機械学習講習会 第三回

- 「自動微分」

traP アルゴリズム班 Kaggle部

2023/xx/xx

今日は講義内で演習もします

第三回: 自動微分

前回のまとめ

- 損失関数の最小化を考える上で、一般の関数の最小化を考えることにした
- 損失関数の厳密な最小値を求める必要はなく、また損失関数は非常に複雑になりうるので、広い範囲の関数に対してそこそこ上手くいく方法を考えることにした
- たいていの関数に対して、導関数を求めることさえできればそれなりに小さい値を探しに行けるようになった
- 逆に、「導関数」は自分で求める必要がある

第三問

最小化してください。

$$\frac{-\log(\frac{1}{1+e^{-x}}+1)}{(x^2+1)}$$

😅

2. fは非常に複雑になりうる

第一回では $f(x) = ax + b$の形を考えたが...

(特にニューラルネットワーク以降) は複雑になる

$$L(W_1, W_2, W_3, \boldsymbol{b_1}, \boldsymbol{b_2}, \boldsymbol{b_3}) = \frac{1}{n}\sum\left(\boldsymbol{y} - W_3\sigma(W_2\sigma(W_1\boldsymbol{x} + \boldsymbol{b_1}) + \boldsymbol{b_2}) + \boldsymbol{b_3})\right)^2$$

$$\sigma(x) = \frac{1}{1 + e^{-x}}$$

(というか、普段我々が使っている数学の記号では書けなくなる)

自動微分

✅ 人間が微分を行うのは限界がある
⇨ コンピュータにやらせよう！

自動微分

正確には「自動微分」は、コンピュータに自動で微分を行わせる手法のうち、関数を単純な関数の合成と見て、特に連鎖律を利用して、陽に導関数を求めることなく微分を行う手法を指します(より狭義に、back propagationを用いるもののみを指すこともあるようです)。　詳しくは、資料末の発展事項を参照してください。

おしながき

- PyTorchの導入
- PyTorchを使った自動微分
- 自動微分を使った勾配降下法の実装
- 発展的話題: コンピュータに微分をやらせる試み
    - ~自動微分のアルゴリズム~

PyTorch

結論: PyTorchを使うと微分ができる.

```
>>> x = torch.tensor(2.0, requires_grad=True)
>>> def f(x):
...     return x ** 2 + 4 * x + 3
...
>>> y = f(x)
>>> y.backward()
>>> x.grad
tensor(8.)
```

($f(x) = x^2 + 4x + 3$の$x = 2$における微分係数8)

そもそもPyTorchとは？　〜深層学習フレームワーク〜

ニューラルネットワーク・ディープラーニングのさまざまな派生系の

- 基本的に構成する部品
- 部品に対してやる作業

は大体同じ！

例) 新しい車を開発するときも、部品(ネジ、タイヤ、エンジンの部品...)は大体同じ、組み立ても大体同じ

⇨ 毎回同じことをみんながそれぞれやるのは面倒
⇨ 共通の「基盤」を提供するソフトウェアの需要がある

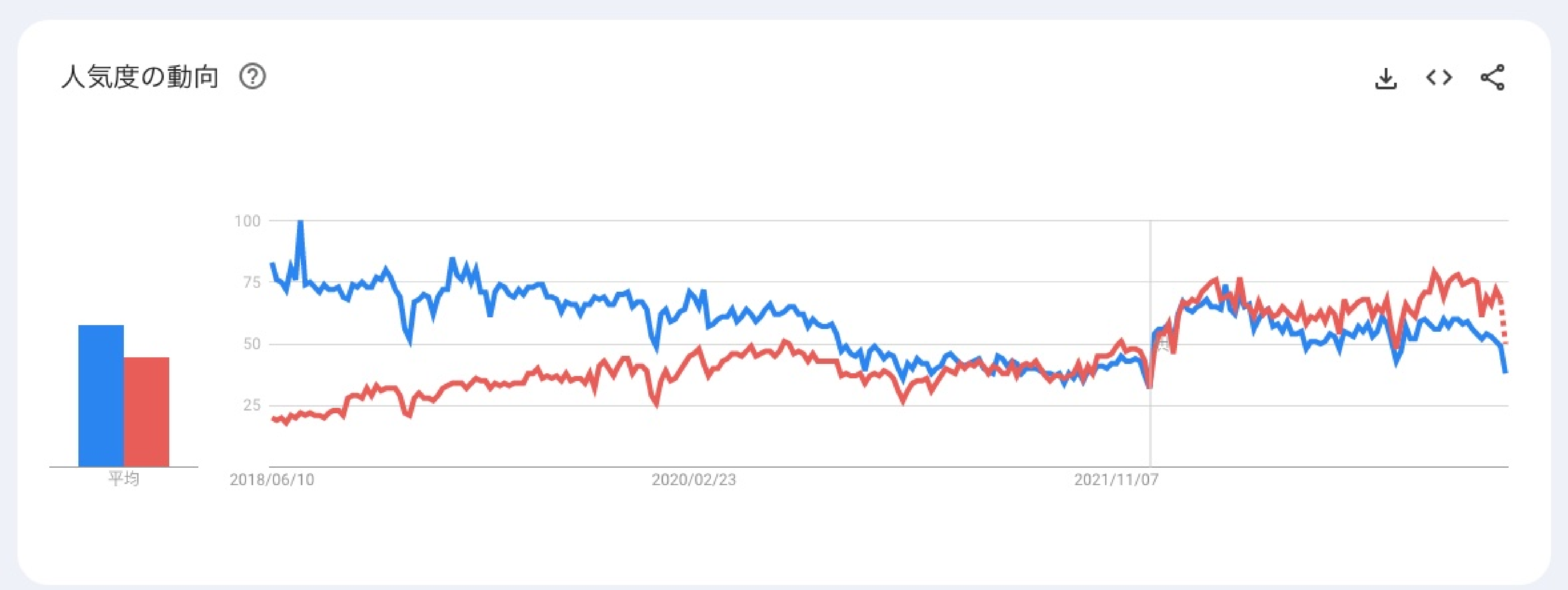
有名なフレームワークたち

- TensorFlow
  - (主に)Googleが開発したフレームワーク
  - 産業界で人気(が、最近はPyTorchに押され気味)
- PyTorch
  - Facebookが開発したフレームワーク
  - 研究界で人気(最近はみんなこれ？)
- Keras
  - TensorFlowを使いやすくしたラッパー
  - とにかくサッと実装できる

個人的には、最近Googleが力を入れているjax/flaxに注目しています。関数型プログラミングの考え方を取り入れていて、最近人気が出始めています。

そもそもPyTorchとは？　～深層学習フレームワーク～

どれがいいの？
⇨ PyTorchを使っておけば間違いない(と、思います)



(赤: PyTorch, 青: TensorFlow)

なので

今回は**PyTorch**を使います！

- C++, CUDAバックエンドの高速な実行
- 非常に柔軟な記述
- 充実した周辺ライブラリ
- サンプル実装の充実 (← 重要!!)

大体の有名フレームワークにそこまで致命的な速度差はなく、記述に関しては好みによるところも多いです。
PyTorchの差別化ポイントは、有名モデルの実装サンプルが大体存在するという点です。
実際に論文を読んで実装するのは骨の折れる作業なので、サンプルが充実していのはとても大きな利点です。

Tensor型

数学の「数」に対応するオブジェクトとして、
PyTorchでは

✅ **Tensor型: 高効率で勾配を保持できる多次元配列**

を使う

⬇ 多次元配列とは？

多次元配列

スカラ・ベクトル(配列)・行列 ... の一般化

- スカラ: 0次元配列
- ベクトル: 1次元配列
- 行列: 2次元配列

```
v = [1, 2, 3]

A = [[1, 2, 3],
     [4, 5, 6],
     [7, 8, 9]]

T = [[[1, 2, 3], [4, 5, 6]],
     [[7, 8, 9], [10, 11, 12]]]
```

Tensor型

```
>>> x = torch.tensor(2.0, requires_grad=True)
```

2.0というスカラを保持するTensor型のオブジェクトを作成
(数 $x = 2.0$を定義)

```
>>> x = torch.tensor([1.0, 2.0, 3.0], requires_grad=True)
```

[1.0, 2.0, 3.0]というベクトルを保持するTensor型のオブジェクトを作成
(数 $\vec{x} = (1.0, 2.0, 3.0)$を定義)

かつては自動微分には `Variable` という名前の型が使われていて、(現在はTensor型に統合された)　Tensorと数学の変数の概念にある程度の対応があることがわかります。

Tensor型

```
torch.tensor(data, requires_grad=False)
```

- `data` : 保持するデータ(配列**っぽい**ものならなんでも)
  - リスト、タプル、NumPy配列、スカラ、…
- `requires_grad` : 勾配(gradient)を保持するかどうかのフラグ
  - デフォルトはFalse
  - 勾配の計算(自動微分)を行う場合はTrueにする

Tensor型

```
>>> x = torch.tensor([[1.0, 2.0, 3.0], [4.0, 5.0, 6.0]], requires_grad=True)
```

[[1.0, 2.0, 3.0], [4.0, 5.0, 6.0]]という行列を保持するTensor型のオブジェクトを作成
(数 $X = \begin{pmatrix} 1.0 & 2.0 & 3.0 \\ 4.0 & 5.0 & 6.0 \end{pmatrix}$ を定義)

(`requires_grad=True` とすれば、勾配計算が可能なTensor型を作成できる)

演習1

これらを勾配計算が可能なTensor型として表現してください。

1. $x = 3.0$
2. $\vec{x} = (3.0, 4.0, 5.0)$
3. $X = \begin{pmatrix} 3.0 & 4.0 & 5.0 \\ 6.0 & 7.0 & 8.0 \end{pmatrix}$

(このページの内容は、実際にやらなくてもやり方がわかればOKです)

↓ 問題の続き次のページへ

演習1

(実際にやってください)

4. **整数** $x = 3$を勾配計算が可能なTensor型として表現することを試みてください。また、その結果を確認して説明できるようにしてください。
5. 1から100までの値を並べた10x10行列、つまり

$$X = \begin{pmatrix} 1 & 2 & \cdots & 10 \\ 11 & 12 & \cdots & 20 \\ \vdots & \vdots & \ddots & \vdots \\ 91 & 92 & \cdots & 100 \end{pmatrix}$$

を勾配計算が可能なTensor型として表現してください。(次ページヒント)

演習1 ヒント

1, 2, 3: 講義資料を遡って、 `torch.tensor` の第一引数と作成されるTensor型の対応を見比べてみましょう。

4: Pythonのエラーは、

```
~~たくさん書いてある~
~~Error: {ここにエラーの概要}
```

という形式です。"~~Error"というところのすぐ後に書いてある概要を確認してみましょう。

5: 3が解けたのであれば、 `torch.tensor` の第一引数にどのようなオブジェクトをが来るべきかはわかるはずです。あとはそのリストの構築方法を考えましょう。

演習1 解答

1~3.

```
# 1
x = torch.tensor(3.0, requires_grad=True)
# 2
x = torch.tensor([3.0, 4.0, 5.0], requires_grad=True)
# 3
x = torch.tensor([[3.0, 4.0, 5.0], [6.0, 7.0, 8.0]], requires_grad=True)
```

次のページへ

演習1: 解答

4.

```
x = torch.tensor(3, requires_grad=True)
```

と入力してみると、

"RuntimeError: Only Tensors of floating point and complex dtype can require gradients"となります。これは、勾配が計算可能なのは浮動小数点数と複素数のみであるということを意味しています。

次のページへ

演習1: 解答

1. (解答例)　一番素直なやつ

```
>>> matrix = []
>>> for i in range(10):
...     row = []
...     for j in range(10 * i + 1, 10 * i + 11):
...         row.append(float(j))
...     matrix.append(row)
...
>>> torch.tensor(matrix, requires_grad=True)
```

6. (解答例2) 一番素直なやつ 内包表記ver

```
>>> x = torch.tensor([[float(i) for i in range(10 * j + 1, 10 * j + 11)] for j in range(10)]
```

他にも別解はたくさんありますが、書き方に関わらず作れればここまでの内容は理解できているので💯)

Tensor型に対する演算

Tensor型は、「数」なので当然各種演算が可能

```python
x = torch.tensor(2.0, requires_grad=True)
```

- 四則演算

```python
x + 2
# -> tensor(4., grad_fn=<AddBackward0>)
```

```python
x * 2
# -> tensor(4., grad_fn=<MulBackward0>)
```

Tensor型に対する演算

各種 数学的な(?) 関数も利用可能

```
torch.sqrt(x)
# -> tensor(1.4142, grad_fn=<SqrtBackward0>)
```

```
torch.sin(x)
# -> tensor(0.9093, grad_fn=<SinBackward0>)
```

```
torch.exp(x)
# -> tensor(7.3891, grad_fn=<ExpBackward0>)
```

自動微分

ここまでの内容は別にPyTorchを使わなくてもできること
PyTorchは、計算と共に勾配の計算ができる！

✅ `requires_grad=True` **であるTensor型に対して計算を行うと、行われた演算が記録されたTensorができる.**

```
x = torch.tensor(2.0, requires_grad=True)
```

足し算をする。

```
y = x + 2
```

⬇

自動微分

```
print(y)
```

これの出力は、
tensor(4., grad_fn=<**Add**Backward0>)
⇨ **Add**という演算が `y` に記録されている！

普通のPythonの数値では、

```
x = 2
y = x + 2
print(y) # -> 4.0
```

`y` がどこから来たのかはわからない(値として4.0を持っているだけ)

自動微分

✅ PyTorchは、 `backward` 関数をつかって記録された演算を辿ることで、勾配を計算できる

```
x = torch.tensor(2.0, requires_grad=True)
y = x + 2
```

⬇

```
y.backward()
```

⬇

```
print(x.grad) # -> tensor(1.)
```

自動微分の流れ

1. 変数(Tensor型)の定義
2. 計算
3. backward()

```python
# 1. 変数(Tensor型)の定義
x = torch.tensor(2.0, requires_grad=True)
# 2. 計算
y = x + 2
# 3. backward()
y.backward()
```

すると、`x.grad` に計算された勾配が格納される。

なぜこんな設計なのか気になった人は、講義が終わったら資料末の「発展的話題: 自動微分のアルゴリズム」を読んでみてください。現段階では、今回はこのセットで計算できる！ということを覚えてもらえればokです。

演習2: 100回唱えよう！

定義→計算→backward(), 定義→計算→backward(), 定義→計算→backw
定義→計算→backward(), 定義→計算→backward(), 定義→計算→backw
定義→計算→backward(), 定義→計算→backward(), 定義→計算→backw
定義→計算→backward(), 定義→計算→backward(), 定義→計算→backw
定義→計算→backward(), 定義→計算→backward(), 定義→計算→backw
定義→計算→backward(), 定義→計算→backward(), 定義→計算→backw
定義→計算→backward(), 定義→計算→backward(), 定義→計算→backw
定義→計算→backward(), 定義→計算→backward(), 定義→計算→backw
定義→計算→backward(), 定義→計算→backward(), 定義→計算→backw
定義→計算→backward(), 定義→計算→backward(), 定義→計算→backw
定義→計算→backward(), 定義→計算→backward(), 定義→計算→backw
定義→計算→backward(), 定義→計算→backward(), 定義→計算→backw
定義→計算→backward(), 定義→計算→backward(), 定義→計算→ba

ありとあらゆる演算が自動微分可能

例1) $f(x) = sin((x+2)+(1+e^{x^2}))$ の微分

```
x = torch.tensor(2.0, requires_grad=True)
y = y = torch.sin((x + 2) + (1 + torch.exp(x ** 2)))
y.backward()
print(x.grad) # -> tensor(-218.4625)
```

例2) $y = x^2, z = 2y + 3$ の微分($\frac{dz}{dx}$)

```
x = torch.tensor(2.0, requires_grad=True)
y = x ** 2
z = 2 * y + 3
z.backward()
print(x.grad) # -> tensor(8.)  ... backward()した変数に対する勾配！(この場合はz)
```

ベクトル、行列演算の勾配

```
x = torch.tensor([1.0, 2.0, 3.0], requires_grad=True)
y = 2 * x[0] + 3 * x[1] + 4 * x[2]
y.backward()
print(x.grad) # -> tensor([2., 3., 4.])
```

$$\vec{x} = (x_1, x_2, x_3)^\mathsf{T}$$

$$y = 2x_1 + 3x_2 + 4x_3$$

$$\frac{dy}{d\vec{x}} = \left(\frac{dy}{dx_1}, \frac{dy}{dx_2}, \frac{dy}{dx_3}\right)^\mathsf{T} = (2, 3, 4)^\mathsf{T}$$

と対応

ベクトル、行列演算の勾配

```
A = torch.tensor([[1.0, 2.0, 3.0], [4.0, 5.0, 6.0]], requires_grad=True)
y = torch.sum(A)
y.backward()
print(A.grad) # -> tensor([[1., 1., 1.],
              #           [1., 1., 1.]])
```

$$A = \begin{pmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \end{pmatrix},\ y = \sum_{i=1}^{2}\sum_{j=1}^{3} a_{ij} = 21$$

$$\frac{dy}{dA} = \begin{pmatrix} \frac{dy}{da_{11}} & \frac{dy}{da_{12}} & \frac{dy}{da_{13}} \\ \frac{dy}{da_{21}} & \frac{dy}{da_{22}} & \frac{dy}{da_{23}} \end{pmatrix} = \begin{pmatrix} 1 & 1 & 1 \\ 1 & 1 & 1 \end{pmatrix}$$

と対応

多変数関数の微分

```
x = torch.tensor(2.0, requires_grad=True)
y = torch.tensor(3.0, requires_grad=True)
z = 2 * x + 4 * y
z.backward()
print(x.grad) # -> tensor(2.)
print(y.grad) # -> tensor(4.)
```

$$z = 2x + 4y$$

$$\frac{\partial z}{\partial x} = 2, \ \frac{\partial z}{\partial y} = 4$$

に対応

実際に適用される演算さえ微分可能ならOK

```python
x = torch.tensor(2.0, requires_grad=True)
def f(x):
    return x + 3
def g(x):
    return torch.sin(x) + torch.cos(x ** 2)

if rand() < 0.5:
    y = f(x)
else:
    y = g(x)
```

✅ ポイント: 実際に適用される演算は、実行してみないとわからないが、適用される演算はどう転んでも微分可能な演算なのでOK.
(if文があるから, for文があるから, 自分が定義した関数に渡したから...ということは関係なく、実際に適用される演算のみが問題になる)

抑えてほしいポイント

- 任意の(勾配が定義できる)計算をTensor型に対して適用すれば、常に自動微分可能
- **定義→計算→backward()** の流れ
- ベクトル、行列など任意のTensor型について微分可能。多変数関数の場合も同様
- 「実際に適用される演算」さえ微分可能ならOK

演習3: 自動微分

1. $y = x^2 + 2x + 1$ の$x = 3.0$における微分係数をPyTorchを使って求めよ。
2. $y = f(\vec{x}) = {x_1}^2 + {x_2}^2 + {x_3}^2$の $\vec{x} = (1.0, 2.0, 3.0)^\mathsf{T}$における勾配をPyTorchを使って求めよ。
3. $W = \begin{pmatrix} 1.0 & 2.0 & 3.0 \\ 4.0 & 5.0 & 6.0 \end{pmatrix}, \vec{x_1} = (1.0, 2.0), \vec{x_2} = (1.0, 2.0, 3.0)^\mathsf{T}$に対して、$y = f(W, \vec{x_1}, \vec{x_2}) = \vec{x_1} W \vec{x_2}$の勾配をPyTorchを使って求めよ。なお、行列積は `torch.matmul` 関数で利用できる。

(次ページヒント)

演習3: 解答

1.

```python
x = torch.tensor(3.0, requires_grad=True)
y = x ** 2 + 2 * x + 1
y.backward()
print(x.grad) # -> tensor(8.)
```

2.

```python
x = torch.tensor([1.0, 2.0, 3.0], requires_grad=True)
y = x[0] ** 2 + x[1] ** 2 + x[2] ** 2
y.backward()
print(x.grad) # -> tensor([2., 4., 6.])
```

演習3: 解答

3.

```
W = torch.tensor([[1.0, 2.0, 3.0], [4.0, 5.0, 6.0]], requires_grad=True)
x1 = torch.tensor([[1.0, 2.0]], requires_grad=True)
x2 = torch.tensor([1.0, 2.0, 3.0], requires_grad=True)
y = torch.matmul(torch.matmul(x1, W), x2)
y.backward()
print(W.grad)
print(x1.grad)
print(x2.grad)
```

思い出すシリーズ: 勾配降下法のPyTorchによる実装

$f(x) = x^2 + e^{-x}$の勾配降下法による最小値の探索

```python
from math import exp

x = 3
lr = 0.0005

# xでの微分係数
def grad(x):
    return 2 * x - exp(-x)

for i in range(10001):
    # 更新式
    x = x - lr * grad(x)
    if i % 1000 == 0:
        print('x_', i, '=', x)
```

勾配降下法のPyTorchによる実装

これまでは、導関数 `grad` を我々が計算しなければいけなかった
⇨自動微分で置き換えられる！

```python
import torch

x = torch.tensor(3.0, requires_grad=True)

def f(x):
    return x ** 2 + torch.exp(-x)

for i in range(10001):
    y = f(x)
    y.backward()
    x = x - lr * x.grad
```

実際に動かすにあたっては軽微な修正が必要ですが、スペースが足りないのでここには載せていません。
詳しくは配布のソースコードを参照してください。

今ならこいつを倒せるはず

最小化してください。

$$\frac{-\log\left(\frac{1}{1+e^{-x}}+1\right)}{(x^2+1)}$$

演習4

1. 複雑な関数

$$f(x) = \frac{-\log(\frac{1}{1+e^{-x}}+1)}{(x^2+1)}$$

の最小値を、PyTorchを用いて勾配降下法を実装することで求めてください。

実装にあたっては、配布ソースコードのPyTorchによる勾配降下法」の項を参照してください。

(次ページ次の問題)

演習4

1. アイスの問題

$$f(x; a, b) = ax + b$$

とします。アイスの売り上げと気温の関係は、リンク先からコピーできます。それぞれのデータが(x_i, y_i)として与えられているとき、

$$L(a, b) = \sum_{i=1}^{n} (y_i - f(x_i; a, b))^2$$

を最小にするa, bをPyTorchを用いて勾配降下法を実装することで求めてください。(次ページ次の問題)

演習4

3. 損失関数

第一回の講義資料では次のような文章がありました。

「「悪さ」を最小化するのではなく「良さ」を最大化すれば良くない？と思った人もいるかもしれないですね。
実はそれはものすごくいい疑問で、次回以降で明らかになっていきます。
ひとまずは一旦疑問として抱えておいてください。」

これに対する答えを考えてみてください。
より具体的に、たとえば「明日は晴れかどうか予測する」、という問題に対して例えば「正解率」を最大化することを考えた場合、どのような点で困るか考えてください。

演習

1, 2.配布ソースコードの該当箇所を確認してよく見比べてください。

3. 勾配降下法を使った学習ではパラメータによる損失(今回の場合は損失ではないですが)の微分を考えていました。つまり、パラメータが微小に変化したときに、損失がどれだけ変化するかを考えていました。今回の場合はこの値はどうなるでしょうか？

演習　解答

1. サンプルコードを参照
2. サンプルコードを参照

演習　解答

2.

出力は最終的に「晴れ」or「晴れじゃない」のいずれかに割り振られる。
したがって、パラメータが微小に変化したとしても、出力は変化しない。
つまり、ほとんどの場合で微分係数が0になってしまう。これでは勾配降下法の枠組みで学習を進めることができない。

発展的話題

発展的話題: 自動微分のアルゴリズム

これらが気になる人向け

- そもそもどうやって微分をしているのか？
- `backward()` 関数はどういう働きをしているのか？
- どうして定義→計算→backward()みたいな使い方をするのか？

めちゃx2余談です

発展的話題: 自動微分のアルゴリズム

✅ ポイント:
最適化の文脈では、基本的にほしいものは「微分係数」であって「導関数」である必要はない

人間が微分をする場合...

例) $f(x) = x^2 + 2x + 1$の$x = 3.0$における微分係数を求めろ

↓

$\underline{f'(x) = 2x + 2}$だから、$f'(3.0) = 8.0$

発展的話題: 自動微分のアルゴリズム

PyTorchでは...

例) $f(x) = x^2 + 2x + 1$の$x = 3.0$における微分係数を求めろ

↓

```
x = torch.tensor(3.0, requires_grad=True)
y = x ** 2 + 2 * x + 1
y.backward()
x.grad # -> tensor(8.)
```

陽に導関数を求めておらず、直接導関数を求めている

発展的話題: 自動微分のアルゴリズム

どうやって？

コンピュータで微分を求めるアルゴリズムの分類

1. 数値微分
2. 数式微分・記号微分
3. 自動微分

数値微分

微分の定義式から直接近似する。

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

⬇ そのままPythonに

```
def diff(f, x, h=1e-4):
    return (f(x + h) - f(x)) / h
```

コンピュータ上で直接極限の計算をするのは大変なので、
代わりに小さい値h(上では0.001)を使って近似する。

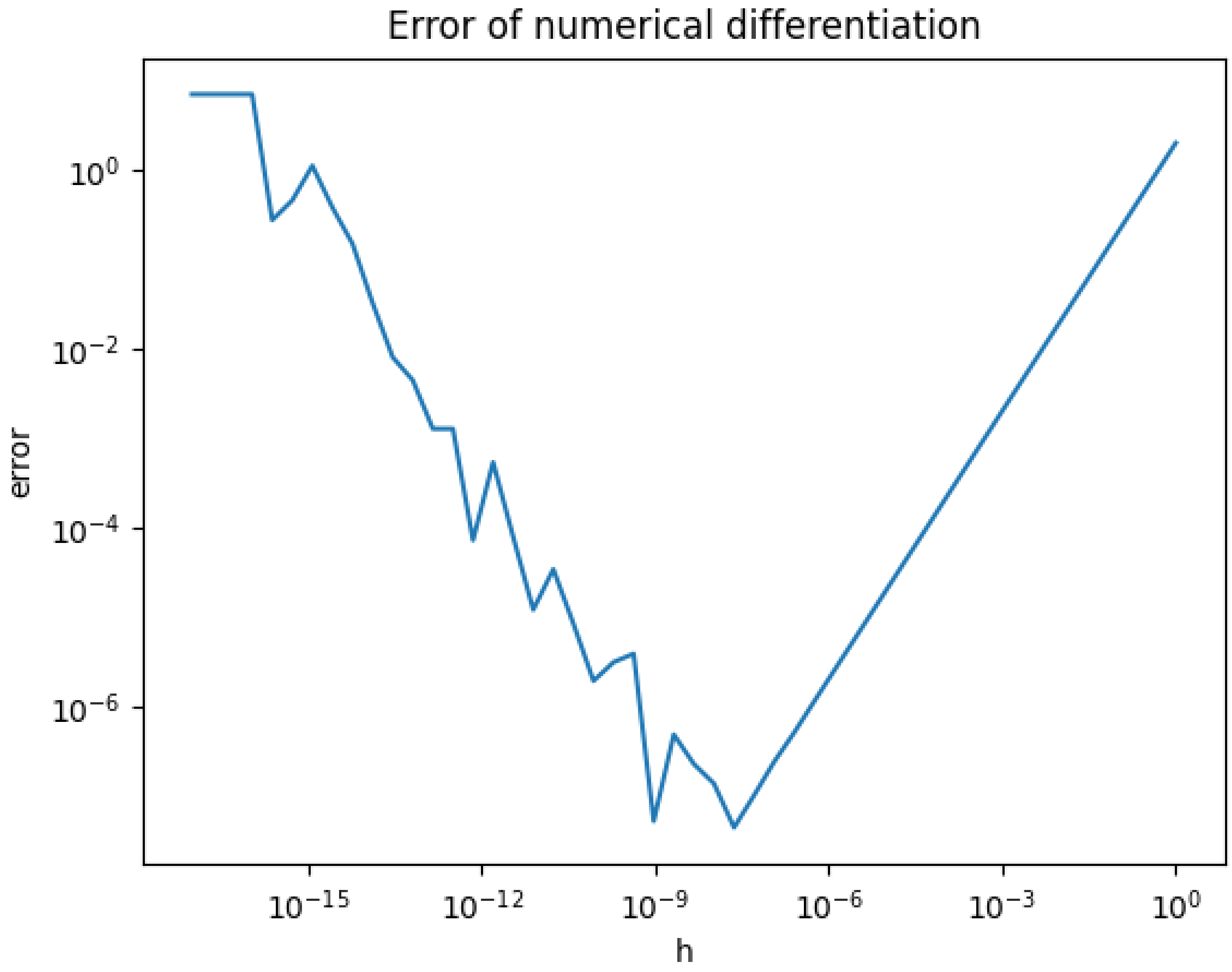
利点

- 実装が簡単
- (計算可能なら)どんな関数でも微分できる

欠点

- 計算量が入力変数の数に比例する
- 誤差が出やすい
  - hをどんどん小さくすればより高精度に計算できるわけではない(浮動小数点数の計算に伴う誤差の影響により一定以上ではむしろ悪化する)

数値微分の誤差

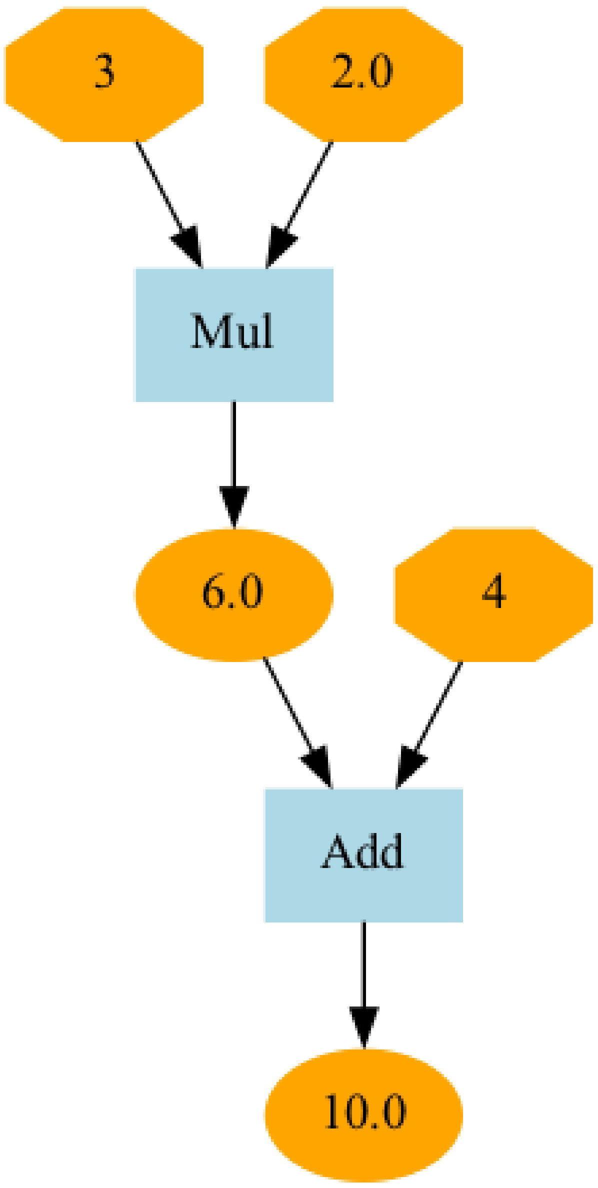


Error of numerical differentiation
error
10^{0}
10^{-2}
10^{-4}
10^{-6}
10^{-15}
10^{-12}
10^{-9}
10^{-6}
10^{-3}
10^{0}
h

数式微分・記号微分

演算は、計算グラフと呼ばれる有向非巡回グラフで表せる。

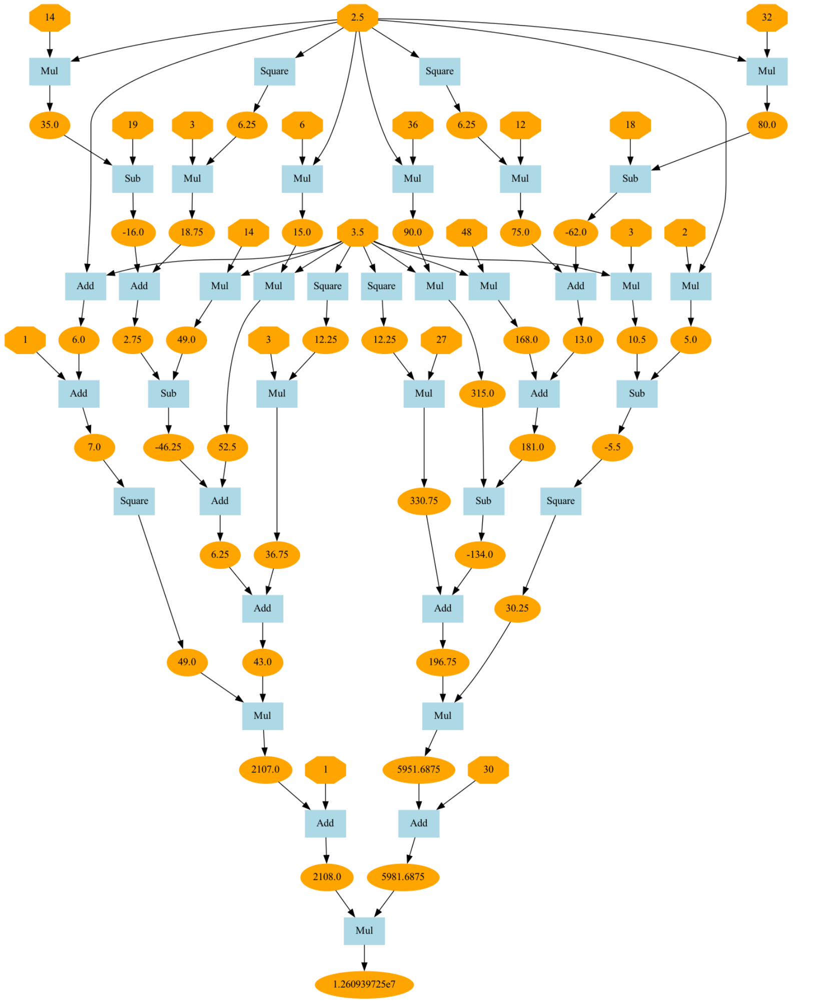
$y = 3x + 4 \rightarrow$



計算グラフ

$$goldstain(x,y) = (1 + (x + y + 1)^2(19 - 14x + 3x^2 - 14y + 6xy + 3y^2))(30 + (2x - 3y)^2(18 - 32x + 12x^2 + 48y - 36xy + 27y^2))$$

$\rightarrow$



計算グラフ

計算グラフに直せば、

式の操作
$\leftrightarrow$　計算グラフの操作

計算グラフに対して適切に変換を行い、
直接導関数を求める手法を**数式微分**や**記号微分**と呼ぶ。

利点

- 一度導関数さえ求められればその後は高速
- (理論的な範囲では)　誤差が出ない

欠点

- **導関数を求めるのは、非常に高コスト**
    - 応用領域では式は容易に非常に複雑になる。(例えば損失はデータ数の分だけ項がある総和として表されるし、尤度関数は標本の数の総積)
    - 例えば積の微分では項が二つに分裂したりする。

⇨ **項の数が「大爆発」して、現実的ではなくなる**

自動微分

- 連鎖律(Chain rule)を利用した微分アルゴリズム

(講義資料にもあるように、自動で微分を求めるアルゴリズムの**一種**が「自動微分」)

連鎖律...

$$\frac{dz}{dx} = \frac{dz}{dy}\frac{dy}{dx}$$

のこと

自動微分

例) $z=(x+y)^2$の$x=3, y=2$における勾配

$a=add(x,y), z=square(a)$である。
(つまり、基本的な関数addと$square$の合成である)

$$\frac{\partial z}{\partial x} = \frac{\partial z}{\partial z}\frac{\partial z}{\partial a}\frac{\partial a}{\partial x} = 1 \times 2a \times 1 = 10$$

$$\frac{\partial z}{\partial y} = \frac{\partial z}{\partial z}\frac{\partial z}{\partial a}\frac{\partial a}{\partial y} = 1 \times 2a \times 1 = 10$$

とすれば計算できる。

自動微分

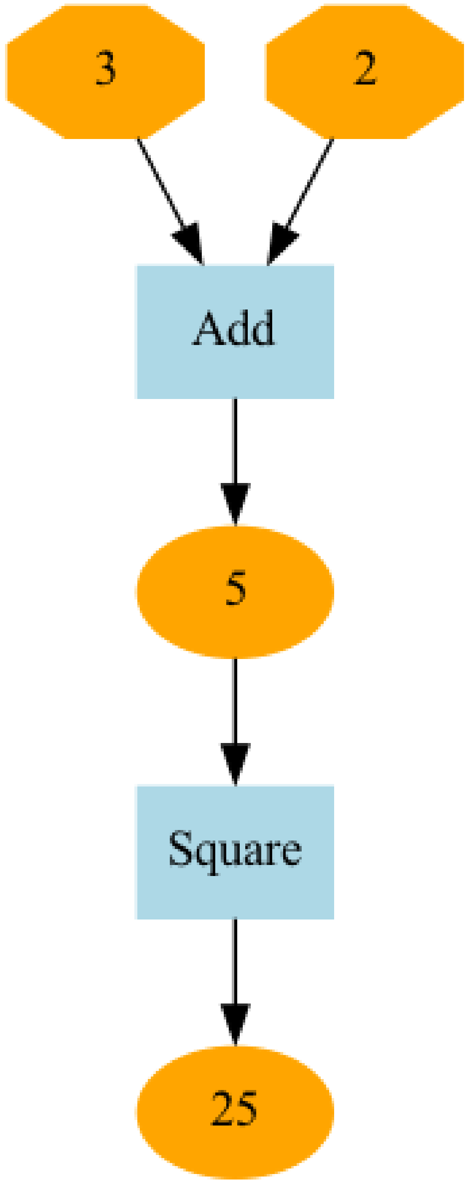
ここで使ったのは...

$\frac{\partial z}{\partial a} = 2a, \frac{\partial a}{\partial y} = 1$という知識

これらは、基本的な関数の微分
⇨ **基本的な関数の導関数さえ定義しておけばこれらの関数の組み合わせのどんな複雑な関数でも微分できる。**

計算グラフとの対応

1. (25)のノードに注目して。勾配を1で初期化する。
2. Squareのノードに進む。
3. Squareの微分は$(x^2)' = 2x$。入力は5なので、$2 \times 5 = 10$を勾配$(= 1)$にかける
4. Addのノードに進む。
5. Addの微分はx, yどちらについても1なので、x, yの勾配は$10 \times 1 = 10$となる。



自動微分

$$\frac{\partial z}{\partial x} = \underline{\frac{\partial z}{\partial z}\frac{\partial z}{\partial a}}\frac{\partial a}{\partial x} = 1 \times 2a \times 1 = 10$$

$$\frac{\partial z}{\partial y} = \underline{\frac{\partial z}{\partial z}\frac{\partial z}{\partial a}}\frac{\partial a}{\partial y} = 1 \times 2a \times 1 = 10$$

共通部の計算は共通化できていた。(初期化部分とSquareの微分に対応)

そして、計算は通常の計算とグラフを逆に辿る(**逆伝播(back propagation)**)の二回で済んだ！

利点

- 計算の効率が非常に良い
  - 共通計算は勝手に共通化される
  - 計算回数は出力変数の数に比例する。
    ⇨ 機械学習は、大量のパラメータを変数として出力は損失(一つの実数)になることが多いので、**圧倒的に効率がいい**

PyTorchとの対応

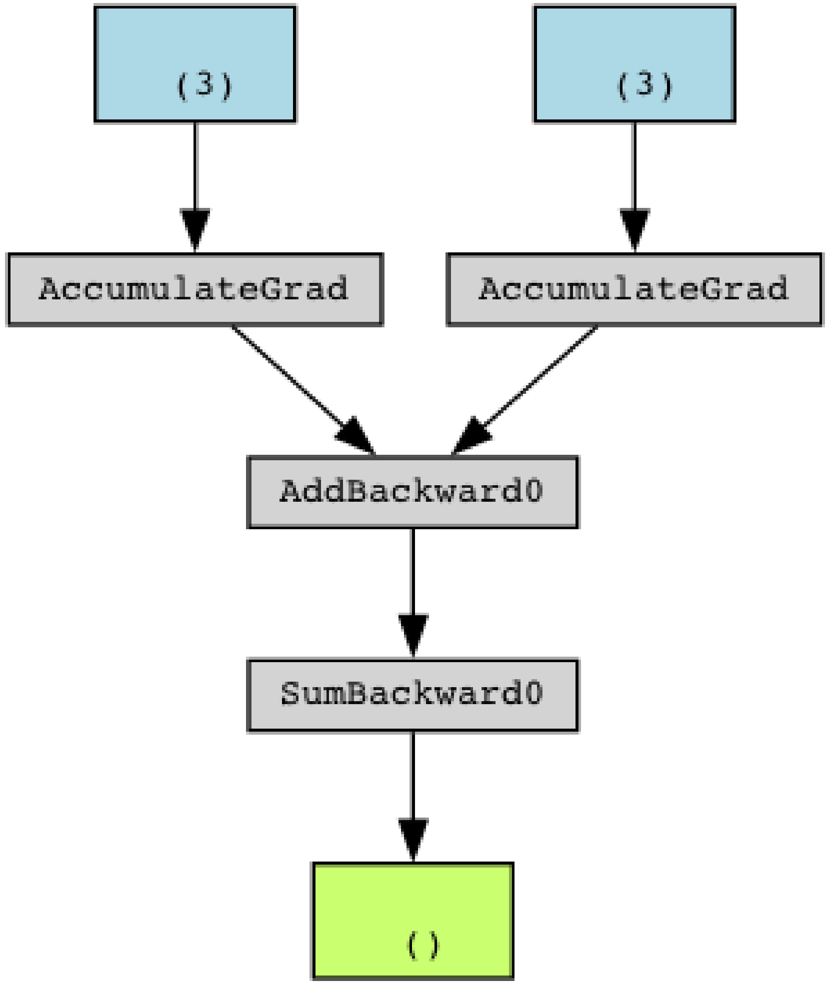
- PyTorchのTensor型に演算をおこなうと、「演算と同時に計算グラフが作られていく」
- `y.backward` を呼び出すことで、yを起点としてグラフを逆に辿り出す

PyTorch上の計算グラフは、torchvizというライブラリを使うと可視化できる。

```
x = torch.tensor([1., 2., 3.], requires_grad=True)
y = torch.sin(torch.sum(x) + 2)
make_dot(y)
```

このように演算と同時に計算グラフを構築するスタイルを**define-by-run**と呼び、これに対して
計算グラフを先に構築してから演算を行うスタイルを**define-and-run**と呼びます。
かつては共存していましたが、今では多くのフレームワークが**define-by-run**を主要なスタイルとして採用しています。実行時に計算グラフを構築する方が圧倒的に柔軟性があるからです。

PyTorchの計算グラフの可視化



forwardモードの自動微分

ここまでで説明したのは、グラフを出力から「逆にたどる」自動微分
⇨ 特に**reverseモードの自動微分**などと呼ばれる

逆に、**forwardモードの自動微分**と呼ばれる手法もある

fowardモードの自動微分では、**二重数**と呼ばれる数を使う。

⬇

実数の範囲内では、

$$x^2 = 0 \Leftrightarrow x = 0$$

ここで、新しく

$$\varepsilon^2 = 0$$

なる数 $\varepsilon \neq 0$ を考えて、実数にこれを加えた集合の演算を考える
(虚数を考えたときと同じ展開)

そこで複素数を考えたときと全く同様に、
$z = a + b\varepsilon$という形の数を考える($a, b \in \mathbb{R}$)

このような形で表される数を**二重数**と呼ぶことにする

$\varepsilon^2 = 0$に注意すれば、

$$
\begin{aligned}
(a + b\varepsilon) + (c + d\varepsilon) &= (a + c) + (b + d)\varepsilon \\
(a + b\varepsilon) \times (c + d\varepsilon) &= (ac) + (ad + bc)\varepsilon
\end{aligned}
$$

とするのが良さそう

すると、実係数多項式

$$f(x) = a_n x^n + a_{n-1} x^{n-1} + \cdots + a_1 x + a_0$$

に対して、

$$f(x + b\varepsilon) = f(x) + bf'(x)\varepsilon$$

となることがわかる
やや雑ですが、微分可能ならテイラー展開することで結局実係数多項式にできるので実係数多項式の微分が正しく計算できるのであれば、微分可能な関数全てに対してうまくいきそう

https://en.wikipedia.org/wiki/Automatic_differentiation#Two_types_of_automatic_differentiationから

実際の実装では、reverseモード同様、基本的な関数に対して二重数の演算を定義しておき、その合成として計算。

$$f(x + b\varepsilon) = f(x) + bf'(x)\varepsilon$$

をもう一度見れば、$f(x + b\varepsilon)$の計算それ自体がfのxにおける微分係数を求める演算と対応している！

普通の関数の計算は計算グラフを入力から「順」に辿っていく
(**順伝播, forward propagation**) ことに他ならない

➡ これをforwardモードの自動微分と呼ぶ

発展的話題: 自動で微分を求めるアルゴリズム　まとめ

- 代表的なアプローチとして、数値微分、記号微分、自動微分があった
- 数値微分は誤差が出やすく、記号微分は計算が現実的ではなかった
- 自動微分は高速でとくにreverseモードの自動微分は機械学習において圧倒的に有用であった

もっと興味がある人は

ゼロから作るDeep Learning ❸:
https://www.oreilly.co.jp/books/9784873119069/
や、
https://arxiv.org/abs/1502.05767, https://arxiv.org/abs/2002.03794
などを読んでみてください。