RNN を用いた c-VEP のニューラルデコーディング

佐藤純一

鷲沢研究室

2016 年 5 月 16 日

Outline

前回の復習

- ニューラルネットワークの歴史，基本
- 時系列データを扱うための Recurrent Neural Network の紹介

Recurrent Neural Network (RNN)

- $w_{ji}^{(in)}$：入力層の重み
- $w_{jj'}$：中間層から中間層への帰還路の重み
- $w_{kj}^{(out)}$：出力層への重み

順伝搬計算

$$u_j^t = \sum_i w_{ji}^{(in)} x_i^t + \sum_{j'} w_{jj'} z_{j'}^{t-1} \quad (1)$$

$$z_j^t = f(u_j^t) \quad (2)$$

$$v_k^t = \sum_j w_{kj}^{(out)} z_j^t \quad (3)$$

$$y_k^t = f^{out}(v_k^t) \quad (4)$$

Recurrent Neural Network (RNN)

逆伝搬計算

誤差 $\delta_j \equiv \frac{\partial E}{\partial u_j}$ は逆伝搬によって求めることができる

$$\delta_j^t = \left(\sum_k w_{kj}^{(our)} \delta_k^{out,t} + \sum_{j'} w_{j'j} \delta_{j'}^{t+1} \right) f'(u_j^t) \tag{5}$$

勾配

$$\frac{\partial E}{\partial w_{ji}^{in}} = \sum_{t=1}^{T} \delta_j^t x_i^t \tag{6}$$

$$\frac{\partial E}{\partial w_{jj'}} = \sum_{t=1}^{T} \delta_j^t z_i^{t-1} \tag{7}$$

$$\frac{\partial E}{\partial w_{kj}^{out}} = \sum_{t=1}^{T} \delta_j^t z_i^t \tag{8}$$

(9)

RNN の問題点

- RNN は理論的に過去の全入力が反映されて出力される
- 実際には，たかだか 10 時刻分程度であるといわれる [1]
  ⇒ RNN の勾配消失問題
- 長期的な記憶を実現するのが難しい

勾配消失問題

総数の多いネットワークでは層をたどるに連れて

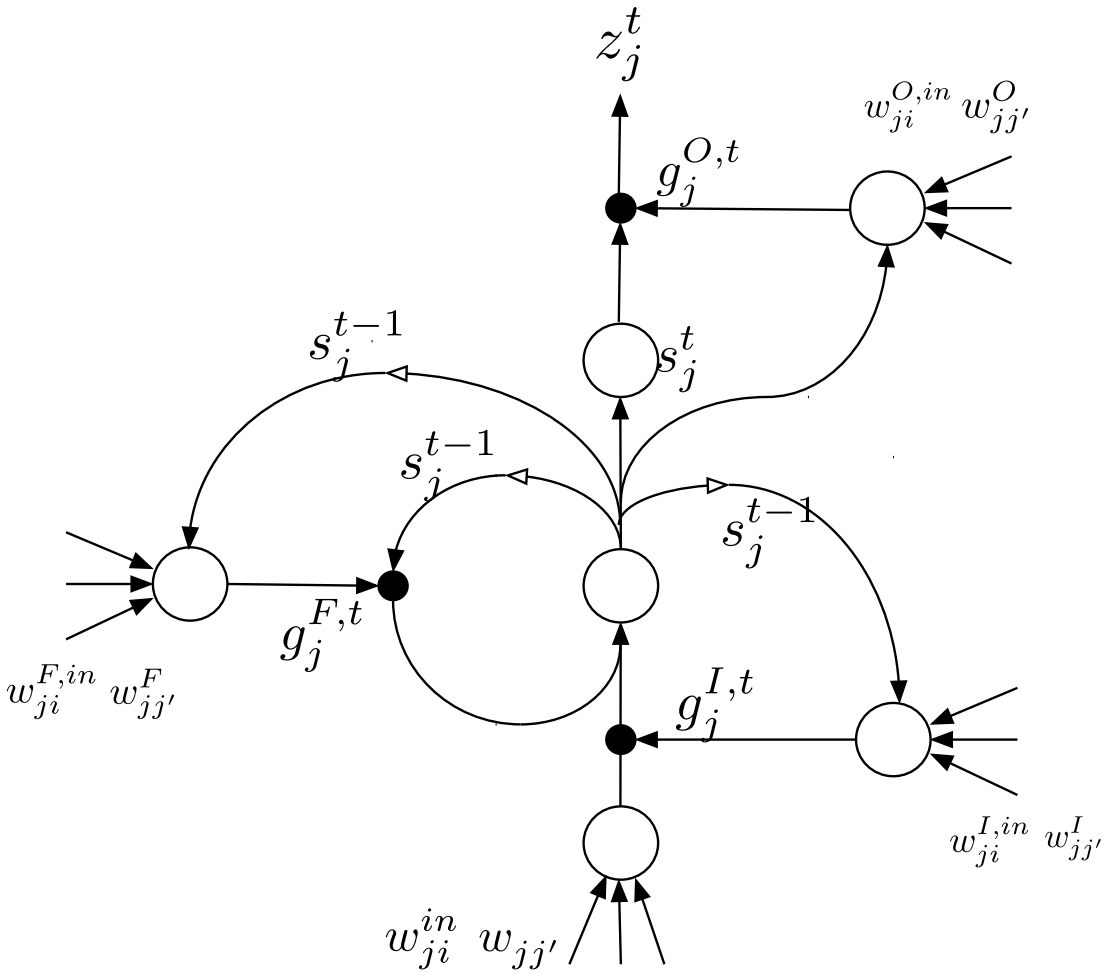
- 勾配の値が爆発的に大きくなる
- 0 に消失する

という性質がある
層が深いネットワークや RNN で問題になる

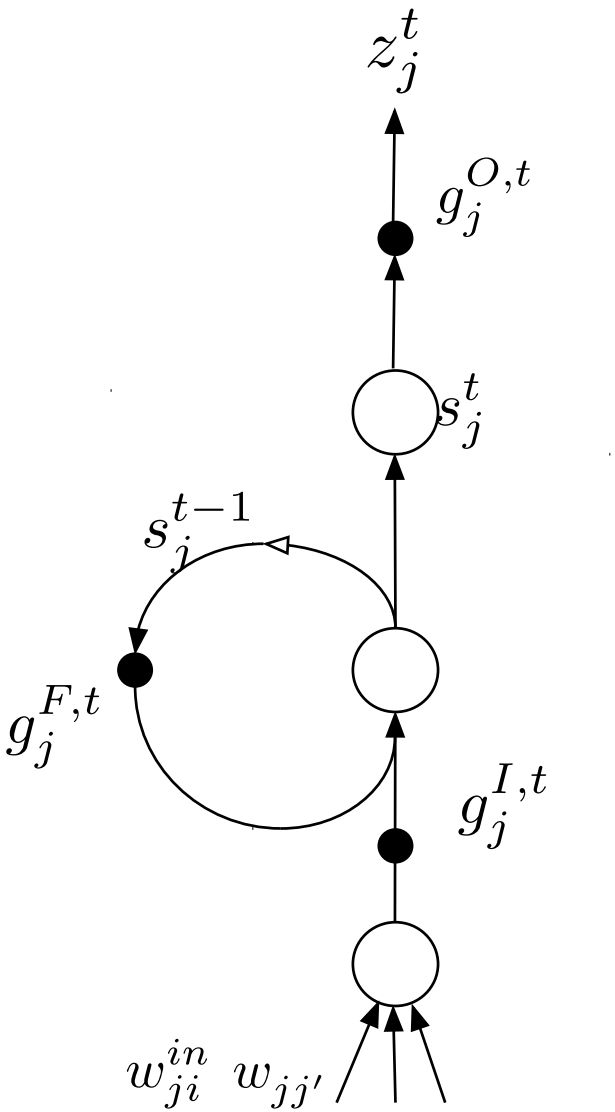
[1] [1] 講談社 深層学習 岡谷貴之

Long Short Term Memory (LSTM)

- RNN で長期にわたる記憶を実現する方法のうち最も成功している手法
- 中間層の各ユニットを下図のメモリユニットで置き換えている



LSTM : 基本



- 中央のユニットはメモリセルと呼ばれ状態 s_j^t を保持する
- 1 時刻を経てメモリセル自身に帰還する
- 入力，帰還，出力にそれぞれゲート (図の黒丸) が存在する
  - $g_j^{F,t} \in [0,1]$: 忘却ゲート
  - $g_j^{I,t} \in [0,1]$: 入力ゲート
  - $g_j^{O,t} \in [0,1]$: 出力ゲート
- $g_j^{F,t}$ が 1 に近いほど状態は記憶され，0 に近いと忘却される
- 忘却ゲートを 1，入力を 0 にするとメモリセルの状態は記録され続ける

LSTM : 順伝搬計算 (1)

- 順伝搬計算は以下の通りに計算される

$$s_j^t = g_j^{F,t} s_j^{t-1} + g_j^{I,t} f(u_j^t) \tag{10}$$

$$u_j^t = \sum_i w_{ji}^{in} x_i^t + \sum_{j'} w_{jj'} z_{j'}^{t-1} \tag{11}$$

- u_j^t は通常の RNN と同様
- 忘却ゲートと入力ゲートはそれぞれ次式で計算される.

$$g_j^{F,t} = f(u_j^{F,t}) = f\left(\sum_i w^{F,in} x_i^t + \sum_{j'} w_{jj'}^F z_{j'}^{t-1} + w_j^F s_j^{t-1}\right) \tag{12}$$

$$g_j^{I,t} = f(u_j^{I,t}) = f\left(\sum_i w^{I,in} x_i^t + \sum_{j'} w_{jj'}^I z_{j'}^{t-1} + w_j^I s_j^{t-1}\right) \tag{13}$$

(14)

LSTM : 順伝搬計算 (2)

- 出力は以下のように計算される

$$z_j^t = g_j^{O,t} f(s_j^t) \tag{15}$$

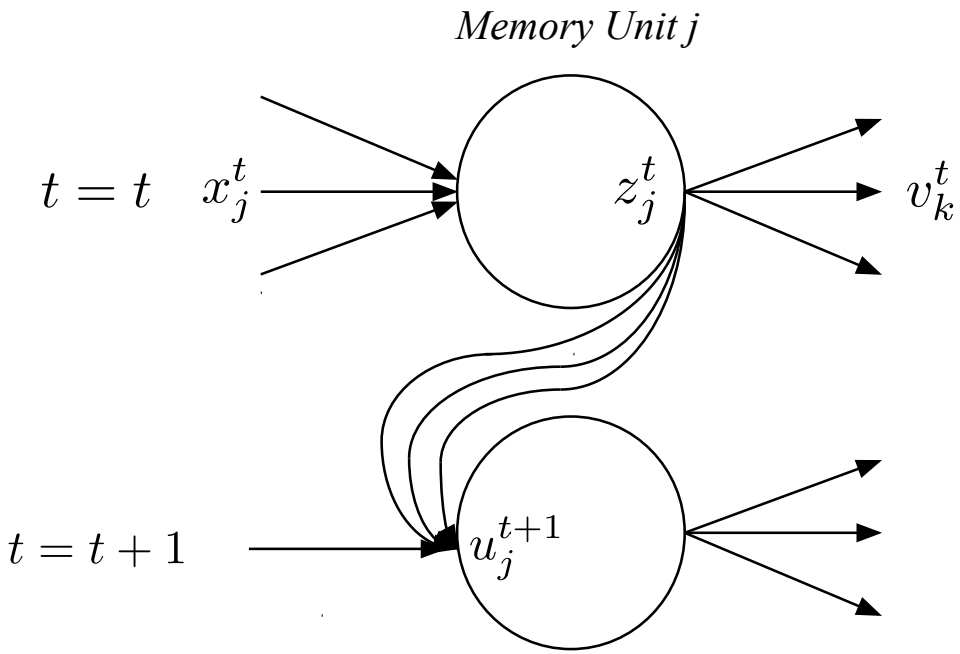
- 出力ゲートは以下で計算される

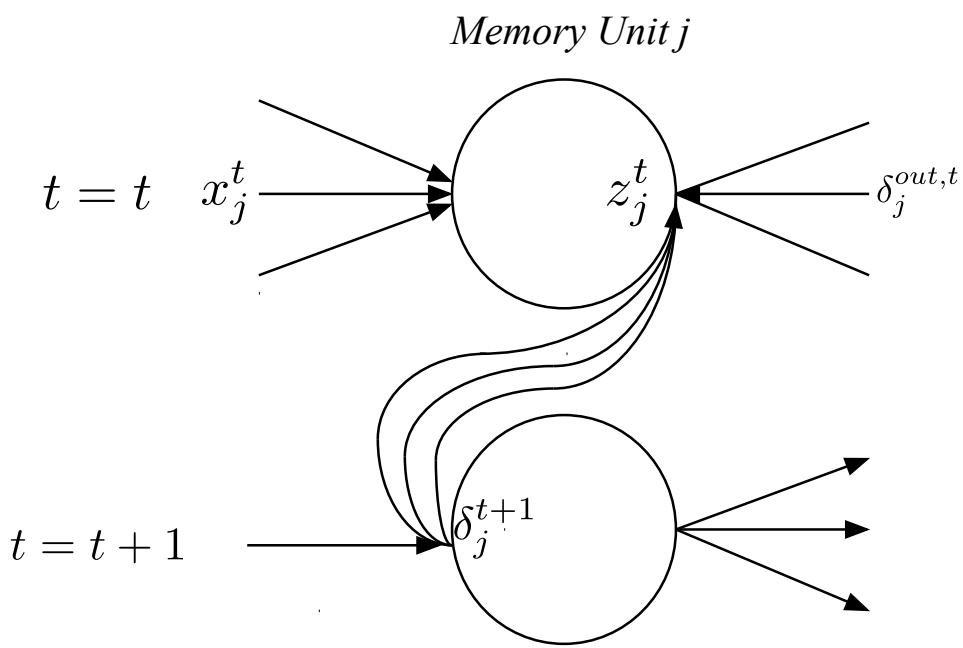
$$g_j^{O,t} = f(u_j^{O,t}) = f\left(\sum_i w^{O,in} x_i^t + \sum_{j'} w_{jj'}^O z_{j'}^{t-1} + w_j^O s_j^t\right) \tag{16}$$

- 出力ゲートのみ s_j^{t-1} でなく s_j^t を加算する
- 各ゲートの活性化関数 f はシグモイド関数（値域 $[0, 1]$）とする

LSTM : 逆伝搬計算 (1)

- 通常の RNN と同様に勾配降下法による学習が可能
- メモリユニットの変数のうち，全体の出力 z_j^t のみが他のユニットへ伝搬する





LSTM : 逆伝搬計算 (2)

- 例として，出力ゲートの値 $g_j^{O,t}$ を出力するユニットのデルタ $\delta_j^{O,t}$ を求める
- 出力層の影響を考えると，連鎖規則により以下の式が成り立つ

$$\delta_j^{O,t} = \frac{\partial E}{\partial u_j^{O,t}} = \sum_k \frac{\partial E}{\partial v_k^t} \frac{\partial v_k^t}{\delta u_j^{O,t}} + \sum_{j'} \frac{\partial E}{\partial u_{j'}^{t+1}} \frac{\partial u_{j'}^{t+1}}{\partial u_j^{O,t}} \tag{17}$$

- 誤差 δ の定義より

$$\delta_k^t = \frac{\partial E}{\partial v_k^t} \tag{18}$$

$$\delta_j^{t+1} = \frac{\partial E}{\partial u_{j'}} \tag{19}$$

LSTM : 逆伝搬計算 (3)

- v_k^t と u_j^{t+1} を $u_j^{O,t}$ の式で表す
- $z_j^t = g_j^{O,t} f(s_j^t) = f(u_j^{O,t}) f(s_j^t)$ より,

$$
\begin{aligned}
v_k^t &= \sum_k w_{kj}^{out} z_j^t && (20)\\
&= \sum_k w_{kj}^{out} f(u_j^{O,t}) f(s_j^t) && (21)\\
u_j^{t+1} &= \sum_i w_{ji}^{in} x_i^{t+1} + \sum_{j'} w_{jj'} z_{j'}^t && (22)\\
&= \sum_i w_{ji}^{in} x_i^{t+1} + \sum_{j'} w_{jj'} f(u_j^{O,t}) f(s_j^t) && (23)
\end{aligned}
$$

LSTM : 逆伝搬計算 (4)

- 式 (17) に式 (22) と式 (23) を代入すると,

$$\delta_j^{O,t} = \frac{\partial E}{\partial u_j^{O,t}} \tag{24}$$

$$= \sum_k \delta_k^{Out,t} f'(u_j^{O,t}) f(s^t) + \sum_j \delta_j^{t+1} w_{jj'} f'(u_j^{O,t}) f(s^t) \tag{25}$$

- 同様の手順で全てのユニットのデルタを求めることができる (計算略)

Gradient Checking

- ネットワークが複雑になると逆伝搬の計算でミスをしやすい
- バックプロパゲーションの実装のミスを確認する際,
  以下の式で勾配の近似値を計算できる

$$\frac{\partial E}{\partial w} = \frac{E(w+\epsilon) - E(w-\epsilon)}{2\epsilon} + O(\epsilon^2) \tag{26}$$

- $\epsilon = 0.0001$ がよく使われる

ライブラリ

ネットワークが複雑になるにつれて,

- 誤差の逆伝搬が複雑になる
- 計算コストが増大する

そのため，コスト関数の記号微分と CUDA コードの自動生成を行えるライブラリを用いる.

ニューラルネットワーク系のライブラリ

- **Theano**
- TensorFlow : 分散並列処理． google 製, gmail, alphago 等
- **Chainer** : 日本製 (PFI), backend: cupy
- Lasagne : backend: theano
- nolearn : これまでの研究に利用. backend: Lasagne
- Keras : backend: theano or tensorflow

Theano

- モントリオール大 Bengio 研究室
- Python 用の数値計算ライブラリ
  - 主にニューラルネットのバックエンドとして利用
  - MCMC 用のライブラリ (PyMC3) にも利用
- C++コードの自動生成
- 関数の微分のサポート (ベクトル，行列の関数に対応)
  - 重みの勾配は目的関数をパラメータで微分するだけでよい
  - 逆伝搬処理を書く必要がない
- CPU / GPU の切り替え

Theano：計算例

```
# x, t はシンボル変数
x = T.dmatrix('x')
t = T.dmatrix('t')
# w1, w2, b1, b2 は shared 変数
# GPU で計算している場合 GPU のメモリに乗る
z1 = T.nnet.relu(T.dot(x, w1) + b1)
y = T.nnet.softmax(T.dot(z1, w2) + b2)
y_f = theano.function([x], y)
cost = - T.mean(t * T.log(y))
gw1, gw2, gb1, gb2 = T.grad(cost=cost, wrt=[w1, w2, b1, b2])

# function でコンパイルされる
train_model = theano.function(
    inputs=[],
    outputs=cost,
    updates=[(w1, w1 - alpha * gw1), (b1, b1 - alpha * gb1),
             (w2, w2 - alpha * gw2), (b2, b2 - alpha * gb2)],
    givens={x: X_train, t: t_train}
)
```

Chainer

Preferred Networks
バックエンド：numpy との互換性の高い Cupy（GPU 計算）
ネットワークの構築に **Define and Run** のアプローチをとる

Define and Run

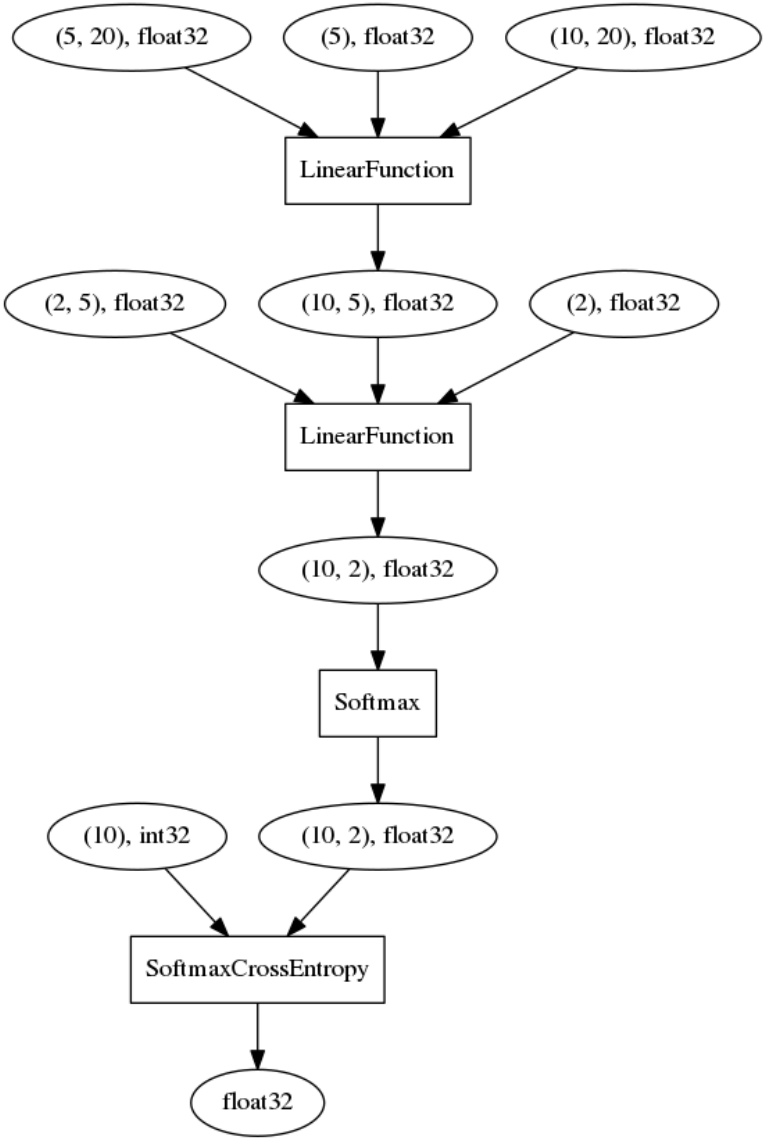
- 計算グラフを feedforward 毎に動的に構築する
- 複雑なネットワークに対応することができる
- 計算グラフの構築に時間が掛かる

Define by Run

- 計算グラフを最初に構築する
- 計算速度が早い
- 複雑なネットワークの対応が難しい（RNN，LSTM）
- Theano, 多くのニューラルネットワークライブラリ

Chainer：計算グラフ構築の例

```
x_v = Variable(x)
t_v = Variable(t)
h1 = model.l1(x_v)
h2 = model.l2(h1)
y = F.softmax(h2)
loss = F.softmax_cross_entropy(y, t_v)
loss.backward()
```



実験

- RNN / LSTM (chainer/GTX titan X を利用)
- 1 周期あたり 126 サンプル ($t = 1, \ldots, 126$, $f_s = 120Hz$)
- 特徴量に遅延あり/なしで実験
- エポック数 10 回（従来の NN では 300 回）
- 0.5 で Dropout を実施
- 隠れ層のユニット数 : 500 (従来の NN では 300)

実験結果：特徴量に遅延を含まないとき

Table: 識別率

	RNN	LSTM
Sub. 1	0.135417	0.041667

Table: 相関係数

	RNN	LSTM
Sub. 1	0.071714	0.021419

実験結果：特徴量に遅延を含むとき

Table: 識別率

	RNN	LSTM
Sub. 1	0.9826	0.9861
Sub. 2	0.7569	0.6907
Sub. 3	0.9201	0.8715
Sub. 4	0.9132	0.8958
Sub. 5	0.8160	0.7917
Ave.	0.8778	0.8472

Table: 相関係数

	RNN	LSTM
Sub. 1	0.7633	0.7704
Sub. 2	0.3982	0.3506
Sub. 3	0.6082	0.5651
Sub. 4	0.6480	0.5997
Sub. 5	0.5031	0.4588
Ave.	0.5842	0.5489

実験結果 (従来)

Table: 識別率

	CCA	LMSE	lasso	NN
Sub. 1	0.9861	**0.9931**	**0.9931**	0.9896
Sub. 2	0.8924	0.9028	**0.9132**	0.9028
Sub. 3	0.9549	0.9861	**0.9931**	0.9792
Sub. 4	0.9132	**0.9271**	**0.9271**	**0.9271**
Sub. 5	0.8576	0.8750	**0.8785**	0.8750
Ave.	0.9208	0.9368	**0.9410**	0.9347

Table: 相関係数

	CCA	LMSE	lasso	NN
Sub. 1	0.2328	0.5853	0.5952	**0.7705**
Sub. 2	0.1496	0.4816	0.4928	**0.5699**
Sub. 3	0.1520	0.4961	0.5038	**0.6516**
Sub. 4	0.2749	0.5821	0.5900	**0.6985**
Sub. 5	0.1581	0.4312	0.4512	**0.5864**
Ave.	0.1935	0.5153	0.5266	**0.6554**

考察

- RNN，LSTM ともに Feedforward NN に比べ低い性能
- RNN と LSTM のパラメータチューニングはほぼ行っておらず
  イテレーション回数も少ないため，識別率，相関係数の向上は可能
- RNN/LSTM の学習は通常のニューラルネットに比べると難しい
- 特徴量に遅延を含めないとうまく学習できない
- 遅延を含めない場合，モデルを複雑にするとトレーニングデータに過学習する
  ⇒ ランダムに遅延した訓練データではうまく学習しない

まとめと今後の課題

まとめ

- RNN で長期的な記憶を実現には LSTM を用いる
- c-VEP BCI に RNN/LSTM を適用した

今後の課題

- パラメータチューニング