj varijabli
   */
  Variable(const T &value, const std::string &name, bool requires_grad =
      true) {
    this->value = value;
    this->evaluated = true;
    this->name = name;
    this->requires_grad = requires_grad;
  }

  void eval() override { }
  void _derive(std::shared_ptr<Expression<T>> seed,
      std::unordered_map<std::string, std::shared_ptr<Expression<T>>>
      &out_map) override {
    if(out_map.find(this->name) != out_map.end()) {
```

```
      out_map[this->name] =
          std::static_pointer_cast<Expression<T>>(std::make_shared<Add<T>>(out_map[this->n
          seed));
    }
    else {
     out_map[this->name] = seed;
    }
  }

  void addSubgraph(Agraph_t *graph, Agnode_t *caller) const override {
    Agnode_t *curr = agnode(graph, (char *) name.c_str(), 1);
    if(caller != nullptr)
      agedge(graph, curr, caller, nullptr, 1);
  }

  virtual std::optional<std::shared_ptr<const Variable<T>>>
      find_variable(const std::string &name) const override {
    if(name == this->name)
      return std::optional(this->shared_from_this());
    else
      return std::nullopt;
  }
};

/**
 * @brief generička klasa koja implementira binarne operatore: \f$ y = f(a,
     b) \f$
 */
template <typename T>
struct BinaryOperator : Expression<T> {
  std::shared_ptr<Expression<T>> left, right;

  /**
   * @brief konstruktor koji prima dva izraza: \f$ a \f$ i \f$ b \f$
```

```
   *
   * @param left lijevi izraz tipa
       std::shared_ptr<autograd::Expression<T>>, \f$ a \f$
   * @param right desni izraz tipa
       std::shared_ptr<autograd::Expression<T>>, \f$ a \f$
   */
  BinaryOperator(std::shared_ptr<Expression<T>> left,
      std::shared_ptr<Expression<T>> right) : left{ left }, right{ right } {
    this->requires_grad = left->requires_grad || right->requires_grad;
  }

  virtual std::optional<std::shared_ptr<const Variable<T>>>
      find_variable(const std::string &name) const override {
    auto left_side = this->left->find_variable(name);
    auto right_side = this->right->find_variable(name);

    if(left_side.has_value())
      return left_side.value();
    else if(right_side.has_value())
      return right_side.value();
    else
      return std::nullopt;
  }
};

/**
 * @brief generička klasa koja implementira unarne operatore: \f$ y = f(x)
     \f$
 */
template <typename T>
struct UnaryOperator : Expression<T> {
  std::shared_ptr<Expression<T>> prev;

  /**
```

```
     * @brief konstruktor koji prima jedan izraz: \f$ x \f$
     *
     * @param prev izraz tipa std::shared_ptr<Expression<T>>
     */
    UnaryOperator(std::shared_ptr<Expression<T>> prev) : prev{ prev } {
      this->requires_grad = prev->requires_grad;
    }

    virtual std::optional<std::shared_ptr<const Variable<T>>>
        find_variable(const std::string &name) const override {
      return this->prev->find_variable(name);
    }
  };
}
```

3.2. Matrični račun

3.2.1. Operacije *element po element*

Pošto su najjednostavnije, krenimo od operacija *element po element*, to su operacije koje su u jednakom obliku definirane i za skalarne varijable, primjerice zbrajanje ili potenciranje. Jakobijan funkcije koja prima vektor $\vec{x}$ dimenzije n i vraća vektor $\vec{y}$ dimenzije m jest:

$$\frac{\partial \vec{y}}{\partial \vec{x}} = \begin{bmatrix} \nabla f_1(\vec{x}) \\ \nabla f_2(\vec{x}) \\ \dots \\ \nabla f_n(\vec{x}) \end{bmatrix} = \begin{bmatrix} \frac{\partial}{\partial \vec{x}} f_1(\vec{x}) \\ \frac{\partial}{\partial \vec{x}} f_2(\vec{x}) \\ \dots \\ \frac{\partial}{\partial \vec{x}} f_m(\vec{x}) \end{bmatrix} = \begin{bmatrix} \frac{\partial}{\partial x_1} f_1(\vec{x}) & \frac{\partial}{\partial x_2} f_1(\vec{x}) & \dots & \frac{\partial}{\partial x_n} f_1(\vec{x}) \\ \frac{\partial}{\partial x_1} f_2(\vec{x}) & \frac{\partial}{\partial x_2} f_2(\vec{x}) & \dots & \frac{\partial}{\partial x_n} f_2(\vec{x}) \\ \vdots & \vdots & & \vdots \\ \frac{\partial}{\partial x_1} f_m(\vec{x}) & \frac{\partial}{\partial x_2} f_m(\vec{x}) & \dots & \frac{\partial}{\partial x_n} f_m(\vec{x}) \end{bmatrix} \tag{3.1}$$

[11]

Ako radimo operacije *element po element*, onda će o rezultatu funkcije ovisiti samo vrijednosti na dijagonali, odnosno vrijedit će:

$$\frac{\partial \vec{y}}{\partial \vec{x}} = \begin{bmatrix} \frac{\partial}{\partial x_1} f_1(\vec{x}) & 0 & \dots & 0 \\ 0 & \frac{\partial}{\partial x_2} f_2(\vec{x}) & \dots & 0 \\ \vdots & \vdots & & \vdots \\ 0 & 0 & \dots & \frac{\partial}{\partial x_n} f_n(\vec{x}) \end{bmatrix} = diag(\frac{\partial}{\partial x_1} f_1(\vec{x}), \frac{\partial}{\partial x_2} f_2(\vec{x}), \dots, \frac{\partial}{\partial x_n} f_n(\vec{x})) \tag{3.2}$$

[11]

Ova jednakost će nam biti vrlo korisna zato što ćemo u rekurzivnom pozivu funkcije *derive* slati vrijednost umnoška vektora s Jakobijanom kao vrijednost parametra *seed*, a množenje s dijagonalnom matricom nam zapravo pojednostavi jednadžbe na način da možemo sve operacije raditi isto kao i sa skalarima. Potrebno je samo definirati osnovni razred *Matrix* te nadjačati potrebne operatore i kod s prethodnog poglavlja može normalno raditi.

Listing 3..2: Definicija osnovnog razreda za rad s matricama

```
class Matrix {
private:
  /**
   * @class opencl_data
   * @brief interna struktura koja sprema cl_mem referencu te se koristi
       isključivo kao std::shared_ptr da bi se automatski oslobodila memorija
       na uređaju
   */
  struct opencl_data {
    opencl_data(cl_mem data) : data{ data } { }
    opencl_data(const int N, const int M) {
      int _err;
      this->data = clCreateBuffer(globalContext, CL_MEM_READ_WRITE, N * M *
          sizeof(float), nullptr, &_err);
      checkError(_err);
    }
    ~opencl_data() { if(this->data != nullptr)
        checkError(clReleaseMemObject(this->data)); }
    cl_mem data;

    operator cl_mem () {
      return this->data;
    }
  };
  /**
   * @brief dimenzije matrice NxM
   */
  size_t N, M;
  /**
   * funkcija koja učita jezgru i postavi argumente za jednu od osnovnih
       operacija (zbrajanje, množenje, itd.)
   */
  inline cl_kernel loadKernel(const Matrix &other, const std::string &name)
      const;
```

```
public:
  /**
   * @brief konstruktor koji prima OpenCL memoriju i parametre N i M
   *
   * @param data cl_mem referenca na memoriju
   * @param N broj redaka u matrici
   * @param M broj stupaca u matrici
   */
  Matrix(cl_mem data, size_t N, size_t M);
  /**
   * @brief konstruktor koji prima ugniježdenu inicijalizacijsku listu s
       elementima tipa float
   *
   * @param mat inicjalizacijska lista
   * @throws std::invalid_argument ako dimenzije liste nemaju smisla, npr.
       {{3, 1, 2}, {1, -1}}
   */
  Matrix(std::vector<std::vector<float>> mat);
  /**
   * @brief "workaround" za korištenje skalara u svijetu matrica, stvara 1x1
       matricu
   *
   * @param x vrijednost skalara
   */
  Matrix(const float x);
  /**
   * @brief prazni konstruktor koji se koristi za neinicijalizirane matrice
   */
  Matrix();

  Matrix operator+(const Matrix &other) const;
  Matrix operator-(const Matrix &other) const;
  Matrix operator-() const;
```

```
    Matrix operator*(const Matrix &other) const;
    Matrix operator/(const Matrix &other) const;

    bool operator==(const Matrix &other) const;
    bool operator!=(const Matrix &other) const;

    /**
     * @brief pretvori matricu u std::string
     *
     * @throws std::invalid_argument ako je matrica veća od 100x100
     */
    std::string toString() const;

    /**
     * @brief getter za broj N
     *
     * @return vraća broj redaka matrice
     */
    size_t getN();
    /**
     * @brief getter za broj M
     *
     * @return vraća broj stupaca matrice
     */
    size_t getM();
    /**
     * @brief referenca na OpenCL podatke
     */
    std::shared_ptr<opencl_data> data;
};
```

3.2.2. Matrično množenje

Bitne su dvije jednakosti kod množenja vektora s matricom ($\vec{y} = \vec{x} \times W$). Prva je: $\frac{\partial \vec{y}}{\partial \vec{x}}$, a druga: $\frac{\partial \vec{y}}{\partial W}$. Za potrebe dubokog učenja trebamo izračunati i jednu i drugu unutar našeg *autograda* zato što nam prva služi za propagiranje greške unazad, a druga vrijednost služi za ispravljanje greške trenutnih težina.
Najbolje je da prvo proučimo izvod prve jednakosti:

$$\begin{aligned}
\frac{\partial \vec{y}}{\partial \vec{x}} &= \frac{\partial}{\partial \vec{x}} \left(\begin{bmatrix} x_1 & x_2 & \dots & x_n \end{bmatrix} \begin{bmatrix} w_{1,1} & w_{1,2} & \dots & w_{1,m} \\ w_{2,1} & w_{2,2} & \dots & w_{2,m} \\ \vdots & \vdots & \ddots & \vdots \\ w_{n,1} & w_{n,2} & \dots & w_{n,m} \end{bmatrix} \right) \\
&= \frac{\partial}{\partial \vec{x}} \left(\begin{bmatrix} \sum_{i=1}^{n} x_i w_{i,1} & \sum_{i=1}^{n} x_i w_{i,2} & \dots & \sum_{i=1}^{n} x_i w_{i,m} \end{bmatrix} \right) \\
&= \begin{bmatrix} w_{1,1} & w_{2,1} & \dots & w_{n,1} \\ w_{1,2} & w_{2,2} & \dots & w_{n,2} \\ \vdots & \vdots & \ddots & \vdots \\ w_{1,m} & w_{2,m} & \dots & w_{n,m} \end{bmatrix} = W^{\mathsf{T}}
\end{aligned}$$

Dakle, Jakobijan je jednak transponiranoj matrici težina. To pravilo ćemo iskoristiti u našoj implementaciji matričnog množenja, odnosno derivacije lijeve strane. Ukoliko se s lijeve strane nalazi matrica, a ne vektor, pravilo i dalje vrijedi jer ćemo kao rezultat imati matricu s N redaka, gdje je N broj redaka prve matrice, pa možemo pretpostaviti da je to veličina trenutnog skupa ulaznih podataka (eng. *minibatch*) te deriviramo sve na jednak način, jedino s više ulaznih vektora. Drugi Jakobijan glasi:

$$\frac{\partial \vec{y}}{\partial W} = \frac{\partial}{\partial W} \left(\begin{bmatrix} \sum_{i=1}^{n} x_i w_{i,1} & \sum_{i=1}^{n} x_i w_{i,2} & \dots & \sum_{i=1}^{n} x_i w_{i,m} \end{bmatrix} \right) = \begin{bmatrix} x_1 \\ x_2 \\ \vdots \\ x_n \end{bmatrix} = X^{\mathsf{T}}$$

Opet vrijedi da možemo samo transponirati drugu varijablu te pomnožiti je s vrijednošću koju smo dobili kao argument u rekurzivnom pozivu. Dakle, glavne formule koje

ćemo koristiti u matričnom načinu rada su:

$$\frac{\partial L}{\partial X} = \frac{\partial L}{\partial Y} W^{\top} \tag{3.3}$$

i

$$\frac{\partial L}{\partial W} = X^{\top} \frac{\partial L}{\partial Y} \tag{3.4}$$

[12]

3.3. Demonstracijski programi

3.3.1. Skalarne funkcije

U ovom kratkom demonstracijskom programu se definiraju varijable x, y i z, nakon čega se definira funkcija $f(x, y, z) = y + x + xyz$ te se ona derivira po varijabli x, zatim po varijabli y. Rezultat te parcijalne derivacije se prikaže grafički (graf funkcije) te se može i izračunati.

Listing 3..3: Demonstracijski program za skalarne funkcije

```
/**
 * @file
 * @brief demonstracijski program u kojem se vizualizira parcijalna
    derivacija zadane funkcije pomoću graphviza
 */

#include "autograd_core/visualize.hpp"
#include "autograd_core/basic_operations.hpp"
#include "autograd_core/autograd_util.hpp"
#include <memory>
#include <iostream>

int main() {
  using namespace autograd;

  auto x = createVariable(10.f, "x");
  auto y = createVariable(5.f, "y");
  auto z = createVariable(1.f, "z");

  auto expr = y + x + x * y * z;

  auto po_x_pa_y = expr->grad()["x"]->grad()["y"];
  // visualize(*po_x_pa_y, "out", true);

  auto expr2 =
```

```
    std::dynamic_pointer_cast<Expression<float>>(std::make_shared<Exp<float>>(x))
    * x + expr;
  visualize(*expr2->grad()["y"], "/tmp/out_image.png", true);
  std::cout << expr2->grad()["y"]->getValue() << std::endl;
  // visualize(*expr2->grad()["x"]->grad()["y"], "out", true);
}
```

3.3.2. Jednostavni klasifikator slika iz MNIST skupa

```
#include "IOptimizer.h"
#include "Module.h"
#include "autograd_core/expression.hpp"
#include "autograd_core/autograd_util.hpp"
#include "autograd_core/visualize.hpp"

#include "layers/Linear.h"
#include "layers/Sigmoid.h"

#include "optimizers/SGD.h"
#include "loss_functions/CrossEntropyLossWithSoftmax.h"

#include "data/IDataset.h"
#include "data/Dataloader.h"
#include "data/impl/MNIST.h"

#include "Util.h"
#include <cstdlib>
#include <memory>
#include <iostream>

class MnistClassifier : public Module {
private:
  std::unique_ptr<Linear> layers[3];
```

```
public:
  MnistClassifier() {
    this->layers[0] = std::make_unique<Linear>(28*28, 128, false);
    this->layers[1] = std::make_unique<Linear>(128, 64, false);
    this->layers[2] = std::make_unique<Linear>(64, 10, false);
  }

  virtual std::shared_ptr<autograd::Expression<Matrix>>
      forward(std::shared_ptr<autograd::Expression<Matrix>> ulaz) {
    for(int i=0;i < 3;i++) {
      ulaz = this->layers[i]->forward(ulaz);
      ulaz = std::make_shared<Sigmoid>(ulaz);
    }

    return ulaz;
  }
};

int main() {
  initCL_nvidia();

  std::unique_ptr<IDataset> trainset =
      std::make_unique<MNIST>("/home/jakov/faks/zavrsni/datasets/MNIST/raw/",
      true, false);
  std::unique_ptr<IDataset> testset =
      std::make_unique<MNIST>("/home/jakov/faks/zavrsni/datasets/MNIST/raw/",
      true, true);

  std::unique_ptr<Dataloader> train_loader =
      std::make_unique<Dataloader>(*trainset, 10);
  std::unique_ptr<Dataloader> test_loader =
      std::make_unique<Dataloader>(*testset, 10);
```

```
  std::unique_ptr<MnistClassifier> classifier =
      std::make_unique<MnistClassifier>();
  std::shared_ptr<IOptimizer> sgd_optim = std::make_shared<SGD>(0.01f);

  int counter = 0;
  while(train_loader->hasNext()) {
    auto [input, labels] = train_loader->nextBatch();
    auto logits = classifier->forward(autograd::createVariable(input, "X"));

    auto loss = std::make_shared<CrossEntropyLossWithSoftmax>(logits,
        autograd::createVariable(labels, "y", false));
    classifier->backward(loss, sgd_optim);

    if(counter % 1000 == 0) {
      std::cout << "loss: " << loss->getValue().toString() << std::endl;
    }

    counter++;
  }

  freeCL();
}
```

U ovom primjeru se zapravo vidi cijela funkcionalnost razvojnog okvira: korišteno je sučelje za učitavanje skupova podataka (*IDataset* i *Dataloader*) te je napravljen jednostavan razred koji nasljeđuje razred Module na način da nadjačavanjem metode *forward* definira unaprijedni prolaz kroz mrežu te prepušta okviru ostatak posla za unatražni prolaz (slično kao kod definiranja mreže u Pytorchu). Vidimo da se mreža sastoji od tri potpuno povezana sloja sa sigmoidalnim aktivacijskim funkcijama. Nakon što je to sve definirano, u glavnoj petlji jednostavno učitavamo podatke te radimo jednu epohu učenja mreže sa periodičkim ispisom vektora funkcije gubitka. Korišten je optimizator SGD (stohastički gradijentni spust) sa veličinom uzorka 10 te unakrsna entropija sa softmaxom kao funkcijom gubitka.

4. Rezultati i rasprava

4.1. Usporedba s pytorchem

U ovom primjeru je napisan isti klasifikator 3.3.2., jedino u Pytorchu. Vidimo iz koda da Pytorch naravno ima više funkcionalnosti, kao što su primjerice ugrađene funckije za prebacivanje u *one-hot* zapis i slično. No, ako pogledamo glavne definicije klase *MNIST-Classifier* iz ovog i prethodnog 3.3.2. primjera, vidimo odakle je došla inspiracija za ovaj projekt. Pytorch je jedino malo elegantniji zato što nije napisan u strogo tipiziranom jeziku, iako je sučelje koje on koristi za automatsko diferenciranje pisano u C++-u.

```
#!/bin/python3

import torch
from torch.nn import Module
from torch import nn
from torch.utils.data import DataLoader
from torchvision import transforms
from torchvision.datasets import MNIST

class MNISTClassifier(Module):
    def __init__(self):
        super(MNISTClassifier, self).__init__()
        self.fc1 = nn.Linear(28*28, 128, False)
        self.fc2 = nn.Linear(128, 64, False)
        self.fc3 = nn.Linear(64, 10, False)

    def forward(self, input):
        out = self.fc1(input)
```

```
        out = self.fc2(out)
        out = self.fc3(out)
        return out

    def get_loss(self, X : torch.Tensor, Y_ : torch.Tensor):
        loss = nn.BCEWithLogitsLoss()
        scores = self.forward(X)

        return loss(scores, nn.functional.one_hot(Y_, 10).float())

    def get_accuracy(self, X : torch.Tensor, Y_ : torch.Tensor):
        scores = self.forward(X)
        probs = nn.functional.softmax(scores, dim=1)
        Y = probs.argmax(dim=1)
        correct = (Y == Y_).sum().float()
        total = Y_.size(dim=0)
        return float(correct) / total

if __name__ == '__main__':
    transform = transforms.Compose([
        transforms.ToTensor(),
        transforms.Lambda(lambda x: torch.flatten(x))
    ])
    trainset = MNIST(root="/home/jakov/faks/zavrsni/datasets/", train=True,
        transform=transform)
    testset = MNIST(root="/home/jakov/faks/zavrsni/datasets/", train=False,
        transform=transform)

    classifier = MNISTClassifier()
    optimizer = torch.optim.SGD(classifier.parameters(), lr=0.01)

    for epoch in range(5):
        dataloader = DataLoader(trainset, batch_size=16)
        for i, (batch_data, batch_labels) in enumerate(dataloader):
```

```
        loss = classifier.get_loss(batch_data, batch_labels)
        loss.backward()
        optimizer.step()
        optimizer.zero_grad()
        if i % 100 == 0:
            print(f"epoch: {epoch}, iteration {i}: loss {loss}")

test_loader = DataLoader(testset, batch_size=1000, shuffle=True)
test_sample = next(test_loader.__iter__())
print(f"accuracy: {classifier.get_accuracy(test_sample[0],
    test_sample[1])}")
```

4.2. Moguće optimizacije

Postoje više mogućih optimizacija u odnosu na trenutno razvijeni sustav. Jedna od prvih je možda lokalizacija korištene memorije za derivaciju jednog izraza [13]. Mogli bismo napisati svoj alokator koji bi unaprijed rezervirao memoriju za derivaciju nekog izraza te koristiti taj komad memorije umjesto da imamo više uzastopnih poziva funkcije *make_shared* što dovodi do potencijalne fragmentacije gomile i usporavanja programa zbog nepotrebnih promašaja priručne memorije.

Druga potencijalna optimizacija jest dodavanje derivacija za često korištene funkcije. Ideja je slična kao kod naše implementacije funkcije sigmoide: derivacija se ne zadaje preko primitivnih operacija, već kao $f(x) * (1 - f(x))$, što u praksi jako pojednostavljuje graf te funkcije (odnosno njene derivacije) te nam omogućuje da ju izračunamo sa što manje računskih operacija.

Neke dodatne optimizacije bi bile otkrivanje "nebitnih" grana u grafu funkcije te uklanjanje istih ili otkrivanje istih grana tako da se izbjegne dvostruko računanje istih podizraza (primjer za to bi bio $f(x)$ u gornjem izrazu).

5. Zaključak

Postoje različiti pristupi u korištenju računala za rješavanje problema automatskog diferenciranja. Jedan od najefikasnijih i korisnicima najpristupačnijih načina je implementacija jednostavnog okvira temeljenog na unatražnom prolazu grafa (za potrebe strojnog učenja 2.1.2.) i nadjačavanju operatora 2.3.1. Ovakav pristup omogućava korisnicima jednostavno definiranje proizvoljnih arhitektura neuralnih mreža nadjačavanjem samo jedne metode, što značajno pojednostavljuje proces razvoja 3.3.2.

Dano programsko rješenje, osim što je efikasno, nudi i visoku fleksibilnost. Lako ga je moguće proširiti dodavanjem novih klasa, čime se osigurava da se sustav može prilagoditi različitim potrebama bez potrebe za promjenama u postojećem kodu. Ovakav princip programske organizacije, gdje se posebna pažnja posvećuje modularnosti i proširivosti, ključan je za uspješnost projekata u području automatskog diferenciranja i neuralnih mreža zato što se na taj način osigurava dugoročna održivost i skalabilnost programskog rješenja, što je od izuzetne važnosti u dinamičnom svijetu tehnologije i znanosti o podacima.

Literatura

[1] A. Dürrbaum, W. Klier, i H. Hahn, "Comparison of automatic and symbolic differentiation in mathematical modeling and computer simulation of rigid-body systems", *Multibody System Dynamics*, sv. 7, str. 331–355, 2002.

[2] A. Ramm i A. Smirnova, "On stable numerical differentiation", *Mathematics of computation*, sv. 70, br. 235, str. 1131–1153, 2001.

[3] Wikipedia, "Automatic differentiation — Wikipedia, the free encyclopedia", http://en.wikipedia.org/w/index.php?title=Automatic%20differentiation&oldid=1214839431, 2024., [mrežno; stranica posjećena: ožujak 2024.].

[4] W. github, http://wuciawe.github.io/math/2017/02/08/notes-on-differentiation.html, [mrežno; stranica posjećena: ožujak 2024.].

[5] M. Foco, M. Rietmann, V. Vassilev, M. Wong, D. Duka, i V. Grover, "P2072r0: Differentiable programming for c+", 2020.

[6] S. Rajamohan, https://srijithr.gitlab.io/post/autodiff/, [mrežno; stranica posjećena: ožujak 2024.].

[7] U. Naumann, "Optimal jacobian accumulation is np-complete", *Mathematical Programming*, sv. 112, str. 427–441, 2008.

[8] P. H. Hoffmann, "A hitchhiker's guide to automatic differentiation", *Numerical Algorithms*, sv. 72, br. 3, str. 775–811, 2016.

[9] "Automatic differentiation package - torch.autograd - pytorch 2.2 documentation", https://pytorch.org/docs/stable/autograd.html#, [mrežno; stranica posjećena: travanj 2024.].

[10] “Opencl”, https://www.khronos.org/opencl/.

[11] T. P. . J. Howard, “The matrix calculus you need for deep learning”, https://explained.ai/matrix-calculus, [mrežno; stranica posjećena: svibanj 2024.].

[12] D. F. . J. Johnson, “Backpropagation for a linear layer”.

[13] C. C. Margossian, “A review of automatic differentiation and its efficient implementation”, *Wiley interdisciplinary reviews: data mining and knowledge discovery*, sv. 9, br. 4, str. e1305, 2019.

Sažetak

Demonstracijski okvir za diferencijabilno programiranje

Jakov Novak

Kako neuralne mreže postaju sve složenije i popularnije, ključno je razumjeti mehanizme koji omogućuju njihovo efikasno učenje. U ovom radu istražujemo matematičke osnove automatskog diferenciranja, što je temeljna tehnika za treniranje neuralnih mreža. Detaljno razmatramo teorijske koncepte koji stoje iza automatskog diferenciranja, uključujući derivacije i gradijente, te njihovu primjenu u optimizaciji modela strojnog učenja. Osim teorijskog pregleda, rad obuhvaća i praktičnu implementaciju jednostavnog sustava za automatsko diferenciranje. Pomoću ovog sustava demonstriramo njegovu primjenu kroz nekoliko programskih primjera. Na kraju, razvili smo klasifikator slika koristeći vlastiti sustav za automatsko diferenciranje, čime smo pokazali praktičnu korisnost i efikasnost razvijenog rješenja u stvarnim zadacima strojnog učenja.

Ključne riječi: autodiff; c++; automatsko diferenciranje; strojno učenje; duboko učenje

Abstract

Demonstrative framework for differenciable programming

Jakov Novak

As neural networks become increasingly complex and popular, understanding the mechanisms that enable their efficient learning is crucial. This paper explores the mathematical foundations of automatic differentiation, a fundamental technique for training neural networks. We thoroughly examine the theoretical concepts behind automatic differentiation, including derivatives and gradients, and their application in optimizing machine learning models. In addition to the theoretical overview, the paper includes a practical implementation of a simple automatic differentiation system. We demonstrate the system's application through several programming examples. Finally, we developed an image classifier using our custom automatic differentiation system, showcasing the practical utility and efficiency of the developed solution in real-world machine learning tasks.

Keywords: autodiff; c++; automatic differentiation; machine learning; deep learning