Chainer 入門

2018/6/27

M2 松村雪桜

Chainerとは

- ニューラルネットワーク実装のためのライブラリ
- 書きやすい、読みやすい
- pip install chainer でインストール

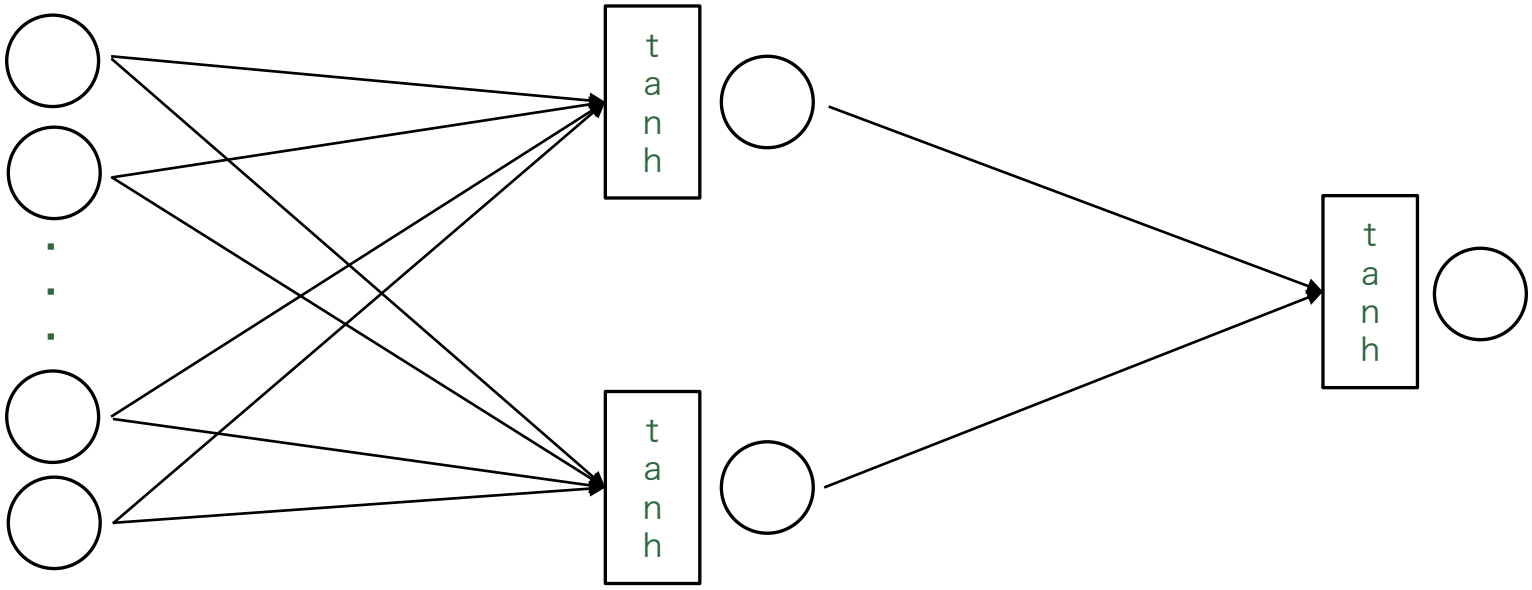
Chainerの基本的な機能

- Variable
- links
- functions
- optimizers
- serializers

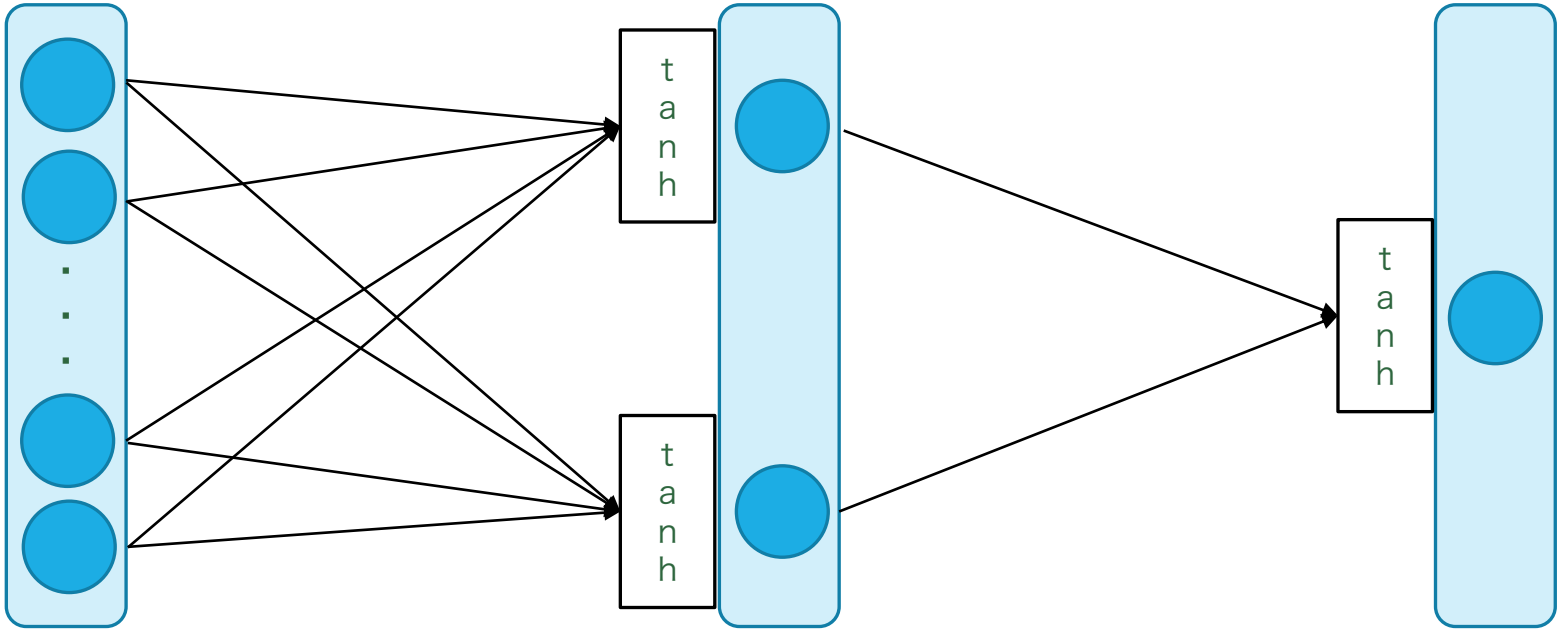
Chainerの基本的な機能

- Variable
- links
- functions
- optimizers
- serializers

ニューラルネット



Variable



Variable

- 変数（ベクトルや行列）に対応するオブジェクト

```
import numpy
from chainer import Variable

x = Variable(numpy.array([[1, 2, 3], [4, 5, 6]], dtype = numpy.float32))
y = x ** 2 + 5 * x
y.grad = numpy.ones((2, 3), dtype=numpy.float32)   #勾配の次元を定義
y.backward()   #微分の計算　(y’ = 2 * x + 5)
print(x.grad)   #xでの勾配
```

Variable

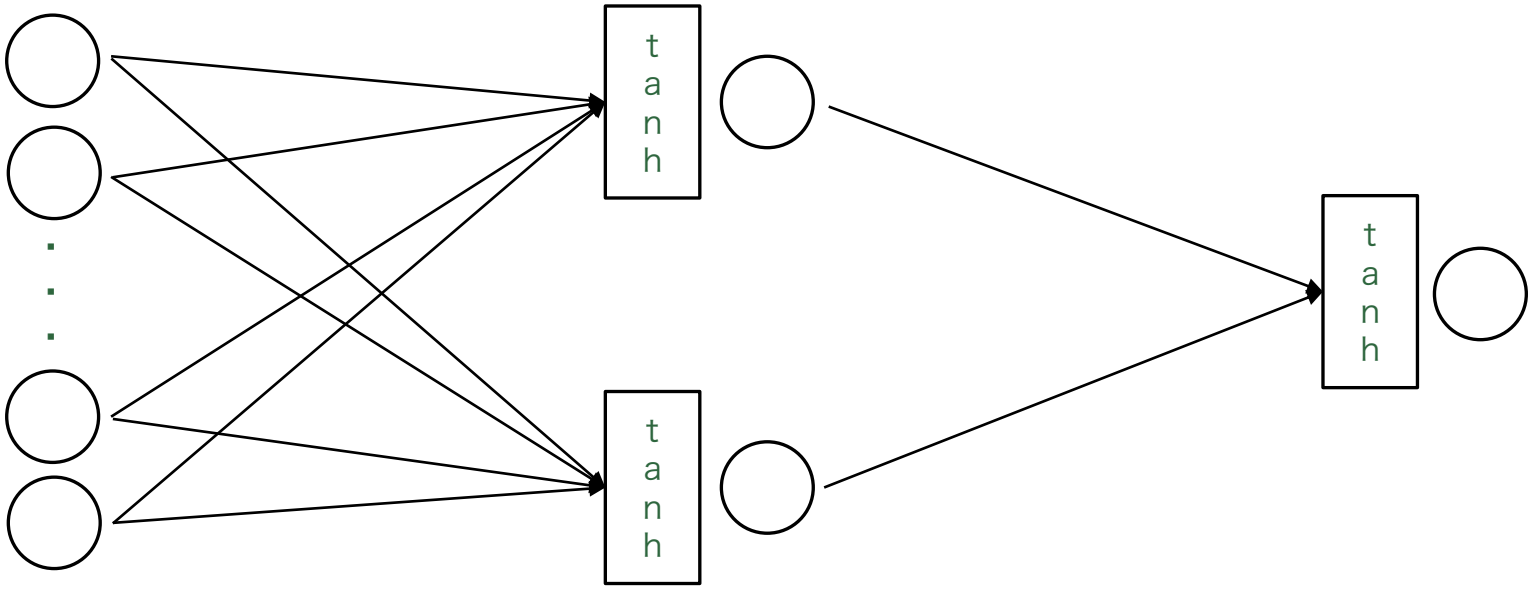
- arrayをVariableにすることで微分の計算が可能に

- Variableとして使用するarrayのdtypeは
  float32またはint32である必要があることに注意
  (おそらく指定しないと32ではなく64になる)

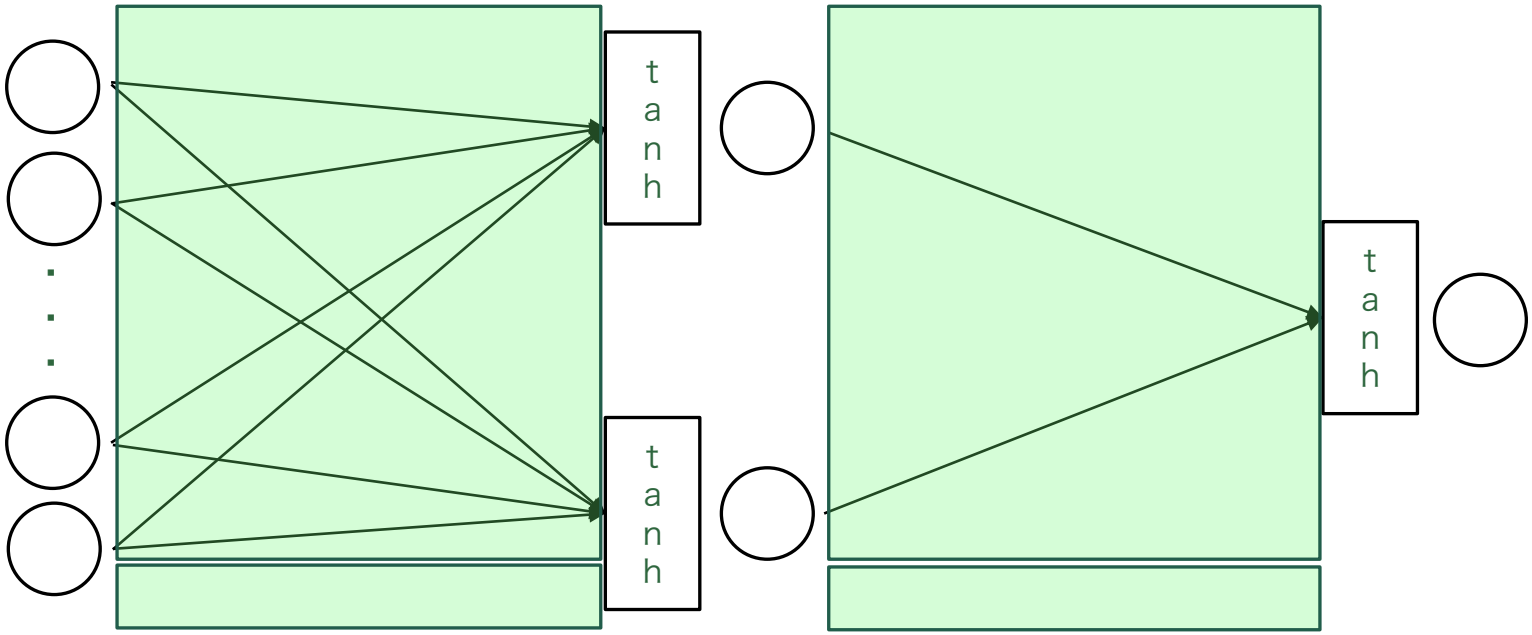
Chainerの基本的な機能

- Variable
- links
- functions
- optimizers
- serializers

ニューラルネット



links



links

- 層（重み行列）に対応するオブジェクト

```
import numpy
from chainer import Variable, links

h = links.Linear(4, 2)    #2×4の重み行列を作成
print(h.W.data)    #hの重み行列
print(h.b.data)    #hのバイアス

x = Variable(numpy.array([[1, 2, 3, 4]]).astype(numpy.float32))
y = h(x)
print(y.data)    #重み行列を通してできた新しいVariable
```

links

- linksで重み行列を定義することで
  重みとバイアスの初期値も定義される

- ほとんどのlinksに入力するVariableは
  バッチサイズ×データの次元数の行列
  である必要がある(バッチサイズ=1でも)

links

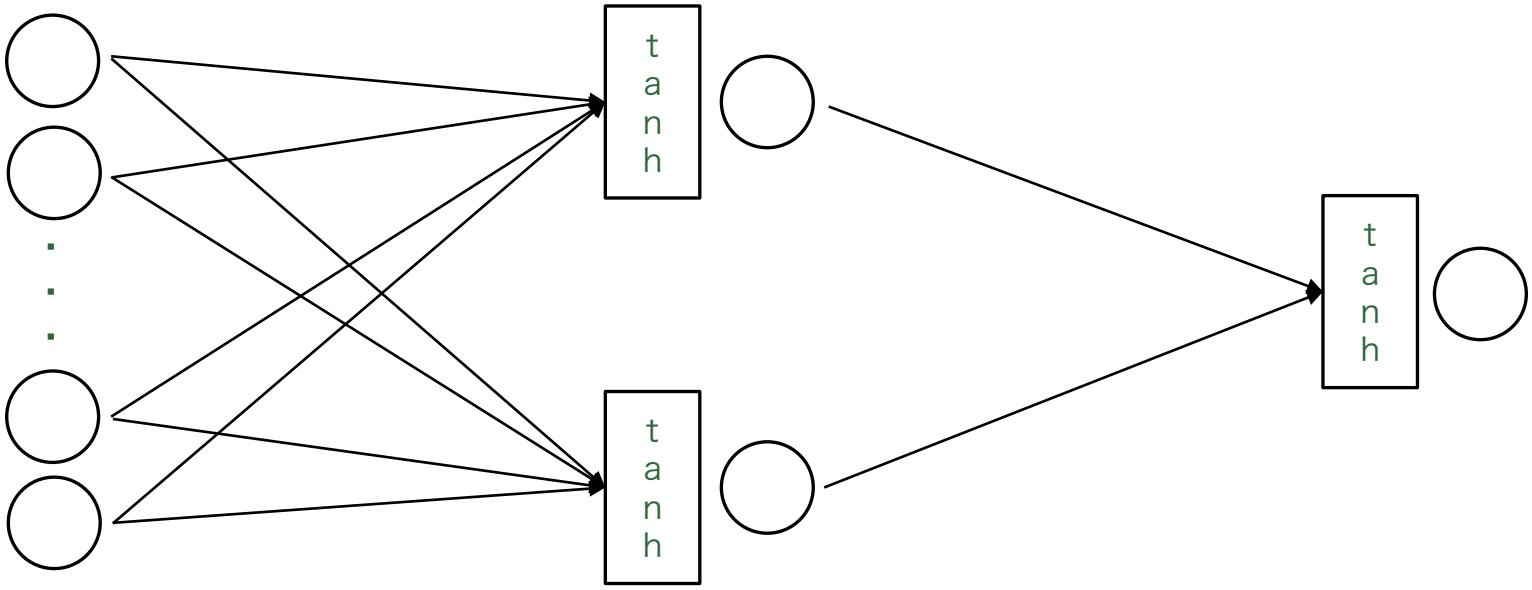
- 用途によっていろいろ用意されている

- FFN → Linear
- RNN → LSTM, GRU
- CNN → Convolution2D
- Word Embedding → EmbedID

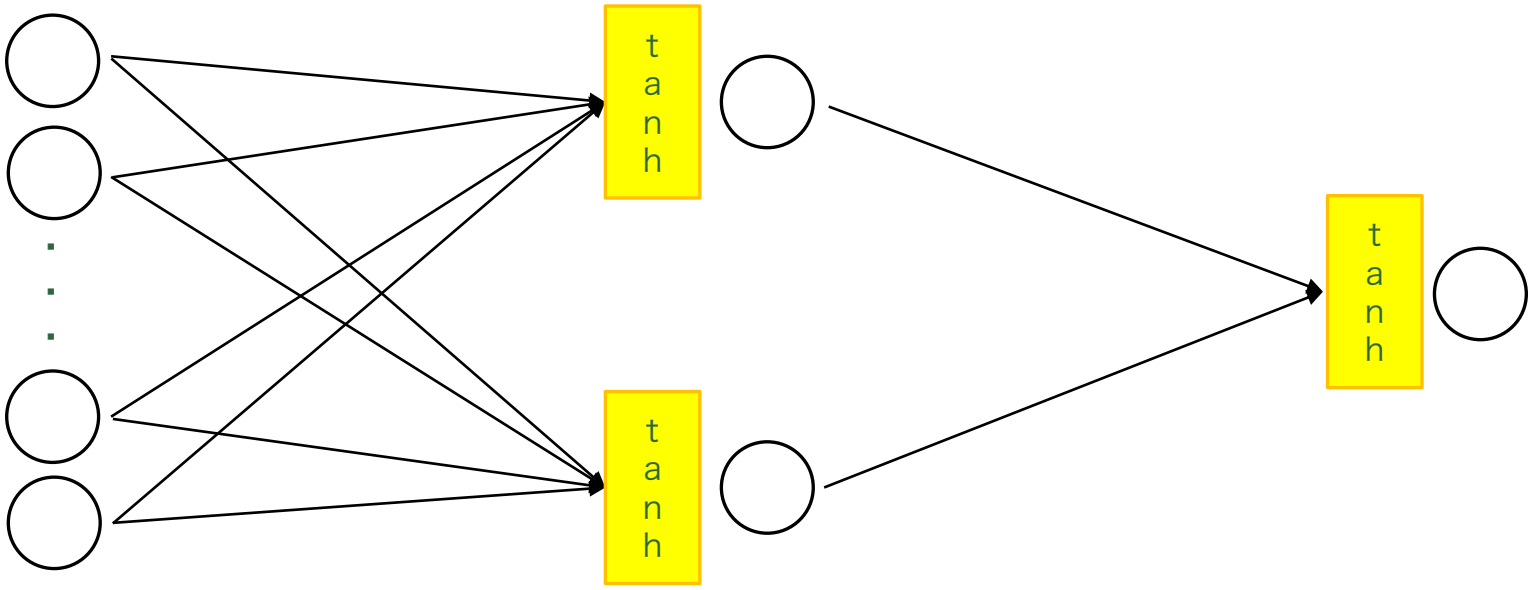
Chainerの基本的な機能

- Variable
- links
- functions
- optimizers
- serializers

ニューラルネット



functions



functions

- Variableに対応した関数

```
import numpy
from chainer import Variable, functions

x = Variable(numpy.array([[1, 2, 3], [4, 5, 6]], dtype = numpy.float32))
y = functions.tanh(x)
print(y.data)
y.grad = numpy.ones((2, 3), dtype=numpy.float32)   #勾配の次元を定義
y.backward()   #微分の計算　(y’ = 1 – y ** 2)
print(x.grad)   #xでの勾配
```

functions

- functionsは活性化関数や損失関数、行列操作関数など、様々な関数が提供されている
- functionsの関数もbackwardで微分が可能

functions

- 活性化関数 → tanh, sigmoid
- 損失関数 → mean_squared_error
  softmax_cross_entropy
- 行列操作関数 → stack, concat

Chainerの基本的な機能

- Variable
- links
- functions
- optimizers
- serializers

optimizers

- 最適化(重みの更新)アルゴリズムに対応する

```
from chainer import optimizers

model = NeuralNetwork(input_size)    #モデルの生成
opt = optimizers.SGD(lr=λ)    #最適化アルゴリズムの定義
opt.setup(model)    #アルゴリズムにモデルをセット
```

optimizers

- 確率的勾配降下法（SGD）をはじめ、AdamやAdaGradなど様々な最適化アルゴリズムが提供されている
- add_hookを使うと重み更新時の正則化

Chainerの基本的な機能

- Variable
- links
- functions
- optimizers
- serializers

serializers

- モデルのセーブ・ロードに関するモジュール

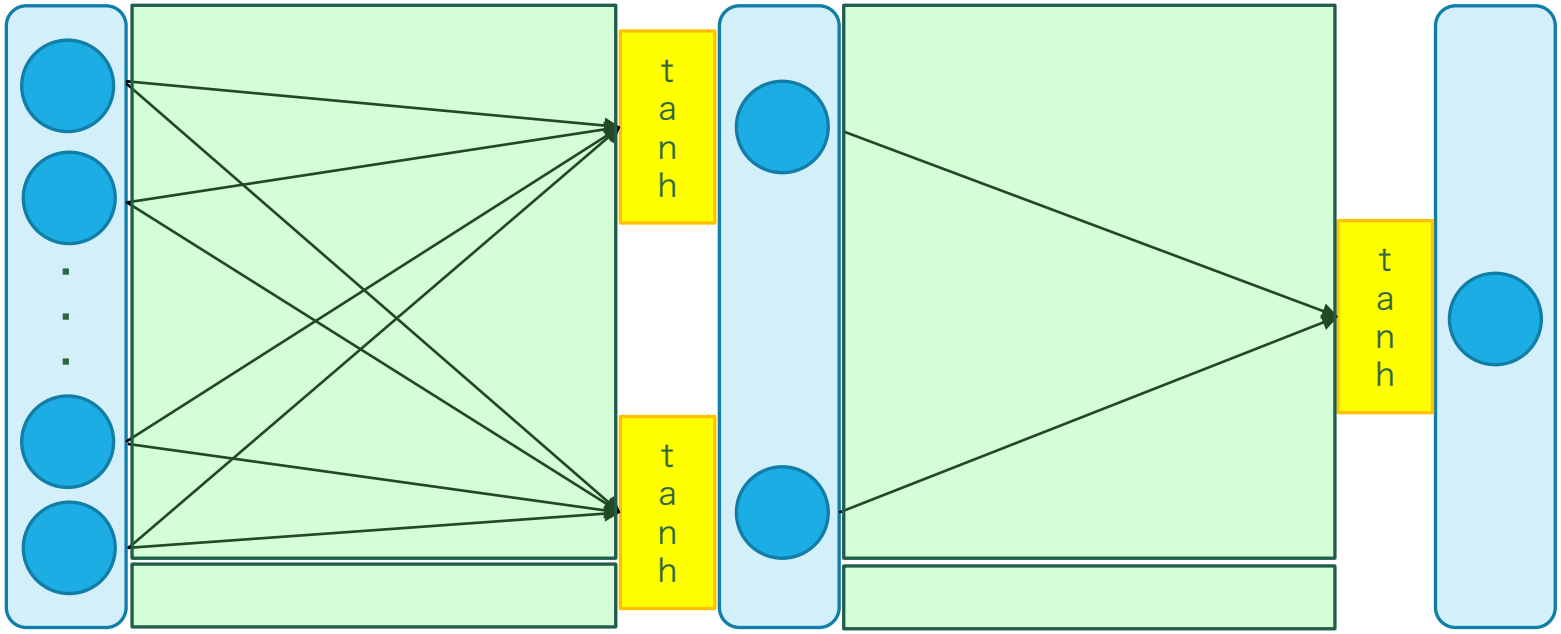
```
from chainer import serializers

serializers.save_npz(model_path, model)     #モデルの保存
serializers.load_npz(model_path, model)     #モデルをロード
```

- pickleのchainer版みたいな感じ
- フォーマットはNPZとHDF5の2つ

実装例

- NLPチュートリアルの課題をChainerで実装
- 文書分類（1 or -1）
  - モデルの定義
  - 最適化アルゴリズムの定義
  - 学習
  - テスト

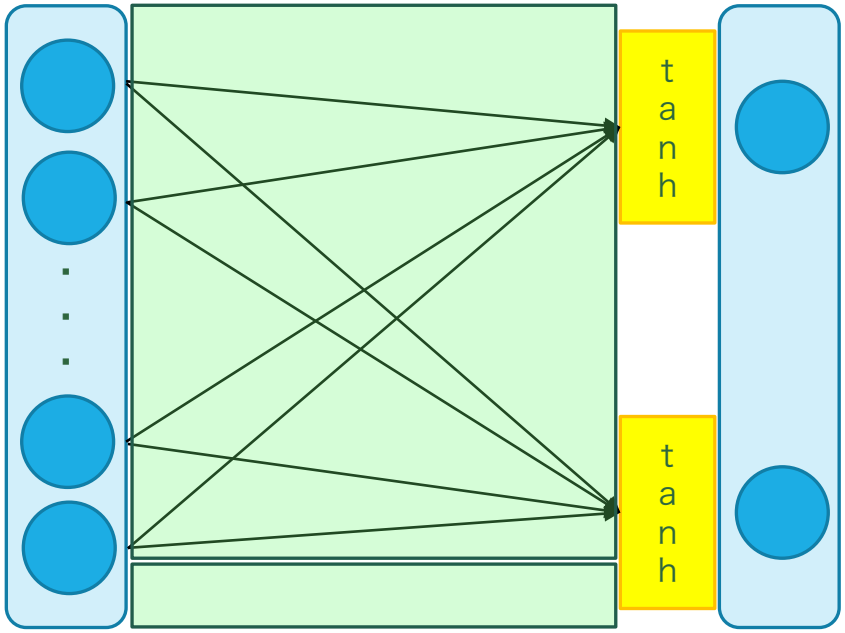
モデルの定義



モデルの定義

```
class NeuralNetwork(Chain):   #Chainクラスを継承
    def __init__(self, input_size):
        super(NeuralNetwork, self).__init__(
            link1 = links.Linear(input_size, 2),   #入力層→中間層
            link2 = links.Linear(2, 1),    #中間層→出力層
        )
    def __call__(self, x, y):
        y_predict = self.forward(x)    #順伝搬でyを予測
        return functions.mean_squared_error(y_predict, y) / 2   #損失関数
    def forward(self, x):
        hidden = functions.tanh(self.link1(x))   #中間層を計算
        y_predict = functions.tanh(self.link2(hidden))    #出力層を計算
       return y_predict

model = NeuralNetwork(input_size)    #モデルの生成
```

順伝搬



φ_0
Vector
(Vocab)

w_0
Matrix
(2, Vocab)

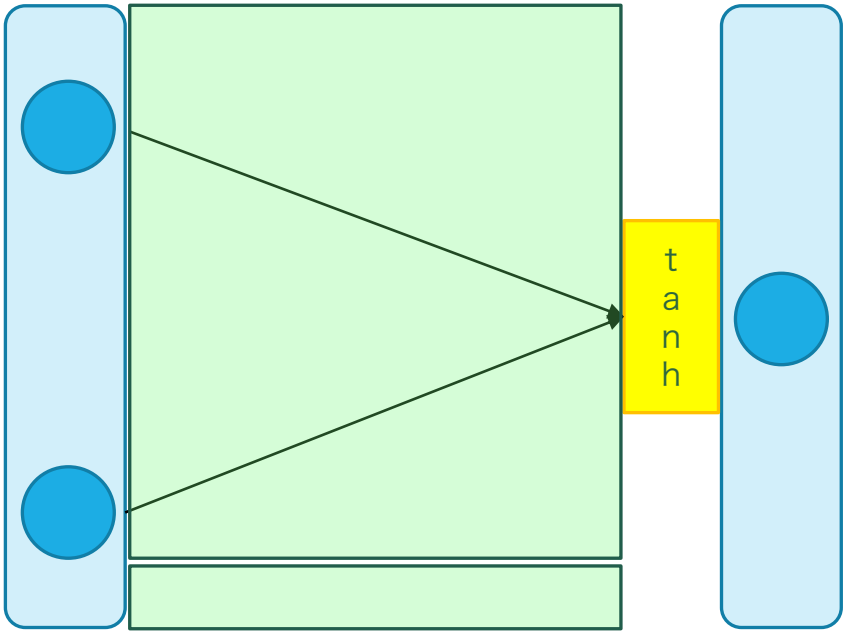
b_0
Vector
(2)

φ_1
Vector
(2)

$$\varphi_1 = \tanh(w_0\ \varphi_0 + b_0)$$

順伝搬

$$\varphi_2 = \tanh(w_1\ \varphi_1 + b_1)$$



モデルの定義

- Chainクラスを継承したクラス内は繋がった一つのネットワークとして認識される
- 順伝搬の定義が簡単にできる

最適化アルゴリズムの定義

- 確率的勾配降下法（SGD）で最適化

```
model = NeuralNetwork(input_size)    #モデルの生成
opt = optimizers.SGD(lr=λ)    #最適化アルゴリズムの定義
opt.setup(model)    #アルゴリズムにモデルをセット
```

- この3行で最適化アルゴリズムが設定可能

学習

```
for i in range(epoch):
        accum_loss = 0    #そのepochでの誤差の合計を初期化
        for phi_0, y in feat_lab:
            model.zerograds()    #勾配の初期化
            loss = model(Variable(phi_0), Variable(y))    #誤差の計算
            accum_loss += loss.data    #計算した誤差を蓄積していく
            loss.backward()    #逆伝搬、勾配の計算
            opt.update()    #重みの更新
        print(accum_loss)  #そのepochでの誤差の合計
serializers.save_npz(model_path, model)    #モデルの保存
```

逆伝搬

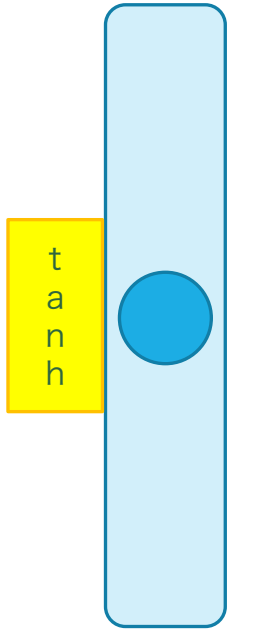
$$\mathrm{err} = \frac{(\varphi_2 - y)^2}{2}$$

$$\delta_2 = \frac{\mathrm{derr}}{\mathrm{d}\varphi_2} = \varphi_2 - y$$

$$\delta'_2 = \frac{\mathrm{derr}}{\mathrm{d}(w_1 \varphi_1 + b_1)}$$

$$= \frac{\mathrm{derr}}{\mathrm{d}\varphi_2} \frac{\mathrm{d}\varphi_2}{\mathrm{d}(w_1 \varphi_1 + b_1)}$$

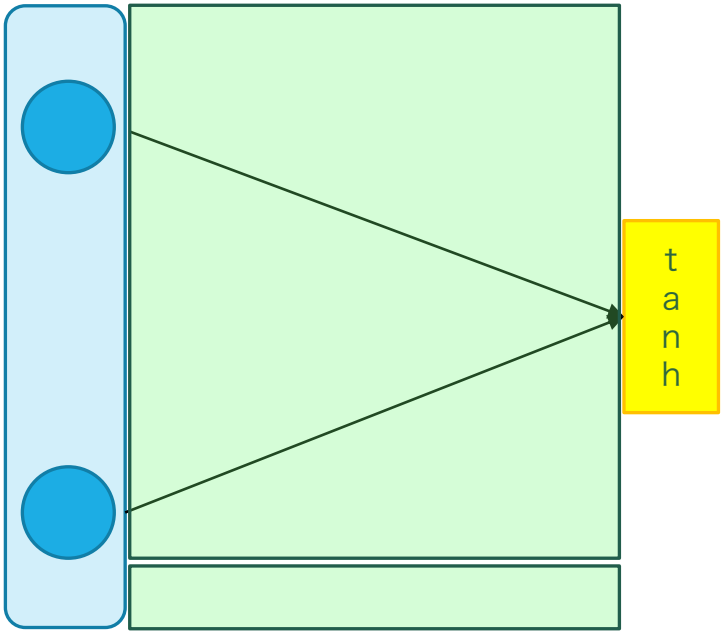
$$= \delta_2 (1 - {\varphi_2}^2)$$



δ'_2 Vector (1)　　δ_2 Vector (1)

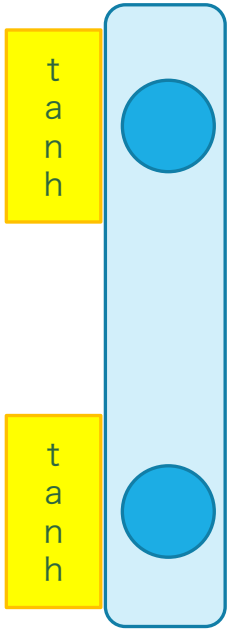
逆伝搬

$$\delta_1 = \frac{\mathrm{derr}}{\mathrm{d}\varphi_1}$$
$$= \frac{\mathrm{derr}}{\mathrm{d}(w_1\,\varphi_1 + b_1)} \frac{\mathrm{d}(w_1\,\varphi_1 + b_1)}{\mathrm{d}\varphi_1}$$
$$= \delta'_2\, w_1$$



δ_1	w_1	b_1	δ'_2
Vector	Matrix	Vector	Vector
(2)	(1, 2)	(1)	(1)

逆伝搬

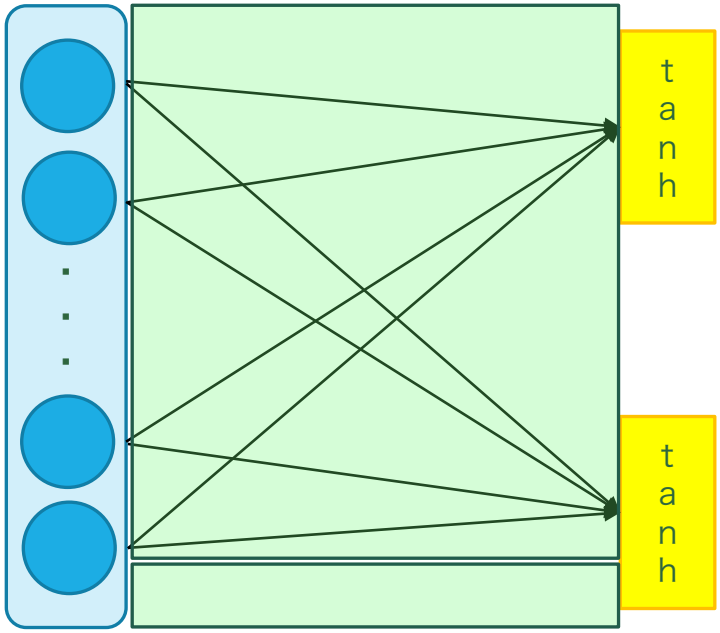


δ'_1 δ_1

Vector (2) Vector (2)

$$\delta'_1 = \frac{\mathrm{derr}}{\mathrm{d}(w_0\,\varphi_0 + b_0)}$$
$$= \frac{\mathrm{derr}}{\mathrm{d}\varphi_1}\frac{\mathrm{d}\varphi_1}{\mathrm{d}(w_0\,\varphi_0 + b_0)}$$
$$= \delta_1(1 - {\varphi_1}^2)$$

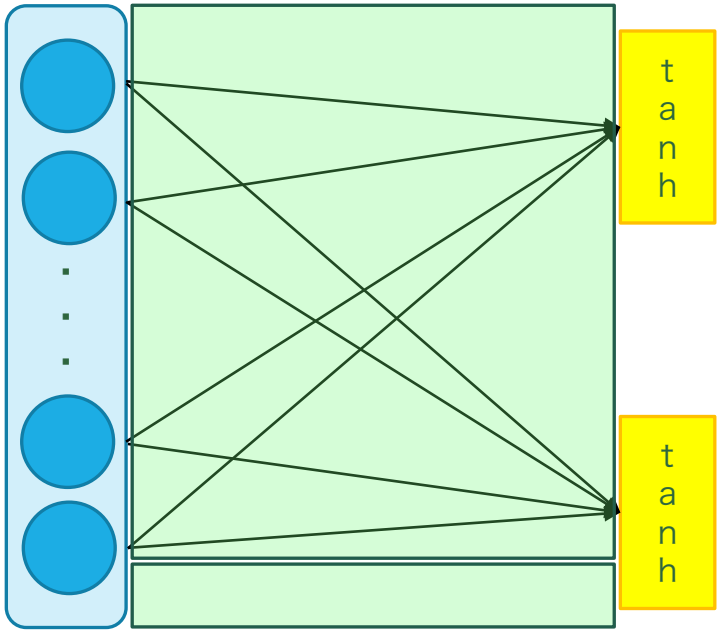
逆伝搬



δ_0	w_0	b_0	δ'_1
Vector	Matrix	Vector	Vector
(Vocab)	(2, Vocab)	(2)	(2)

$$\delta_0 = \frac{\mathrm{derr}}{\mathrm{d}\varphi_0}$$
$$= \frac{\mathrm{derr}}{\mathrm{d}(w_0\,\varphi_0 + b_0)} \frac{\mathrm{d}(w_0\,\varphi_0 + b_0)}{\mathrm{d}\varphi_0}$$
$$= \delta'_1\, w_0$$

重みの更新

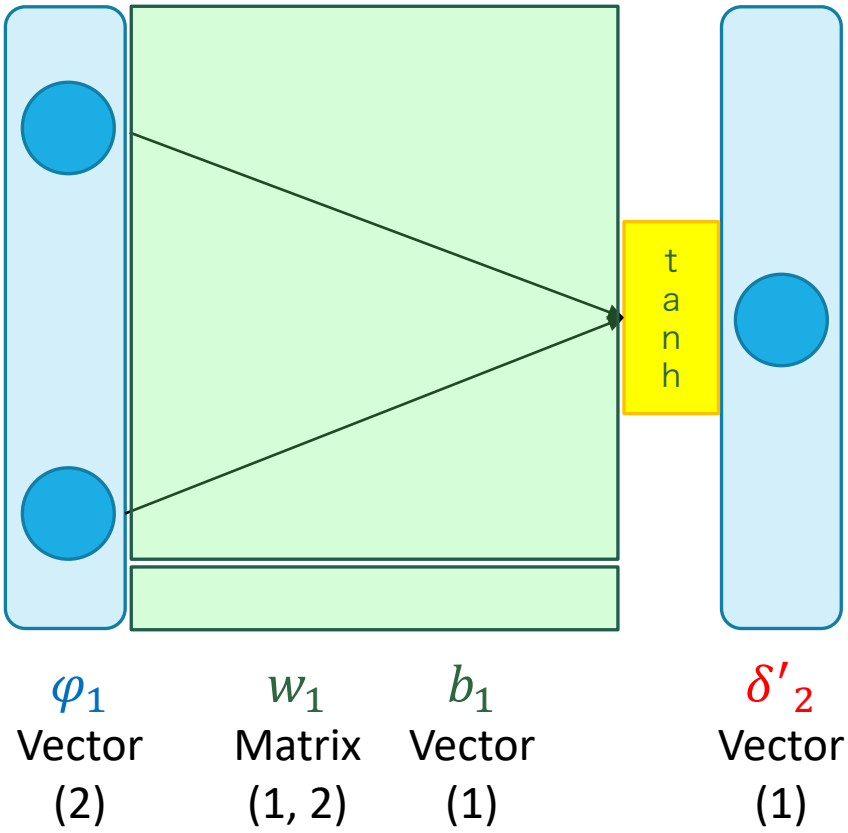


φ_0	w_0	b_0	δ'_1
Vector	Matrix	Vector	Vector
(Vocab)	(2, Vocab)	(2)	(2)

$$
\begin{aligned}
w_0 \mathrel{-}= \lambda \frac{\mathrm{derr}}{\mathrm{d}w_0} \\
&= \lambda\, \delta'_1 \frac{\mathrm{d}(w_0\, \varphi_0 + b_0)}{\mathrm{d}w_0} \\
&= \lambda\, \delta'_1\, \varphi_0 \\
b_0 \mathrel{-}= \lambda \frac{\mathrm{derr}}{\mathrm{d}b_0} \\
&= \lambda\, \delta'_1 \frac{\mathrm{d}(w_0\, \varphi_0 + b_0)}{\mathrm{d}b_0} \\
&= \lambda\, \delta'_1
\end{aligned}
$$

重みの更新

$$w_1 \mathrel{-}= \lambda \frac{\mathrm{derr}}{\mathrm{d}w_1}$$
$$= \lambda\, \delta'_2 \frac{\mathrm{d}(w_1\ \varphi_1 + b_1)}{\mathrm{d}w_1}$$
$$= \lambda\, \delta'_2\ \varphi_1$$
$$b_1 \mathrel{-}= \lambda \frac{\mathrm{derr}}{\mathrm{d}b_1}$$
$$= \lambda\, \delta'_2 \frac{\mathrm{d}(w_1\ \varphi_1 + b_1)}{\mathrm{d}b_1}$$
$$= \lambda\, \delta'_2$$



学習

- 複雑だった逆伝搬の計算、重みの更新がそれぞれ1行ずつで記述可能

```
loss.backward()
optimizer.update()
```

テスト

```
model = NeuralNetwork(input_size)
serializers.load_npz(model_path, model)    #モデルをロード
    with open(fin_path, "r") as fin:
        for x in fin:
            phi_0 = create_features(x)
            score = model.forward(Variable(phi_0))    #順伝搬でscoreを計算
            y = 1 if score.data >= 0 else -1    #scoreからラベルを予測
```

サンプルコード

- NLPチュートリアルのChainer版サンプルコードは以下にあります

https://github.com/yukio326/NLPtutorial/blob/master/tutorial07/train-chainer-nn.py

https://github.com/yukio326/NLPtutorial/blob/master/tutorial07/test-chainer-nn.py

GPUで使いたい

- GPUで使う場合はnumpyではなくcupy
- cupyもpip install cupyでインストール
- モデルの定義後に以下2行を記述

```
cuda.get_device(GPU番号).use()
model.to_gpu()
```

まとめ

- Chainerを使うと複雑な構造が簡単に書ける
- 特に勾配の計算を記述しなくて良いのが楽
- 今回紹介したのはほんの一部で
  他にも便利な関数・機能がたくさんある

余談

- よく使いそうなネットワーク構造はクラスとしてサーバー上に実装している
- /clwork/yukio/YukioNMT/src/networks.py
- 使いたい人、参考にしたい人はどうぞ