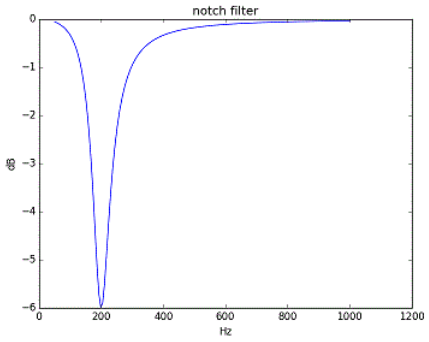
＜ Chainerを使って 2次 IIR ノッチ フィルターを学習できるか？ ＞

2次の IIR ノッチ フィルターの係数をChainerで学習できるかどうか試みる。
今一のフィット感しかない線形システムを（非線形要因も含むであろう）ディープラーニングを使って より合致するようにチューニングできないか？と言うのが今回の動機である。しかし、結論として、ニューラルネットワーク(RBM,LSTM,・・・）ではうまく学習できなかった。
その理由は、今回のようなノッチ型のデジタルフィルターを実現するには　連続的で微妙な計算精度が必要なこと、そして、ニューラルネットワークが得意とする内部状態が　非線形的に　ざっくり　分離できるような対象になっていないため　と考えられる。
結局、chainerを　過去とその過去の状態を記憶しておくRNN?構成にして、単なる”線形”の推定計算器としてもちいることにした。

ノッチ フィルターの周波数特性の例を以下に示す。ノッチフィルターはある特定の周波数成分（ノイズなど）を除去する目的で使われる。当初は、活性化関数を使うことを考えると出力の値が±１以内に収まる方がよい考え、信号が大きくなるブースト　タイプでなく、信号が小さくなるノッチ　タイプを用いることにした。



（デジタル）フィルターは以下の計算式で計算される。

2次のIIRフィルターの入出の計算の例

y[0]= b[0]＊x[0]　+ b[1]＊x[−1] + b[2]＊x[−2]
y[0]= y[0] − a[1]＊y[−1] − a[2]＊y[−2]

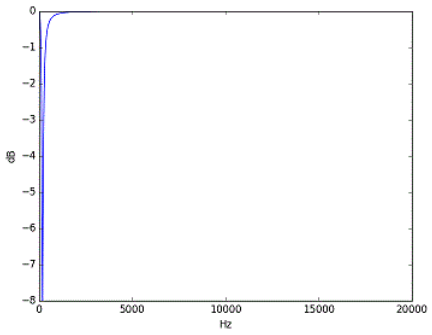
x[],y[]は　入力と出力
a[], b[]は　フィルターの係数
[0]は現在の値、[−1]は１個前の値、[−2]は2個前の値を示す
a[0]は１としておく

上記の計算式の中の係数a[],b[]をノッチフィルターの場合　具体的に書いてみると

```
+++ comparison of iir filter coefficient
   initial  present target
a[0] 1.0 1.0 1.0
a[1] -1.98637463995 -1.9853 -1.98103697815
a[2] 0.987055554261 0.985857248306 0.981587006828
b[0] 0.996771572606 0.995247 0.99445867572
b[1] -1.98637463995 -1.9853 -1.98103697815
b[2] 0.990283981655 0.99061 0.987128331107
```

左列が減衰周波数が200Hzのもの、右列が減衰周波数180Hzのものであり、小数点3桁目以降の微妙な差しかないことが分かる。
計算途中に活性化関数など　非線形にゆがめてしまうものが入ると　フィルターそのものの再現が怪しくなる。

上図のノッチ フィルターの周波数特性の例は　ある特性の周波数の部分を拡大したものであるが、
ナイキスト周波数の帯域からみると、下図の様に下線のゴミのようになってしまう。
SIN波を入力してSIN波を出力するニューラルネットワークは合成できるのであるが、このゴミのような微妙な周波数特性を学習できるようにはできなかった。



chainerを　過去とその過去の状態を記憶しておくRNN?構成にして、単なる”線形”の推定計算器として使うことにした。これでは、他の数値計算手法と変わらないので　わざわざchainerを使う必要はないのであるが、やってみることにした。

ネットワークの構成を class CharRNNとして以下のように定義した。

```
class CharRNN(FunctionSet):
    def __init__(self, in_size=IN_SIZE0, out_size=OUT_SIZE0, hn_units=HIDDEN_UNITS0, train=True):
        super(CharRNN, self).__init__(
            # 1st coefficient fixed to 1
            l2_x = L.Linear(hn_units, hn_units, nobias=True),
            l2_h = L.Linear(hn_units, hn_units, nobias=True),
            l3_h = L.Linear(hn_units, hn_units, nobias=True),
            #l3_x = L.Linear(hn_units, hn_units, nobias=True),
            #l4_h = L.Linear(hn_units, hn_units, nobias=True),
            #l5_h = L.Linear(hn_units, hn_units, nobias=True),
            l6=L.Linear(hn_units, out_size, nobias=True),
        )

        ....

    def forward_one_step(self, x, y, state, train=True, dropout_ratio=0.0):
        h1 = x
        c2  = self.l2_x(F.dropout(h1, ratio=dropout_ratio, train=train)) + self.l2_h(state['h2']) + self.l3_h(state['h3'])
        h3=state['h2']
        h2=h1
        c3  = c2 - self.l2_h(state['h4']) - self.l2_x(state['h5']) - self.l3_h(state['h5']) + state['h5']
        #c3= F.tanh(c3)
        h5=state['h4']
        h4=c3
        t = self.l6(c3)

        state  = {'h2': h2 , 'h3': h3, 'h4': h4 , 'h5': h5}

        self.loss = F.mean_squared_error(y, t)
        self.prediction=t

        ....
```

h2,h3,h4,h5が入力と出力それぞれの過去とその過去の状態を記憶することを意味し、forward_one_stepは(デジタル)フィルターの計算式に倣っている(以下の表も参照のこと)。前述の理由から活性化関数(tanh,・・・)はコメント文にして使っていない。
推定計算を実際に行ってみると、初期値＞0の場合、直流が加算しているようなもので、そのうちにバイアス要素は零へ収束していくことが分かる。そこで、Linearのオプションでnobias=Trueを指定して はじめからバイアス無しにしている。

optimizer ADAMのステップ毎にパラメーターを更新する大きさアルファの値も、デフォルトの値より1桁小さくした。微妙な変化しかない超平面上を うろつくので大きすぎると行き過ぎて収束しないようである。

```
#optimizer = optimizers.Adam(alpha=0.001, beta1=0.9, beta2=0.999, eps=1e-08)  # default of optimizaier Admam
optimizer = optimizers.Adam(alpha=0.0001 )  # Change: optimizaier Admam alpha value from default
```

下記の表は、フィルターの係数とCharRNNの中のパラメーターとの対応を示したものである。
2次のIIRフィルターの係数a[],b[]は6個あるが、ノッチフィルターはそのうちの3個だけを使って実現できる。

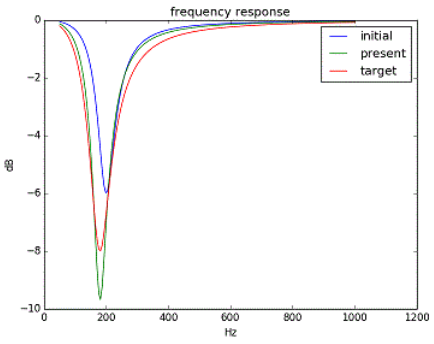
ファイルの係数とCharRNNの係数の対応の表

2次IIR ノッチフィルターの係数	CharRNNの中の係数(Linear関数のバイアスを無しで使用)
b[0]=1.0 + (kc * (1.0-b3)/2.0)	l2_x.W
b[1]=-1.0 * (1.0 + b3) * np.cos(wc)	l2_h.W
b[2]=b3 - (kc * (1.0 - b3) /2.0)	l3_h.W
a[0]=1.0	l3_x.W　1に固定
a[1]=-1.0 * (1.0 + b3) * np.cos(wc)=b[1] =b[2]	l4_h.W 符号は逆になる
a[2]=b3 =b[2]+b[0]-1	l5_h.W 符号は逆になる
	非線形な活性化関数は使用せず、1倍とした

係数を束縛してこの3個だけを使えば ネットワークが ノッチ型の周波数特性をもつことは(値が正ならば)保証されるはずであるが、自由度を拡張して 4個、5個にした場合の結果も最後方に示しておく。

今回の課題は
初期値 initial
減衰周波数200Hz
減衰利得-6dB
Q(減衰する周波数の幅に相当)8
の2次 IIR ノッチフィルター からスタートして
目標 target
減衰周波数180Hz
減衰利得-8dB
Q(減衰する周波数の幅に相当)8
の訓練データを与えた学習調整後の値 present
が目標にどれだけ近づけるか？

下図は 初期値、目標、学習調整後、それぞれの周波数特性の比較例である。
減衰周波数は目標に達しているが、減衰利得や幅は外れている。

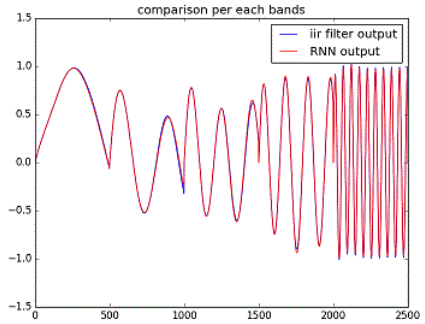


ネットワークのパラメータは乱数で初期化されるが、ここでは対象の素性はおよそわかっている場合を想定し、乱数ではなく上記のように具体的な値を初期値として与えいる。

訓練は、異なる5個の周波数帯域の中から それぞれランダム選んだ周波数をもつSIN波の入出力信号を使っておこなった。
異なる5個の帯域が同時？に成り立つような、連立方程式を解くような効果を期待している。
ナイキスト周波数からみると減衰する周波数帯域はごく狭い範囲のため、均等に分割してしまうとその特徴は埋もれてしまう。
入力に一様乱数をつかう場合でも、このように狭い特徴の学習は難しい。
そこで、特徴を表す部分をクローズアップした、以下の6個の周波数で区切られた周波数帯域(BAND)を使用した。

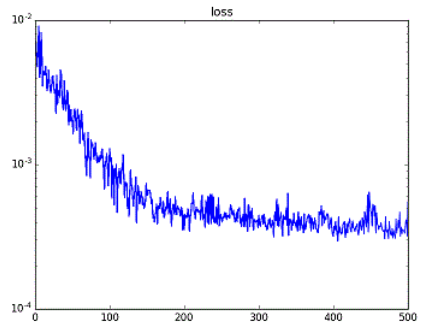
```
...frequency of band for batch
i,band = 0 50.0
i,band = 1 91.0282101513
i,band = 2 165.722700867
i,band = 3 301.708816827
i,band = 4 549.280271653
i,band = 5 1000.0
```

下図は、1帯域(1バンド)あたり　500サンプル分の　正規のIIRフィルターで計算したものと訓練学習したRNN?ネットワーク出力の比較例である。1から500までは初めのバンド、501から1000までが次のバンド、更に次は1001 から1500までと、合計5バンド分2500個を表示している



ネットワークのパラメータの更新は、5帯域(5バンド)の　シーケンス長　500サンプル毎に行う。

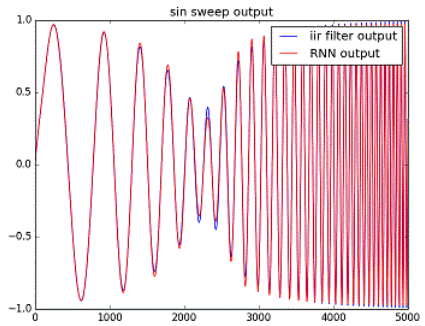
損失は、正規のIIRフィルターで計算したものと 訓練学習したRNN?ネットワーク出力 の差の 二乗平均で評価した。
下図は、パラメーター更新回数とその損失(ロス)の変化の様子。



初期値、学習調整後の、目標値、それぞれのフィルターの係数a[],b[]の値。

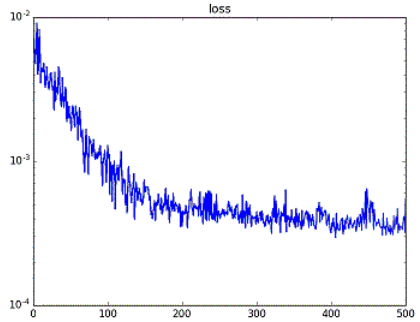
```
+++ comparison of iir filter coefficient
    initial  present target
a[0] 1.0 1.0 1.0
a[1] -1.98637463995 -1.9853 -1.98103697815
a[2] 0.987055554261 0.985857248306 0.981587006828
b[0] 0.996771572606 0.995247 0.99445867572
b[1] -1.98637463995 -1.9853 -1.98103697815
b[2] 0.990283981655 0.99061 0.987128331107
```

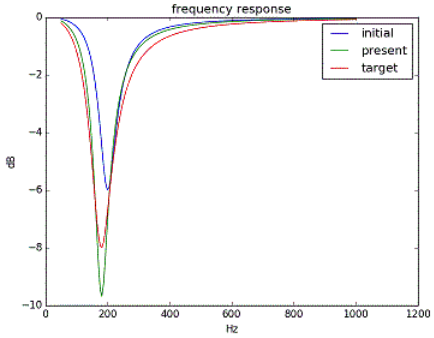
下図は、入力信号に 低い周波数から高い周波数まで連続的に周波数を変化(スイープ)させたSIN波を使ったものの比較で、減衰周波数付近で振幅が減衰することが分かる。

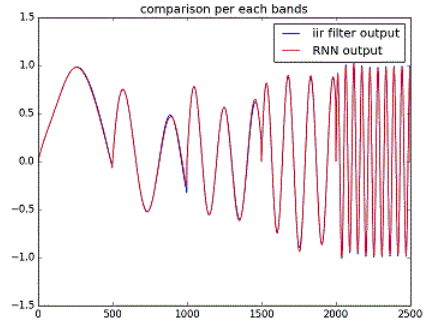


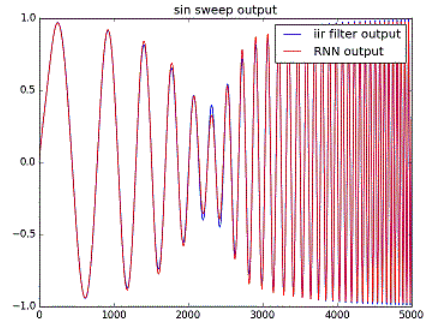
以下は、自由度3個、4個、5個の場合の結果の比較である。

3 parameters model. l2_x, l2_h, l3_h
import CharRNN1 as rnn

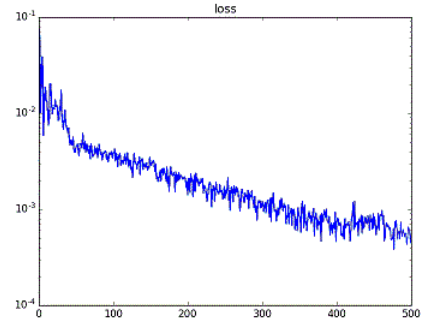


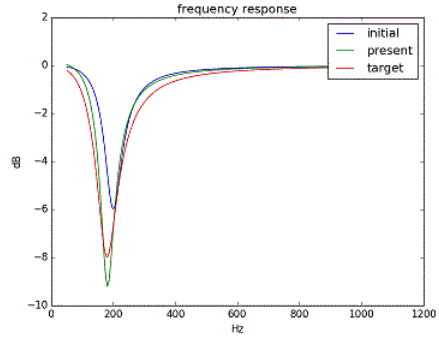


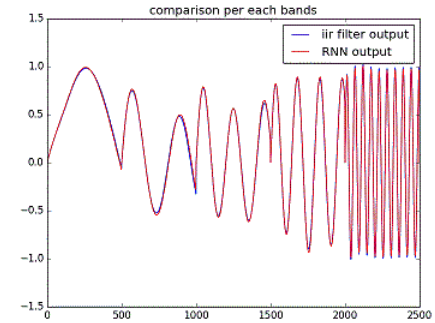


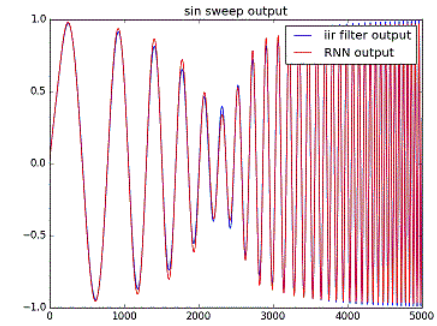


4 parameters model. l2_x, l2_h, l3_h, l5_h
import CharRNN2 as rnn









5 parameters model. l2_x, l2_h, l3_h, l5_h ,l4_h
import CharRNN3 as rnn