Synopse

Aksel Mannstaedt

6. maj 2023

Indhold

`https://github.com/unic0rn9k/autodiff`

Resumé

I projekt forløbet, har jeg arbejdet med at udvikle en crate (Rust programmeringssprogets svar på biblioteker) til automatisk differentiering, beregnet til machine learning. Grundet applikationen af craten, har det været relevant at bygge support for differentiering af algebraiske operationer for skalare, samt matricer.

Under projekt forløbet har det været relevant at eksperimentere med forskellige måder at repræsentere grafen på og derudover at overveje mere generale måder at repræsentere data, samt algebraiske operationer på. Til dette formål, er en række forskellige relevante faglige metoder blevet anvendt.

Til sidst har jeg testet, og evalueret projektet i forholdet til en række forskellige metrikker og benchmarks.

1 Problemformulering

Hvordan kan man effektiv finde derivativer af arbitrære funktioner? Hvordan kan dette gøres med algebraiske objekter, så som vektore, matricer og tensorere? Hvordan kan et sådan system anvendes til at løse andre problemer, så som OCR?

2 Programbeskrivelse

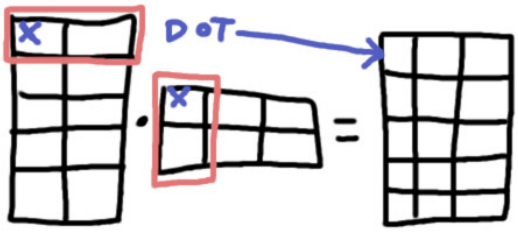
Mattematik og programmering handler begge, om abstrakt problem løsning. Mattematik bliver derfor også anvendt i mange forskellige områder af programmering. Her er linear algebra og differentiering ofte brugt, specielt også i kombination med hinanden. Linear algebra er for eksempel essentielt i spil udvikling, hvor det blandt andet bruges til fysik og grafik. Begge dele bliver også brugt i machine learning, som projektet har haft fokus på.

Linear algebra, er en gren af mattematik, der dækker en lang række under emner. Typisk i programmering, og i dette projekt, vil der dog kun være fokus på nogle af de matematiske objekter i linear algebra, navnlig matricer og vektore. En n-dimensionel vektor, kan faktisk bare fortolkes som en $nx1$ matrix. En matrix er et rektangulært array af skalare. I praksis kan det implementeres som et array af arrays af skalare. Indeksering i det yderste array, ville indeksere matricens kolonner, ligeledes ville det inderste array indeksere i en række.

Der eksistere en nogen forskellige operationer, der er defineret for matriser. Dimensionerne af en matrise noteres rækker × kolonner. De to af de mest centrale operationer for matricer er transposes og matrix mulitplikation. Et transpose af en matrice, er når man bytter rundt på rækker og koloner, altså vil en $m \times n$ matrix blive til en $n \times m$, hvor elementet $(0, 1)$ vil ende på $(1, 0)$ plads.
Matrix multiplikation (forkortet til matmul). Matrix produktet er ikke kommunikativt, altså for matriser er $a \cdot b$ ikke det samme som $b \cdot a$.

$$a \cdot b = c$$

Her er den første kolonne i c, dot produktet mellem den første hver række i a, og den første kolonne i b. Dette gentages for alle rækker i a, vor hver iteration resultere i en ny række i c. Derfor, hvis a har dimensionerne $m \times n$, skal b have



Figur 1: Visuelizering af matrix multiplikation

Hvad er differentiering? Differentialet af en funktion, er i sig selv en funktion, der beskriver **hældningen** funktionen, på et givent punkt, med henhold til en uafhængig variabel. Her er hældningen altså den effekt en ændring i en uafhængig variable (også kaldet en parameter, i machine learning), vil have på en afhængig variabel (outputtet af funktionen).

For eksempel vil hældningen, det partielle derivativ $\delta y/\delta x$, af $y = 2x$ være $2 + 0x$, der en stigning i x på 1, vil resultere i at y stiger med 2. Det samme princip gælder for ikke-lineære funktioner, her vil derivativet dog være hældningen af tangenten på det punkt.

Et partielt derivativ er et derivativ, med henhold til en given variabel, hvor alle andre variabler bliver betragtet som konstante.

Helt specifikt er den matematiske defination af et derivativ, for en funktion, $f(x)$

$$f'(x) = \lim_{h \to 0} \frac{f(x+h) - f(x)}{h}$$

For vektor funktioner, vil sådan en hældning selv være repræsentere som en vektor. Ligeledes har vektor funktioner, differentieret med henhold til en vektor parameter, matricer som hældning. En sådan hældnings matrise, heder en gradient eller en Jacobian og bliver typisk noteret med »∇ parameter symbol«. En Jacobian bliver generalt brugt til at beskrive alle partielle derivativer for en vektor funktion, af flere uafhængige variabler, hvilket også kan beskrive en vektor funktion af en vektor.

$$\vec{v} = [x, y]$$

$$\vec{f}(v) = [f_1(v), f_2(v)]$$

$$\nabla f = \begin{bmatrix} \delta f_1/\delta x & \delta f_1/\delta y \\ \delta f_2/\delta x & \delta f_2/\delta y \end{bmatrix}$$

Differential regning bruges til mange forskellige ting, så som numerisk ligningslæsning, machine learning og computer-graphics.
Man kan matematisk differentiere alle funktioner, ved at anvende differentierings regler systematisk. Derfor kan man også lave algoritmer, der kan differentiere funktions udtryk, hvilket er utroligt brugbart i situationer hvor man skal bruge derivativet af en funktioner dynamisk, eller bruge derivativet af mange forskellige funktioner, hvor det ville være besværligt at udregne i hånden og hvis man skal veligeholde funktioner, der kan blive opdateret i fremtiden, så deres derivativ forbliver korrekt. Denne teknik kaldes for autodifferentiering[3] (forkortes til AD), og til dette formål bruger man typisk det der hedder en autodifferentieringsgraf.

I machine learning laver man »modeler« af data. Her er en model, en algoritme der kan beskrive noget trænings data kontinuert. Altså man bruger modelen til at kunne mappe fra alle mulige inputs til et output, ud fra »eksempler« (trænings dataen). Trænings dataen er ikke altid forud indsamlet data, men kan også i nogen generative modeler, være genereret mens modelen træner. Man siger derfor at machine learning modeler kan lære fra data.

Deep learning er en græn af machine learning, hvor autodifferentiering bliver anvendt meget. Her opstilles funktioner af nogle inputs, og nogle parametre. Herefter bruges en cost funktion, til at finde hvor stor en afvigelse der er mellem trænings dataen og modelens prediction.
Ud fra cost funktionen, bruger man så $\delta cost/\delta parameter$ til at minimere costen, for et givent data punkt. Dette gøres simpelt ved at trække $\delta cost/\delta parameter$ fra parametren, der hældningen altid vil være af same type algebraisk objekt, som parametren. Fx vil en matrix parameters hældning på index $[0,0]$, beskrive hældningen af dens tilhørende parameter på samme index.

Altså ville vi forvente, at hvis vi differentierede en machine learning model for en specifik parameter, fx $(\delta cost/\delta parameter)[0,0]$, så ville give en float, der ville beskrive ændringen i vores cost, hvis vi øgede den givene parameter med 1.

En af de mest hyppige typer af modeler (som typisk bliver brugt som en lille del af støre modeller), er fully-connected nets (også kaldet et dense layer). Et dense layer, tager en vektor som input, og har en vægt matrix, samt en bias vektor, som parametre. I et dense layer, udregnes dens output, ved at tager matrix produktet mellem dens input, og vægt matricen, hvor efter man lægger biaset til. Dette bliver derefter kørt igennem en aktiverings funktion, som introducere en non-linearitet (det hjælper med at få modelen til at tilpasse data). En populær aktiverings funktion, til klassificerings opgaver, er softmax, der den kan konvertere en arbitrer vektor, til en normal fordeling, hvor hver element i vektoren, vil svares til en sandsynligheds værdi, for at input dataen er af en given klasse.

2.1 Typer auto differentiering

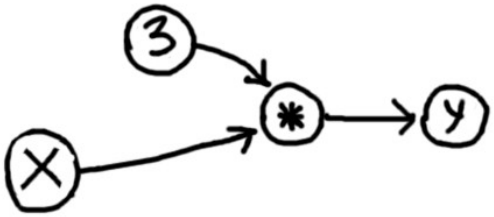
Man kan opdele de forskellige typer autodifferentierings tekniker i to kategorier, numerisk og symbolsk. Numerisk autodifferentiering er utroligt nemt at implementere, men kræver typisk mere processorkraft, end en symbolsk løsning, og giver i øvrigt kun approximater til et specifikt tangenter.

Symbolsk autodifferentiering kan derimod anvendes, til at finde generale løsninger til derivativer, som derefter kan bruges til udregne hældninger på specifikke punkter, på en effektiv måde.

Til symbolsk differentiering, repræsentere man funktions udtryk, som graf strukture, hvor efter man programmatisk kan anvende differentierings regler til at udregne afledte funktioner, repræsenteret som graf struktur.

En graf struktur, er en datastruktur der beskriver hvordan 'nodes' er forbundet med 'edges'. Graf strukture kan bruges til at repræsentere en bred række af data. Git repræsentere for eksempel versioner af et repository som en graf. I autodifferentiering bruger man typisk en graf struktur til at repræsentere matematiske udtryk, der kan evalueres til en skalar værdi. Mattematiske udtryk kan repræsenteres som DAGs (Directed Asyclic Graphs), hvilket vil siges, edges i grafen har en 'retning', der de evalueringen af de yderste led, afhænger af de inderste.

Inden for symbolsk autodifferentiering findes der to teknikker, forward og reversen mode, som hver har deres fordele og ulemper. Uanset hvilken teknik man bruger, kan man optimere graf strukturen efterfølgende, men typisk vil forward mode, resultere i grafer der har færrest operationer, når der er flere outputs end inputs, og omvendt med reverse mode.



Figur 2: En visualisering for grafen af y = x * 3

2.2 Kravspecifikationer

I machine learning beskæftiger man sig typisk med funktioner af mange uafhængige variable, og en enkelt afhængig variabel (cost funktionen). Derfor vil målet med projektet være at udvikle en reverse mode autodifferentierings graf, der tillader at kunne differentiere kode med henhold til udvalgte arbitrære parametre, der man ikke altid vil have brug for at bruge processorkraft på at differentiere med henhold til alle parametre.

Før starten på projektets udviklingsforløb, har jeg opstilt en række minimumskrav, og nogle »ekstra« krav. Minimums kravene, placeret lige over ekstra kravene, er de mindste nødvendige krav, for at det ville være muligt at lave en demo mnist classifier. Ekstra kravene er nogle krav jeg tænkte kunne være fede at have opfyldt, men som ikke var strengt nødvendige for projekteres success.

Kravspecifikation	**Opfyldt**
Graf struktur der kan repræsentere algebraiske operationer.	Ja
Simpel AD af funktioner med 1 input og output.	Ja
AD for funktioner med flere inputs og outputs.	Ja
Kan differentiere med henhold til -arbitrere parametre i et udtryk.	Ja
Support for algebraiske objekter som vektore og matricer.	Ja
AD af operationer med algebraiske objekter.	Ja

Ekstra	**Opfyldt**
Compiletime evaluering af derivativer	Ja
Multithreading	Nej
GPU support	Nej

3 Metode og tooling

Til udviklingen af projektet, har jeg valgt at bruge Rust programmerings sproget, af grunde der bliver gennemgået senere. Her har jeg anvendt Rusts test suite, build system, samt nalgebra (`https://lib.rs/nalgebra`), som er en crate til tensor mattematik, med support for matricer og vektore.

Under udviklingsprocessen har jeg anvendt unit tests, samt en lidt løsere version af TDD (Test Driven Development). I TDD opstilles unit tests til kode, før selv implementeringen begyndes. Dette har jeg ikke helt gjort, der jeg ikke ville låse mig fast i en API på forhånd. I stedet har jeg valgt nogen funktions udtryk på forhånd, som ville kunne differentieres med minimumskravene opfyldt. Som test funktion har jeg primært brugt softmax funktionen, som er en aktiverings-funktion i deep learning. Softmax funktionen ville sådan kunne anvendes, til at lave et demo neural net, til at klassificere mnist data sættet.

Til dette formål, har jeg anvendt Rusts indbyggede test suite, som kommer med deres package manager/build system cargo. Her kan man simpelt annotere unit tests med '#[test]' (man kan ligeledes bruge '#[bench]' til benchmarks) macroen, og derefter køre 'cargo t', eller 'cargo test' i sin terminal, for at eksekvere unit testene.

Figur 3: 'cargo test' der viser alle passing unit tests

En af værktøjerne der blev anvendt i udviklings/test processen, er sagemath [4]. Sagemath er et opensource CAS program, som har en del ligheder til maple. Sagemath understøtter symbolsk differentiering, hvilket har været rigtig brugbart, til at tjekke om koden virker, uden kun at sammenligne numeriske resultater.

```
SageMath version 9.7, Release Date: 2022-09-19
Using Python 3.10.8. Type "help()" for help.

sage: x,y = var('x,y')
sage: derivative(exp(y)/(exp(y)+exp(x)),y)
e^y/(e^x + e^y) - e^(2*y)/(e^x + e^y)^2
sage: derivative(exp(y)/(exp(y)+exp(x)),y)(x=1,y=2).n()
0.196611933241482
```

Figur 4: eksempel på differentiering med sagemath

Udover TDD, har jeg anvendt rapid-prototyping. I denne metode, udvikles et MVP (Minimum Working Product) hurtigst muligt, uden at tænkte på performance af koden. Herefter raffineres produktet, indtil et tilfredsstillende resultat er opnået.

Jeg har brugt github til version control, der git tillader at lave nye branches at teste kode på, samt at kunne pulle gamle commits, til at lave yderligere benchmarking og testing, til synopsen.

4 Funktionalitet

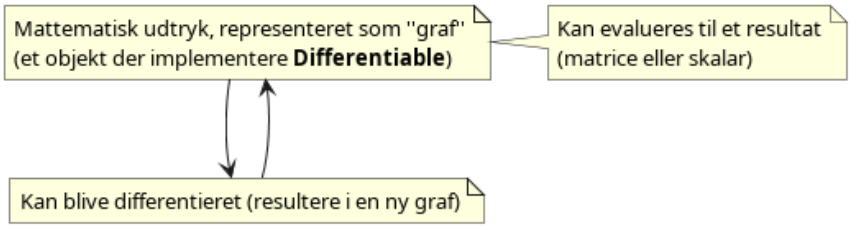
Autodiff grafen, kan bygges ved hjælp af en normal mattematisk syntax, der mattematik operatorene fra 'std::ops' er implementeret. Herefter kan man kalde '.eval()' på det resulterende graf objekt, for at få et numerisk resultat. Desuden kan man også kalde '.derivative(x, grad)' for at få grafen til derivativet med henhold til 'x'.

Derivativer for Plus, Minus, Gange, Dividere, samt Sum (af en matrise eller vektor) og Exp (eksponenten/e^x) kan alle differentieres, samt funktioner opbygget af disse operationer.

Dette betyder at softmax funktionen, som eksempel, kan implementeres ved: 'Exp(x)/Sum(Exp(x))'.

5 API Design (Brugergrænseflade)

Udover at have brugt sagemath til at tjekke resultater, har jeg også været inspireret af sagemaths API. Deres elegante løsning, er simplet hen bare at man kan kalde '.derivative([x])' på et funktions udtryk, for at differentiere med henhold til 'x'.



Dog adskiller min implementeringen sig lidt fra sagemaths, der derivative funktionen også tager et argument til en gradient af den ydre funktion, der metoden er implementeret recursivt. Her kan man dog altid give 1, eller en matrise af 1 taller, som argument, hvis der ikke er en endu en ydre funktion.

```rust
#[test]
fn basic() {
    use crate::prelude::*;
    let x = 2f32.symbol("x");
    let y = 3f32.symbol("y");
    let f = (&x * &y + &x * &x) / (&y + &x);

    let [dx, dy] = f.derivative(["x", "y"], 1f32);

    assert_eq!(dy.eval().0, 0.);
    assert!((dx.eval().0 - 1.).abs() < 1e-6, "{} ≠ 1", dx.eval().0);
}
```

Figur 5: Koden til 'basic' unit testen set på figur 1.

6 Pseudokode

Logikken til differentierings processen, er udlet ved at omskrive de mattematiske regler for differentiering, til funktioner der beskriver processen **programmatisk**.
Til dette formål, har det været relevant at lave pseudokode, der beskriver hvordan man ville differentiere de forskellige regler i hånden, **programatisk**.

Her bruges en ny syntax til at beskrive reglerne. f' er bare det partielle derivativ af f med henhold til en arbitrer parameter. Sådan bliver additions reglen (reglen der anvendes til at differentiere en additions node) til $f + g \to f' + g'$. Ligeledes er produkt reglen $f \cdot g \to f \cdot g' + g \cdot f'$.
Forward mode differentiering, er simplest at forstå, ud fra differentierings reglerne, der de er omskrevet rimeligt direkte.

```
y = 2x+3 = Add(Mul(2,x), 3)


# wrt er kort for with respect to (med henhold til)
# self.lhs er venstrhånds siden af det givene undtryk, mens self.rhs er højre.
impl Add:
    fn derivative(self, wrt):
        self.lhs.derivative(wrt) + self.rhs.derivative(wrt)


impl Mul:
    fn derivativ(self, wrt):
        self.lhs.derivativ(wrt) * self.rhs + self.rhs.derivative(wrt) * self.lhs
```

pseudokode over, vil resultere i rekursive kald, til 'derivative', hvilket altså betyder, at de inderste udtryk, vil blive differentieret først.
I reverse mode differentiere man de yderste led først, og giver det partielle derivativ af den ydre funktion (ofte refereret til som 'tape') videre i hvert kald. Sådan kan man bruge kæde reglen, $f(g) \rightarrow f'(g) \cdot g'$, til at regne det endelige derivativ ud.

Altså ville addition og multiplikation, i reverse mode, blive til **følgene pseudokode:**

```
impl Add:
    fn derivative(self, wrt, d):
        self.lhs.derivative(wrt, d) + self.rhs.derivative(wrt, d)

impl Mul:
    fn derivativ(self, wrt, d):
        self.lhs.derivativ(wrt, d * self.rhs) + self.rhs.derivative(wrt, d * self.lhs)
```

Det samme princip blev anvendt til at udlede alle de andre operationer. De originale differentierings reglerne er mattematisk beviste, og relativt simple at forstå, dog ville det være for meget at gennemgå i denne synopsis.

```
impl MatMul:
    fn derivativ(self, wrt, d):
        self.lhs.derivativ(wrt, d * self.rhs.transpose()) + self.rhs.derivative(wrt, self.lhs.transpose() * d)
        # gange operationerne over, er matmul operationer.
```

6.1 Generics og traits

I pseudo koden over, er indeholder 'Add' strukturen (en struktur er basically det samme som en klasse), et led der har typen 'Mul' og et led med typen 'Int'. Her kan felterne i operations strukturene altså have flere forskellige typer. I programmeringssprog med statiske typer, som Rust, bliver dette kaldt polymorphism.
Plymorphism i Rust, er bygget op omkring et koncept de kalder for 'trait'. En trait er en meta struktur, der beskriver en rækker metoder, der skal defineres for funktionen, for den KAN implementere traiten.

Rust har 2 typer polymorphism: Generics og dynamic dispatch. Dynamic dispatch er nemmest at anvendte, der det bare kræver at man definere hvilken trait en given type skal implementere. Dynamic dispatch fungere ved at anvende specielle pointere, der både indeholder en adresse til dataen objektet indeholder, men så også en ekstra addresse til et »vtable«, som bare er et array af funktion pointers, til alle metoder i traiten. Derfor kommer dynamic dispatch altså med lidt ekstra overhead, der det kræver at holde styr på mere (samt mere fragmenteret) data, på runtime.

Derfor har jeg valgt at anvende generics, som er kendetegnet ved at de er inkapsuleret mellem '< >'. En generic kan have en hvilket som helst type, der implementere alle dens constraints, men dens typer bliver avilket på compiletime.
For eksempel kan funktionen add3, defineres for en hvilketsomhelst type der implementere add operationen, med følgende kode. 'fn add3<T: Add>(a: T, b: T, c: T) -> T { a + b + c }'. Her er T en genric, og derfor vil add3 virke med både u8, f32 og alle andre typer der implementere 'std::ops::Add'.

6.2 Implementering & Kode

Implementationen af autodifferentieringsgrafen, er skrevet i Rust [5]. Dette er af to grunde, 1. Rust's memory safety mode, der tillader at skrive high-performance kode, uden at risikere at introducere undefined behaviour [1] i sin kode og 2. Rust's generiske type system, specifikt fordi det under støtter traits, generics og GAT's (Generic Associated Types [2]).

Der er to koncepter fra disse features, der bliver anvendt i koden:

1. **Lifetimes** /levetid bliver anvendt til at indikere til rust compileren, hvor »lang tid« et stykke data lever, så den kan garantere programets memory safety. Her er en levetid notere med ' symbolet (i modstæning til ", brugt til chars).

   Her er det borrow checkeren, der garaneteret at alt data attribueret med en givene leve tid, ikke bliver deallokeret før der ikke er flere references (en borrow checked pointer) i brug, med den leve tid.

I koden over repræsentere levetiden, 'a, den tid i program eksekveringen, hvor alt data der anvendes i differentieringen af Self, eksistere, og derfor også den tidsramme, hvor Self kan differentieres. Derfor er Self parametren, i Δ typen, også begrænset, til 'a levetiden. Dette betyder altså, at typen af derivativet af Self, kun er en gyldig type, når Self lever, imens Self

```rust
pub trait Differentiable<'a> {
    type Δ<D>
    where
        Self: 'a;
    type T;

    fn eval(&self) -> Self::T;

    fn derivative<const LEN: usize, D: Clone>(&'a self, k: [&str; LEN], d: D)
        -> [Self::Δ<D>; LEN];
```

Figur 6: Differentiable trait defination

kan differentieres. Med andre ord, garantere det, at det er en ugyldig operation (giver en error på compiletime), at differentiere data, der er blevet deallokeret.

1. **GATs**

```rust
pub trait Differentiable<'a> {
    type Δ<D>
    where
        Self: 'a;
    type T;

    fn eval(&self) -> Self::T;

    fn derivative<const LEN: usize, D: Clone>(&'a self, k: [&str; LEN], d: D)
        -> [Self::Δ<D>; LEN];
```

Punkterne markeret med 1. og 2. er associerede typer. Det er typer der skal devineres når traiten implementeres for en type. 1. er en GAT, der den sig selv afhænger af en type D. Her er D typen af gradienten af den ydre funktion, differenteret.

7 Test

Initielt (givet rapid prototyping metoden) blev grafen implementeret med Arcs i stedet for references. Arc (Atomic Reference Counting) er et alternativ til borrow checking, der garantere memory safety på runtime, ved, som navnet antyder, at øge en atomisk (en type integer der tillader thread safety) counter, hver gang Arcens data bliver referenced. Herefter nedsættes tælleren hver gang en reference bliver dropped (ikke bliver brugt mere). Først når tælleren rammer nul, vil dataen så blive deallokeret.

Bemærk at benchmarksne ikke inkludere differentierings tiden, men kun tide det tager at evaluere et udtryk. Dog ville compileren sandsynligt kunne optimere differentierings tiden helt væk, grundet brugen af GATs

```
autodiff  HEAD (712a7b8) ()  v0.1.0
 |> cargo bench
   Compiling autodiff v0.1.0 (/home/unic0rn9k/Documents/autodiff)
    Finished bench [optimized + debuginfo] target(s) in 0.64s
     Running unittests src/lib.rs (target/release/deps/autodiff-16db16c9ae9d16b4)

running 1 test
test basic ... bench:         267 ns/iter (+/- 215)

test result: ok. 0 passed; 0 failed; 0 ignored; 1 measured; 0 filtered out; finished in 3.49s
```

Figur 7: Arc AD tager ca 267na at eksekvere

```
98  #[bench]
99  fn basic(bencher: &mut Bencher) {
100     bencher.iter(||{
101         let x = black_box(Arc::new(2f32));
102         let y = black_box(Arc::new(3f32));
103         let f = black_box(Arc::new(Add(
104             Arc::new(Mul(x.clone(), y.clone())),
105             Arc::new(Mul(x.clone(), x.clone())),
106         )));
107         f.eval();
108     })
109 }
```

Figur 8: Arc AD kode

```
running 5 tests
test basic ... ignored
test mat::gradient_decent ... ignored
test value::ord ... ignored
test test::basic        ... bench:           1 ns/iter (+/- 0)
test test::hand_written ... bench:           1 ns/iter (+/- 0)

test result: ok. 0 passed; 0 failed; 3 ignored; 2 measured; 0 filtered out; finished in 11.41s
```

Figur 9: reference AD tager ca lige så lang tid som en håndskrevet version at eksekvere.

```
7   #[bench]
8   fn basic(b: &mut Bencher) {
9       let x = black_box(2f32.symbol("x"));
10      let y = black_box(3f32.symbol("y"));
11      b.iter(|| {
12          black_box(&x * &y + &x * &x).eval();
13      });
14  }
```

Figur 10: reference AD kode

7.1 Mnist demo

`https://github.com/unic0rn9k/autodiff/blob/softmax/examples/mnist/src/main.rs` **19999 accuracy: 70.00% på ca 8min compiletime og ca 2min runtime**
mnist craten (`https://lib.rs/mnist`) blev brugt til automatisk at downloade, og dele mnist data settet op i trænings data og validerings data. Data settet bliver opdelt sådan, for at sikre at modelen ikke bare har memoriseret trænings dataen. mnist classifieren convergerede (stoppede med at blive bedre eller være) ved en accuracy af omkring 70% på validerings dataen. Det er altså ikke en super god accuracy, men det er heller ikke pointen. Classifieren tilpassede 20000 datapunkter på 2 minutter! Dog tog det 8 minutter at compile.

8 Konklusion

Alle kravene i problemformuleringen er opfyldt, dog med et par hængepartier.

Jeg ville gerne have haft tid til at lave GPU support til de algebraiske operationer. Dette ville have været muligt at implementere relativt nemt, ved brug af BLAS (Basic Linear Algebra System), som er en API til effektive og generale linear algebra subroutiner. Her er ideen at specialiserede implementationer af BLAS specifikationerne specifikt til diverse devices, på en optimal måde. Dette ville have været muligt at nå, hvis jeg ikke havde valgt at bruge nalgebra til disse operationer, der det ville have taget for lang tid at refaktorere.

Yderligere blev kompilerings tiderne absurdt lange, ved differentiering af mere komplicerede udtryk, grundet brugen af GATs. I mnist klassifier koden, bliver autodiff craten ikke brugt til at differentiere det yderste lag af cost funktionen, der det begyndte at tage flere timer at compile koden. Det er ikke kun et problem med koden, men også med Rust. Lange compiletimes er et kendt drawback i Rusts communities.

9 Logbog

10/3 lavet forward mode ad for floats og basic graf
22/3 implementeret std::ops for nodes

22/3 ryddet op i project structure, og fixet bug med derivativer blev prematurely evaluated.
24/3 generalisrede implementationer af traits og operationer
27/3 Beygndte på differentiering for matricer
28/3 backwards mode, der ikke virker
mistede resten af logbogen. Der er mere information i commit historien på git repositoriert

Litteratur

[1] The Rust Foundation. Behavior considered undefined. `https://doc.rust-lang.org/reference/behavior-considered-undefined.html`.

[2] The Rust Foundation. Generic associated types initiative. `https://rust-lang.github.io/generic-associated-types-initiative/index.html`.

[3] Richard D. Neidinger. Introduction to automatic differentiation and matlab object-oriented programming. `http://academics.davidson.edu/math/neidinger/SIAMRev74362.pdf`, 2010.

[4] William A. Stein. System for algebra and geometry experimentation. `https://www.sagemath.org/`, 2005.

[5] the Rust Team. Rust. `https://rustlang.org`.