StuCrs

StuCrs

著:臼井千裕　鈴木翔天

作成日2025//

StuCrs

DEEPLEARNING FOR RUST

はじめに

近年、人工知能は私たちの生活に溶け込み、様々な利便性をもたらしてきました。対話型のAIなど、今ではAIは仕事やプライベートといった際に欠かせない存在となってきています。そんなAIにたよる日常生活が当たり前になりつつあるこの時代において、AIの信憑性や、人の学習能力の低下、情報の操作、そして情報漏洩といったAIの危険性が重視され始めています。こういった危険性が顕在化している背景に、AIの急速な大衆化があると考えます。AIの研究者とは違い、未知のものである一般の人々がインターネットやメディアにより、ここ数年であっという間にAIというものが身近なものとなり、簡単に利用できるようになりました。しかし、人々は突然現れた、日常で欠かせない得体の知れないAIというものを「ただ便利だから」、「みんなが使っているから」、ということで十分理解せずに利用しようとしているのではないでしょうか。人々の日常生活に溶け込む人工知能はもはや、現代社会において教養として学ぶ必要があります。なんとなく文を送ったら、返事が来たというような、裏側を見ない表面的なAIの理解だけでの使用は、浅はかで、無責任なAIの使用と言えるのではないでしょうか。AIはどのような仕組みで構成されているのか、裏側でどのような処理がされているのか、そういったAIの裏側を学ぶことで、私たちは真のAIとの共存が可能になるでしょう。このドキュメントでは、現代のAIの基礎である深層学習の仕組みを、新しいプログラミング言語Rustで一から実装するプロセスを通じて、AIの裏側を学びます。高校生である筆者らにとって、この開発は大胆な挑戦であり、多少乱暴で荒々しい冒険となりますが、ぜひ最後までお付き合いください。

フレームワーク**StuCrs**のコンセプト

現在、深層学習のフレームワークはTensorFlowやPytorchというように複数存在します。それらのフレームワークには様々な特徴、コンセプトがありますが、そんな中で私たちはーからRustで深層学習のフレームワーク:**StuCrs**を開発、実装しました。

TensorFlowやPyTorchといったフレームワークはオープンソースであり、公開されています。試しにGitHubでTensorFlowの中のコードを見てみましょう。するとどうでしょうか、ファイルやフォルダーがたくさん存在し非常に複雑に構成されています。また、ファイルの中のコードもパフォーマンスを最適化するために、様々な関数が組み合わさって、読み手にとって解読しづらいものとなっています。既存の多くのフレームワークは様々なプロジェクトでフレームワークとして利用されるためにパフォーマンスや機能の豊富さ、バグの少なさなどが重視されます。そこで私たちは逆の立場として、シンプルな構造のフレームワークの構築を目指し、それを読者が理解して1からフレームワークを実装してもらうことで深層学習の原理を探究してもらう、そんな目標をもってこのような研究に至りました。

環境

環境の詳細はGitHubのREADMEをご覧ください。

1: 実数での自動微分

1: 変数の作成

変数とはデータを格納する箱のようなものです。はじめにこの変数をVariableという名前で構造体として実装してみましょう。

```
fn main() {
    let mut a = Variable::init(1.0);
    println!("{}", a.data);
    a.data = 5.0;
    println!("{}", a.data);
}

struct Variable {
    data: f32,
}

impl Variable {
    fn new(data: f32) -> Self {
        Variable { data }
    }
}
```

まずvariableという構造体を実装します。フィールドとして小数をあつかえるf32型をdataとして保持します。また、コンストラクタを生成するための初期化の関数**new()**を定義します。これにより、f32のデータを渡すことでデータを保持する変数、**Variable**型を生成することができます。main関数を実行すると、**Variable**のデータを見ることができます。

2: 関数の作成

先ほど変数を実装しましたが、変数はどのように作り出されるのでしょうか。それは関数です。変数を入力として関数に渡し、出力として新たな変数を生み出します。それでは関数を実装していきましょう。

2.1: Functionトレイトの実装

はじめにRust独自の仕様であるトレイトを理解する必要があります。トレイトとは様々な構造体がある特定のふるまいをすることを保証するものです。ここでいうふるまいとは、トレイトを継承した構造体がすべて同じある関数を保持しているということです。

今後関数は様々な構造体で使用するのでトレイトを使用して関数を定義します。では実際に関数をトレイトで実装してみましょう

```
fn main() {
  let x = Variable::init(2.0);
     println!("{}", x.data);
     let f = Square {};
     let y = f.call(&x);
     println!("{}", y.data);
     println!("{}", x.data);
 }
trait Function {
    fn call(&self, input: &Variable) -> Variable {
        let x = input.data;
        let y = self.forward(x);
        let output = Variable::new(y);
        output
    }

    fn forward(&self, x: f32) -> f32;
}

struct Square {}

impl Function for Square {
    fn forward(&self, x: f32) -> f32 {
        x.powf(2.0)
    }
}
```

まずは**Function**という**trait**を実装します。ここでは**call**と**forward**というメソッドを作ります。今後このtraitには多くの関数（Exp関数やsin関数など）を追加するためcallにはすべての関数に共通する「**Variable**からデータを取り出す」、「計算結果を**Variable**型にして返す」という2つの機能のみを追加します。具体的な計算はforwardにやらせます。

次にSqureという構造体を実装します. その後implキーワードという定義されたメソッドをimplブロック内で具体的に実装できる機能を用いることで**Squre関数**という2乗の計算を実装します。main関数を実行すると2の2乗の結果がでます。

2.2: Exp関数の実装

新しい関数を1つ実装します。今回はe(ネイピア数)のx乗という関数を実装します。

```
struct Exp {}

impl Function for Exp {
    fn forward(&self, x: f32) -> f32 {
        x.exp()
```

Squre構造体と同じように実装します。変更点はなかの計算がeのx乗に変わっただけです。

3: 微分の理論

微分は機械学習の分野において重要なものです。私たちはこれから微分を使って機械学習の核心を実装していきます。それに先立ち、この章では微分の効率的な求め方について解説します

3.1: チェーンルール

チェーンルールとは複数の関数が組み合わさった「合成関数」を微分する際に、それぞれの関数の微分を掛け合わせることで、合成関数の微分（導関数）を求められるというものです。式にすると

$$\frac{dy}{dx}=\frac{dy}{dy}\cdot\frac{dy}{db}\cdot\frac{db}{da}\cdot\frac{da}{dx}$$

となり、xに関するyの微分は各関数の微分の積によって表わすことができます。つまり合成関数の微分は各関数の局所的な微分へ分解できるということです。

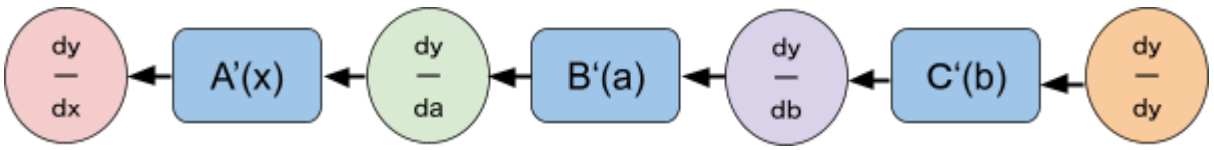
3.2: バックプロパゲーションの理論

$$\frac{dy}{dx}=\frac{dy}{dy}\cdot\frac{dy}{db}\cdot\frac{db}{da}\cdot\frac{da}{dx}$$

の式を変形すると

$$\frac{dy}{dx}=\left\{\left(\frac{dy}{dy}\cdot\frac{dy}{db}\right)\frac{db}{da}\right\}\frac{da}{dx}$$

この式は出力方向から入力方向へ微分を計算していくことを表しています。この計算をグラフによって表すと

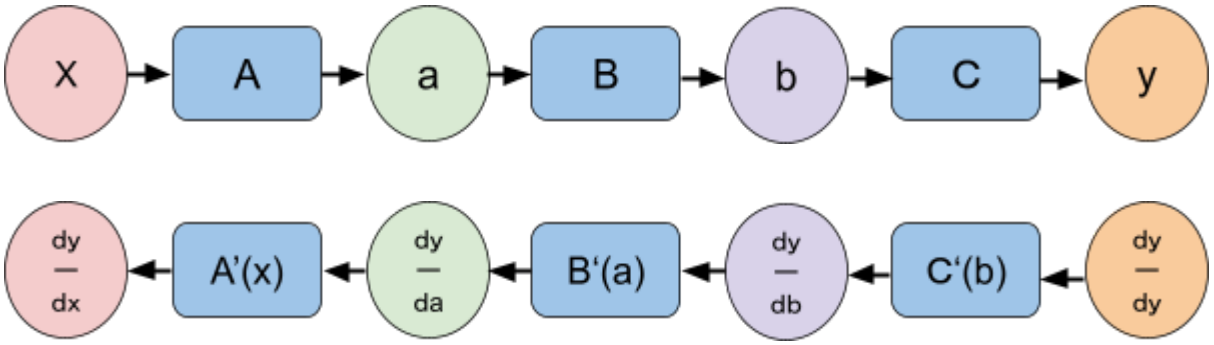


グラフにすることで微分の流れがわかりやすく、このグラフを見るとyの各変数に関する微分が伝播することでxに関するyの微分が求まっていることがわかるこれがバックプロパ

ゲーションです。ここでの重要な点は伝播するデータはすべて「yに関する微分」ということです。

なぜこのバックプロパゲーションが微分の効率的な求め方なのかというと機械学習は基本的に大量のパラメータを入力として「損失関数」を求めていくものです。つまり私たちは損失関数の各パラメータに関する微分を求める必要があります。そのような場合バックプロパゲーションを用いれば一回の伝播で全てのパラメータに関する微分を求められるのです。

ここで通常の計算と微分を求める計算の比較をしてみると



この2つの図を見てわかるように順伝播と逆伝播には明確な対応関係があります。そしてC'(b)に注目するとこの微分の計算をするにはbという値が必要になります. 同様なことが各計算にもいえます。つまり逆伝播をするためには順伝播で求めたデータが必要であるということです。そのためバックプロパゲーションを実装するには順伝播を行い、その各変数の値を保持しなければなりません。このことを念頭において次は実際にバックプロパゲーションを実装していきましょう。

4: 微分の実装（手作業）

前の章では微分の理論について説明してきましたが、この章ではVariable構造体とFunctionトレイトを拡張して微分をもとめていきます。

4.1:Variable構造体とFunctionトレイトの拡張

Variable構造体は微分の値を保持するために、通常の値に加えてそれに対応した微分の値を持つように変更します。いままでのFunctionトレイトは通常の計算をする機能しか持っていませんでした。そのため微分の計算をするbackward機能と、逆伝播のために順伝播する際に入力された値を保持する機能を追加します。

```
#[derive(Debug)]

struct Variable {
    data: f32,
    grad: Option<f32>,

impl Variable {
    fn new(data: f32) -> Self {
        Variable { data, grad: None }

trait Function {
    fn new() -> Self;

    fn call(&mut self, input: & Variable) -> Variable {
        let x = input.data;
        let y = self.forward(x);

        let output = Variable::new(y);
        self.set_input(input);
        output
    }


    fn forward(&self, x: f32) -> f32; // 引数f32
    fn backward(&self, gy: f32) -> f32; // 引数f32

    fn set_input(&mut self, input:Variable) {}
```

Variableは フィールドとして初期状態や勾配が不要な場合はNoneとし、逆伝播で計算された後にはSome(f32)で値を持つようにする

ためOption<f32>型でgradを保持させます。関数**new()**にgradとしてNoneを持たせgradを保持します。

Functionは　　入力されたinputを
Functionインスタンスが、入力されたVariableインスタンスを記憶させるためset_input関数を実装します。なぜ＆mut selfにするのかというとfn set_input は、入力の情報をselfに書き込むために、&mut selfが必須となるからです。

4.2:Squre構造体とExp構造体の拡張

続いて、具体的な関数の逆伝播を実装していきます。まずは2乗の計算をするSqure構造体です。y＝X**2の微分は2Xとなることから実装します。次にy＝e**Xの計算をするExp構造体です。この微分の値はe**Xとなりこれをもとに実装していきます。

```
struct Square{
    input: Option<& Variable>,
}
impl Function for Square {
    fn new() -> Self {
        Self { input: None }
    }
    fn forward(&self, x: f32) -> f32 {
        x.powf(2.0)
    }

    fn backward(&self, gy: f32) -> f32 {
        let x = self.input.unwrap().data;
        2.0 * x * gy // = gx
    }

    fn set_input(&mut self, input: & Variable) {
        self.input = Some(input);
    }
}


struct Exp {
    input: Option<&Variable>,
}
```

```
implFunction for Exp{
    fn new() -> Self {
        Self { input: None }
    }

    fn forward(&self, x: f32) -> f32 {
        x.exp()
    }

    fn backward(&self, gy: f32) -> f32 {
        let x = self.input.unwrap().data;
        x.exp() * gy // = gx
    }

    fn set_input(&mut self, input: &Variable) {
        self.input = Some(input);
    }
}
```

新しい関数インスタンスを作成するためfn new() ->Selfを用いることで Self { input: None } と、初期状態では入力が未設定であることを示し、inputフィールドをNoneで初期化しています。

順伝播時に記憶しておいた入力x の元の値を取り出して微分計算に使用するため self.input.unwrap().dataを使います。なぜunwrap()を使っていいのかというとbackwardメソッドが呼び出される前に、必ずset_inputが呼び出されている（順伝播が完了している）つまりset_inputは必ず何らかの値を持っているためunwrap()が使えます。

逆伝播のためにbackwardメソッドを追加します。このメソッドでは出力側から伝わる微分が渡されます。「その引数で渡された微分の値」と「その関数の微分の値」を掛け算してその値をf32型として返していきます。

4.3:バックプロパゲーションの実装

実際に微分をしてみましょう。

```
fn main() {
    let mut x = Variable::new(0.5);
    println!("{:?}", x);
```

```
    let mut A = Square::new();
    let mut B = Exp::new();
    let mut C = Square::new();


    let mut a = A.call(&x);
    let mut b = B.call(&a);
    let mut y = C.call(&b);


    y.grad = Some(1.0);
    b.grad = Some(C.backward(y.grad.unwrap()));
    a.grad = Some(B.backward(b.grad.unwrap()));
    x.grad = Some(A.backward(a.grad.unwrap()));
    println!("{}", x.grad.unwrap());
}
```

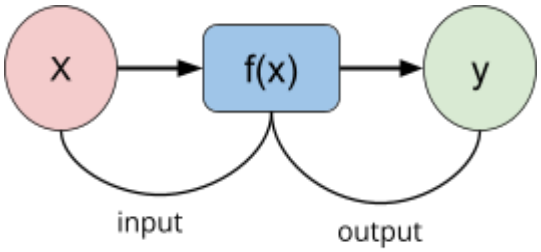
以上が微分の手作業による実装です。次の章ではこの微分を自動化していきます。

5: 微分の実装（自動化）

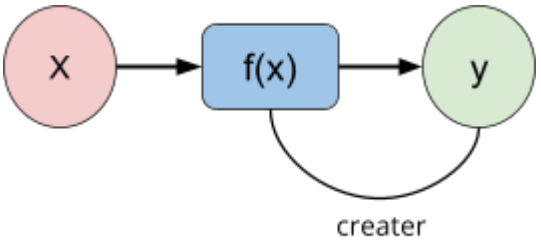
前の章では微分を実装することができました。しかし逆伝播の計算を自分自身で書く必要があります。つまり長い計算グラフを考えたときその逆伝播のコードをすべて手作業で書かなくてはなりません。この章では順伝播が行われたならばその逆伝播を自動的に行われる仕組みを作ります。

5.1:微分の自動化のための変更点

微分を自動化するためには、変数と関数の「つながり」について考えなければなりません。関数の目線から変数がどのようにみえるかというと、変数は「入力される変数」と「出力される変数」の2種類存在します。



続いて、逆に変数の目線から関数がどう見えるのか考えてみましょう。ここで注目すべき点は「関数の作成」の章で言ったように変数は関数によって作り出されるということです。関数は変数を入力として関数に渡し、出力として新たな変数を生み出します。言い換えると変数にとって関数は「creater（生みの親）」です。



ではその関数と変数の「つながり」を私たちのコードに取り入れましょう。

```
struct Variable {
    data: f32,
    grad: Option<f32>,
    creator: Option<Rc<RefCell<dyn Functions>>>,
    name: Option<String>,
}

impl Variable {
    fn new(data: f32) -> Rc<RefCell<Self>> {
        Rc::new(RefCell::new(Variable {
            data,
            grad: None,
            creator: None,
            name: None,
        }))
    fn set_creator(&mut self, func: &Rc<RefCell<dyn Functions>>) {
        self.creator = Some(Rc::clone(func));
    }
```

```
}

trait Function{
    fn call(&mut self, input: &Rc<RefCell<Variable>>) ->
Rc<RefCell<Variable>>;
    fn forward(&self, x: f32) -> f32; // 引数f32
    fn backward(&self, gy: f32) -> f32; // 引数f32
    fn get_input(&self) -> Rc<RefCell<Variable>>;
    fn get_output(&self) -> Rc<RefCell<Variable>>;
```

このコードを説明する前にRc<RefCell型となぜRc<RefCell型を使うのかという説明をしようと思います。

Rcは「参照カウント」型で複数の所有者を可能にしますが内部のデータへの不変な参照しか提供しません。RefCellは実行時における可変性（Interior Mutability）を可能にします。つまりRc<RefCell型は所有権の共有と内部のデータを可変に操作できるというRustでは難しい特徴を両立できる型なのです。私たちは複数のVariableで共有し、必要に応じて内部のデータを変更できるcreaterを作らなければなりません。なのでRc<RefCell型を使うのです。

Variableの変更点について説明します

フィールドとしてOption<Rc<RefCell<Functions>>>型をcreaterとして保持し、Option<String>型をnameとして保持します。また、初期化の関数**new()**にもcreaterとnameを追加します。次にcreatorを保持するための関数set_createrを追加します。この関数はFunctionの共有された所有権をコピーし、Variableのcreatorに格納します。これにより、Variableは、自分を生み出したFunctionが解放されるのを防ぎつつ、その情報にアクセスできるようになります。

Functionの変更点について説明します。

関数としてget_inputとget_outputを追加します。この関数は2つともRc<RefCell<Variable>>型を返すものです。これによって順伝播の計算をすると生成したoutputに「createrが生みの親である」というoutputとcreaterのつながりをつくります。これこそが動的に「つながり」を作る核心の部分です。

5.2:backwardメソッドの追加

5.1で作り出した微分の自動化では
関数を取得して
関数の入力を取得

関数のbackwardメソッドを呼ぶ
という処理が繰り返し登場して、その都度コードを書いて設定をしなくてはなりません。
そこで繰り返しの処理を自動化できるようにVariable構造体にbackwardというメソッドを追加します。

```
 fn backward(&self) {
        let mut funcs: Vec<Rc<RefCell<dyn Functions>>> =
            vec![Rc::clone(self.creator.as_ref().unwrap())];

    let mut last_variable = true;
        while let Some(f) = funcs.pop() {
            let x = f.borrow().get_input();

            if last_variable {
                let y_grad: f32 = 1.0;
                x.borrow_mut().grad = Some(f.borrow().backward(y_grad));
                last_variable = false;
            } else {
                let y = f.borrow().get_output();
                let y_grad = *y.borrow().grad.as_ref().unwrap();
                x.borrow_mut().grad = Some(f.borrow().backward(y_grad));
            }

            if x.borrow().creator.is_none() {
                break;
            };

            funcs.push(Rc::clone(x.borrow().creator.as_ref().unwrap()));
        }
    }
}
```

このbackwardメソッドではVariableのcreaterから関数を受け取り、その関数の入力を取り出した後にbackwardメソッドを呼び出すという処理をループを使って実装しています。詳細な内容としては

```
let mut funcs: Vec<Rc<RefCell<Functions>>> =
            vec![Rc::clone(self.creator.as_ref().unwrap())];
```

で最初の要素として、現在のVariable(self)を生み出したFunctionのcreaterを取得します。unwrap()は、このVariableが既に何らかの計算結果であるという前提(終端ではない)なので使えます。

`let x = f.borrow().get_input();:` では取り出したFunctionの入力変数(Variable)を取得します。

今回は最初に処理するVariableか途中のVariableかで場合分けします。なぜかというと最初に処理するVariableはdy/dyなので絶対1です.途中のVariableは以前のFunctionから勾配を取得する必要があるからです。

y_grad: Functionは出力側から伝わってきた微分の値です。

`f.borrow().backward(y_grad)`: ではFunction のbackwardメソッドを実行し、微分の値を計算します。

`x.borrow_mut().grad = Some(...)`:では 計算された微分の値をVariable のgrad フィールドに書き込みます。`borrow_mut()`が使われているのは、内部のgradフィールドを変更するためです。

`if x.borrow().creator.is_none()` がtrueの場合、Variable はこれ以上遡る必要のないノードです。つまり微分がすべて終わったことを意味します。ここでループを終了する必要があるのです。

`funcs.push(...)`では、次のFunctionをスタックに追加します。これにより、次のループでそのFunctionを処理することができます。

5.3:各関数の変更点

5.2によってVariableの自動微分はできましたが、まだ各関数には対応していません。なのでつぎはSqure構造体を例にして実装していきましょう

```
impl Function for Square {
       fn call(&mut self, input: &Rc<RefCell<Variable>>) ->
Rc<RefCell<Variable>> {
        let x = input.borrow().data;
```

```
        let y = self.forward(x);

        let output = Variable::new(y);

        self.input = Rc::clone(input);

      self.output = Rc::downgrade(&output);

        let self_square: Rc<RefCell<dyn Function>> =
Rc::new(RefCell::new(self.clone()));
        output.borrow_mut().set_creator(&self_square);

        output
    }


        let x = self.input.borrow().data;
        2.0 * x * gy // = gx
    }

    fn get_input(&self) -> Rc<RefCell<Variable>> {
        Rc::clone(&self.input)
    }

    fn get_output(&self) -> Rc<RefCell<Variable>> {
        Rc::clone(self.output.upgrade().as_ref().unwrap())
    }
}
impl Square {
    fn new() -> Rc<RefCell<Self>> {
        Rc::new(RefCell::new(Self {
            input: Variable::new(0.0),
            output: Rc::downgrade(&Variable::new(0.0)),
        }))
    }
}
```

Square::new()の変更点

Rc<RefCell<Self>>を返すことで、Square関数自体が共有可能かつ内部可変なオブジェクトとして扱われるようにします。

逆伝播で値を取得するために使用するためVariableへの強い参照（所有権を共有）つまりinputの値を長時間保持しなくてはならないため。self.input = Rc ::clone(input);でcloneを使っています。

self.output = Rc::downgrade(&output);はなぜcloneしないのかというともしFunctionがoutputを強い参照で持ってしまうと、Variableのcreatorフィールドが Functionを参照し、Functionがoutputを参照するという循環参照が発生します。そのため 参照カウントを増やさないdowngradeを使っています。

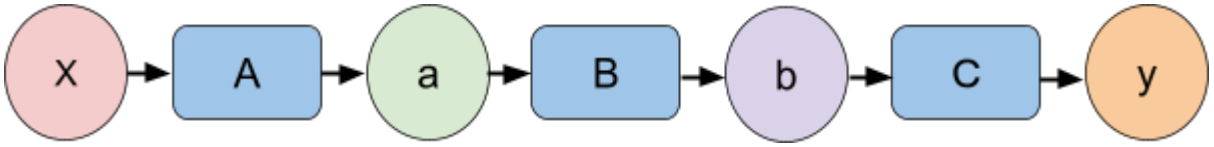
output.borrow_mut().set_creator(...):では 出力Variableのcreatorフィールドに、この計算を行ったFunctionインスタンスへの参照を与えることで逆伝播の経路をつくります。

これを参考にして自分自身の手でExp関数を変更してみましょう。答えはGithubリポジトリを参照してください。

5.4:add関数の導入と2変数への拡張

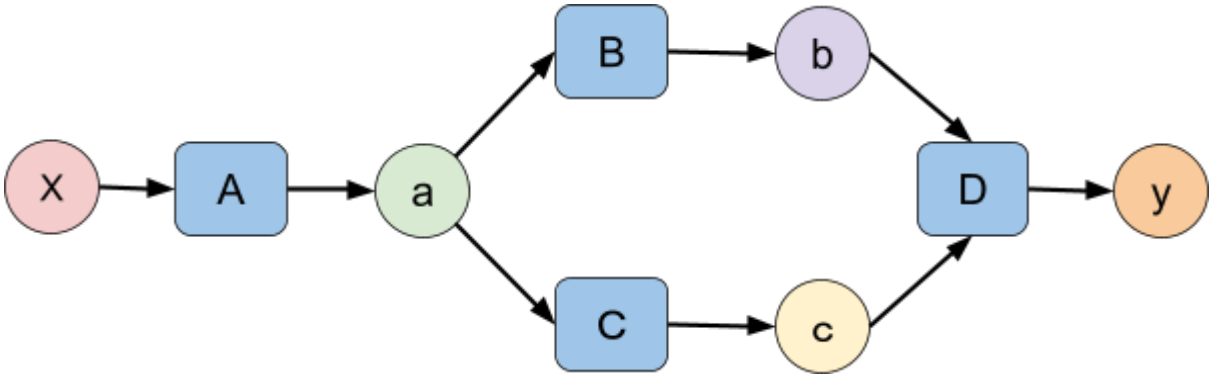
6: 微分の理論（複雑な計算）

これまでわたしたちは一直線に並ぶ計算グラフを扱ってきました。

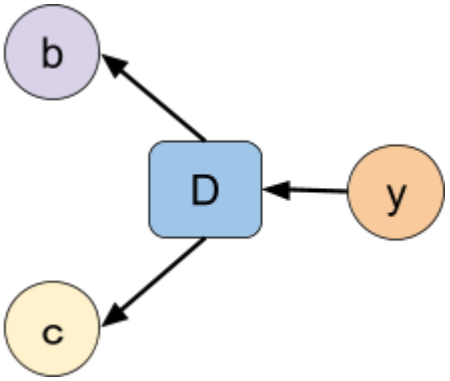


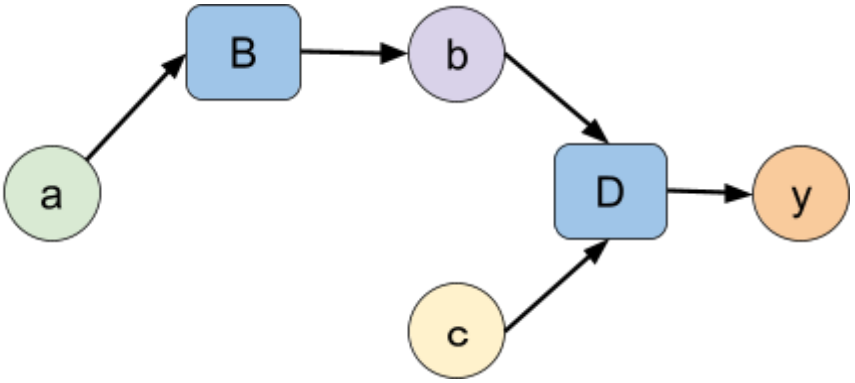
しかし変数と関数の「つながり」はそのような一直線とは限りません。これまでの実装により私たちはadd関数や2変数への拡張を行ってきました。それによってより複雑な「つながり」を作ることができました。しかし今のフレームワークでは複雑な「つながり」の逆伝播をすることができません。

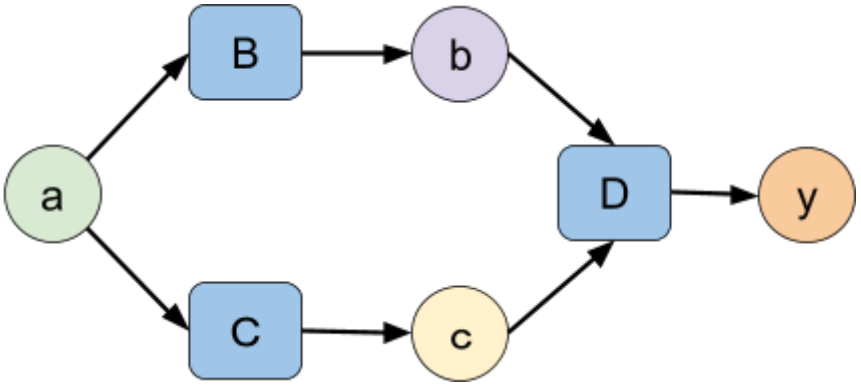
はたして今のフレームワークにどのような問題があるのか調べるために1つのシンプルな計算グラフについて考えましょう。

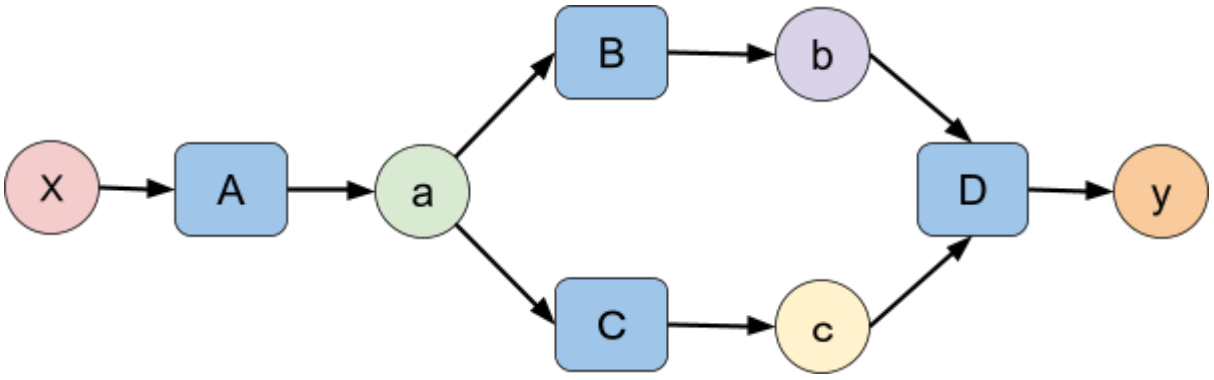


この計算グラフで注目したい点は変数aです. 前の章で紹介したようにaの微分にはaの出力側から伝播する2つの微分が必要になります。その点を注力して正しい計算グラフを考えると

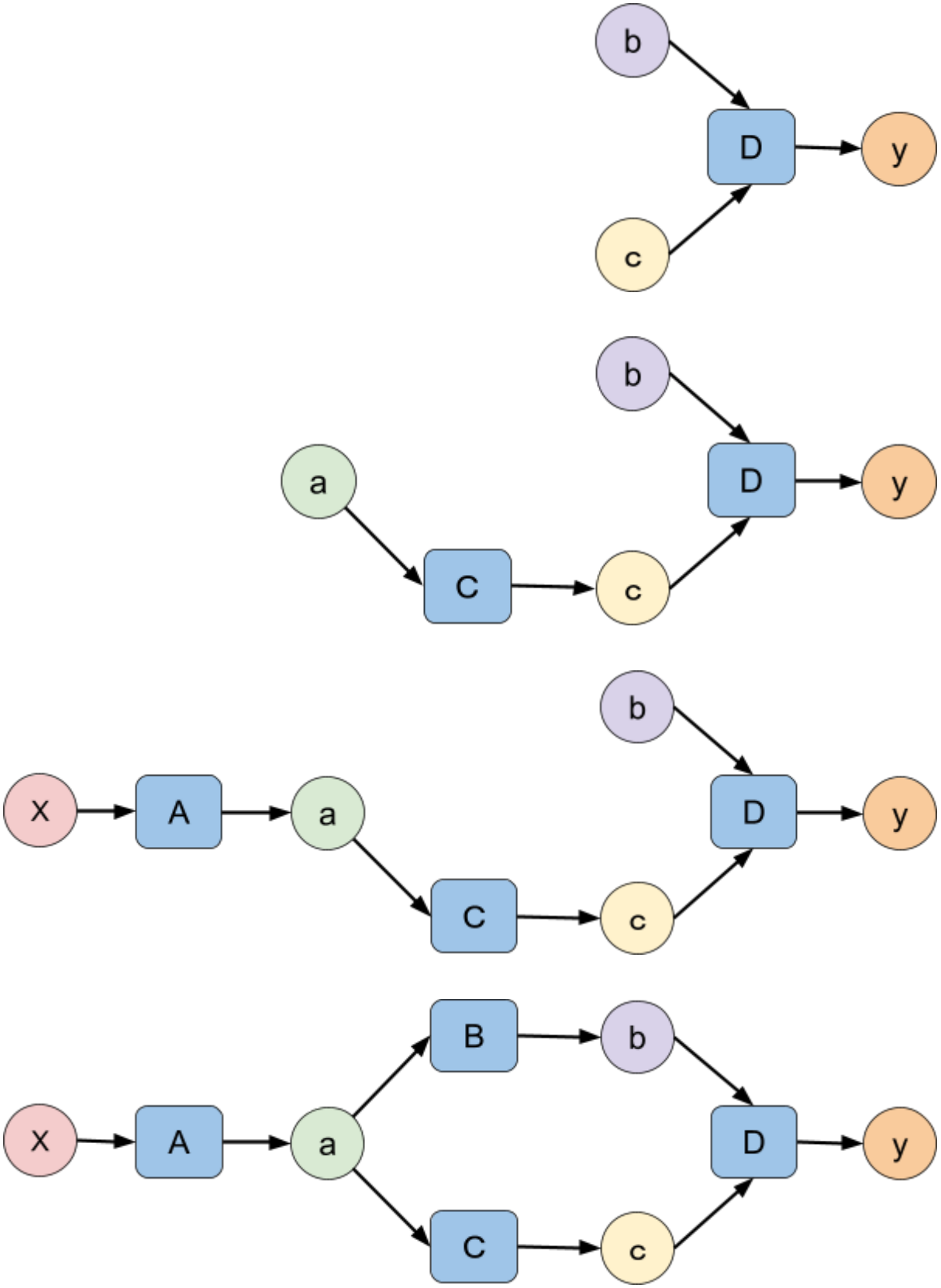


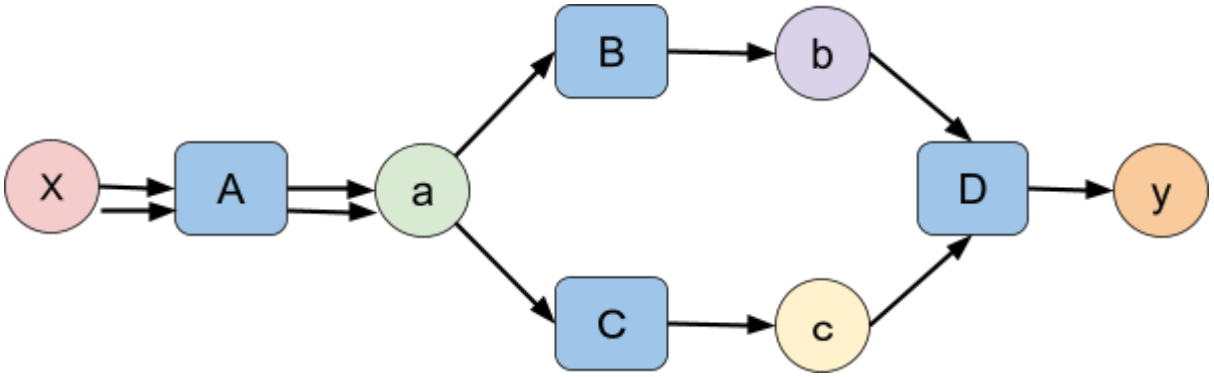




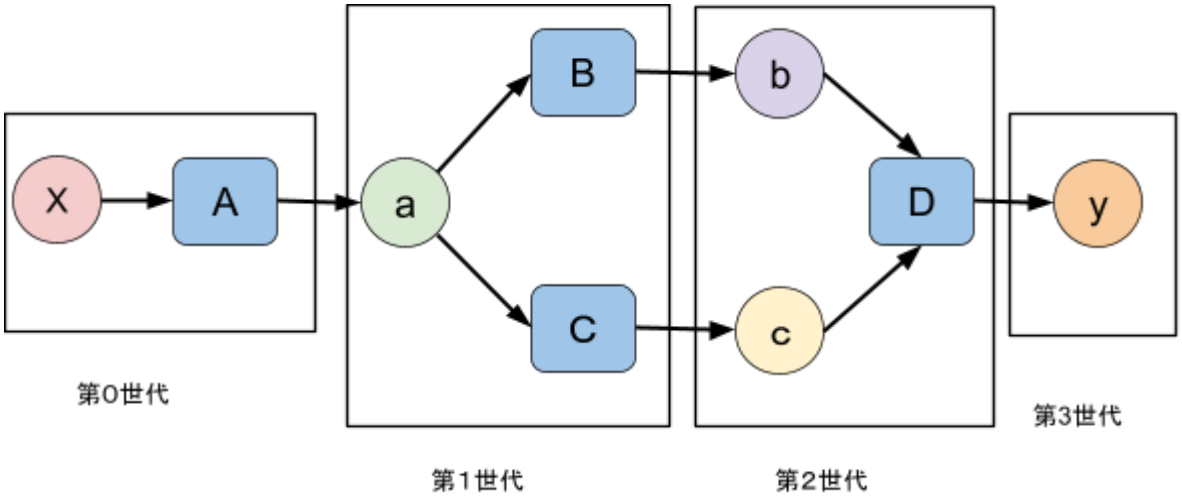


という順番で伝播していますが関数の目線からいうとD、B、C、Aの順番で逆伝播を行っています。大切なのは関数B、Cの逆伝播を終わらせてから、関数Aの逆伝播をおこなうということです。続いて私たちの計算グラフを示します。


b
D
y
c
b
a
D
y
C
c
b
X
A
a
D
y
C
c
B
b
X
A
a
D
y
C
c



となってしまいました。私たちがかんがえなくてはならないのは、いままでのbackwardメソッドでは計算グラフが直線的だったため、何も考えずに前の関数を呼んでいましたが今後は適切な関数を取り出すことが求められます。つまり関数に「優先度」を与えなくてはなりません。この計算グラフの例でいえばAよりもBが高い優先度を持っていればそれにしたがってBを先に取り出します。それではどのように「優先度」を設定しましょうか？私たちは順伝播を行うとき「関数」が「変数」を生み出すという「つながり」を作っています。その「つながり」によって関数と変数の関係を次のように表せます。



このようにすれば正しい順序で逆伝播を実装することができます。では次の章ではこの「世代」を実装していきましょう。

7: 微分の実装（複雑な計算）

まずVariableにgenerationというフィールドを持たせましょう。このgenerationには世代の値を保持します。例えば前のグラフの**X**は第0世代なので0という値を持ちます。

さらに関数が複雑になり、変数(**Variable**)と関数(**Function**)が増えてくると、管理が大変なため、今後のためにそれぞれの構造体に**id**をつけましょう。

```
struct Variable {
    data: f32,
    grad: Option<f32>,
    creator: Option<Rc<RefCell<dyn Function>>>,
    name: Option<String>,
    generation: i32,
    id: u32,
}


impl Variable {
    fn new(data: f32) -> Rc<RefCell<Self>> {
        let id = NEXT_ID.fetch_add(1, Ordering::SeqCst);
        Rc::new(RefCell::new(Variable {
            data,
            grad: None,
            creator: None,
            name: None,
            generation: 0,
            id: id,
        }))
    }
```

ここではgenerationをi32型、idをu32型に設定します。

2: 演算子のオーバーロード

前のステップでは実数による自動微分を実装しました。しかしまだ演算子に対応しておらず計算をするためにいちいち書かなくてはいけません。このステップの目標としては＋や＊などの演算子に対応することです。例をあげるなら、aとbをVariable構造体としたときに　y＝a*bと書けると便利になります。これから＋や*など演算子が扱えるようにVariable構造体を拡張していきます。

1: 掛け算と割り算をする関数の作成

まず演算子を使えるようにVariable構造体を拡張する前に私たちのフレームワークに掛け算や割り算をする関数を実装する必要があります。

2: 演算子のオーバーロード