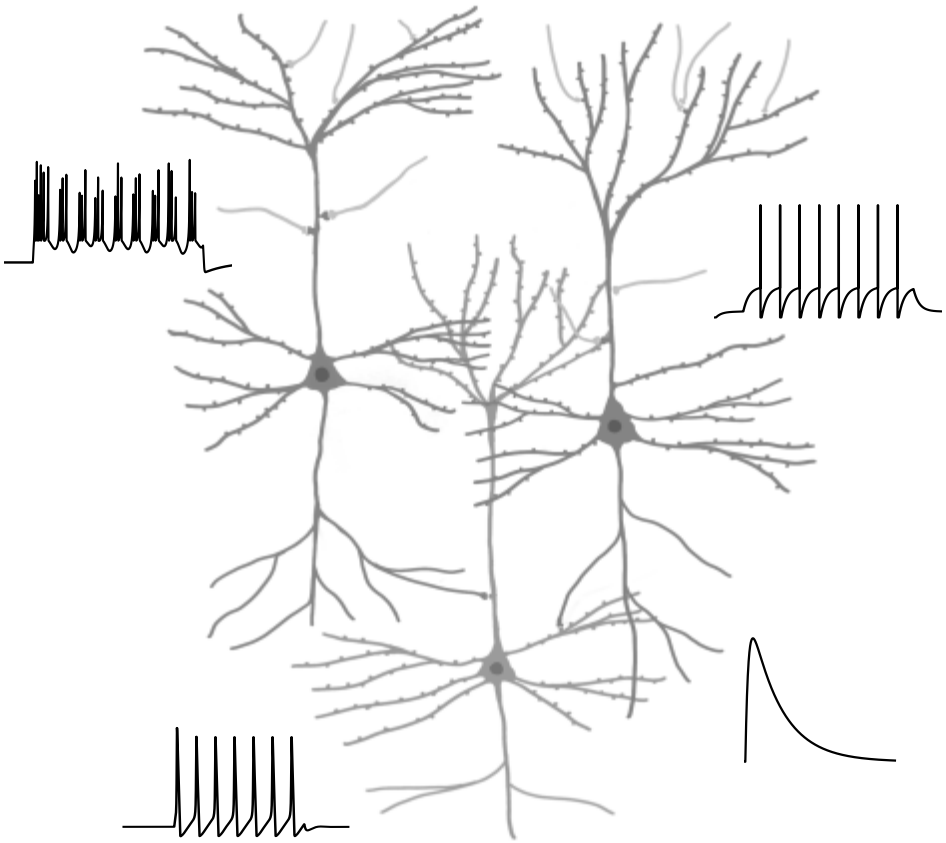
Pythonで

ゼロから作る

Spiking Neural Networks

NumPyによるSNNの
シミュレーションと学習則の実装



ゼロから作る Spiking Neural Networks

【第 2.05 版】

山拓　著

前書き

この本は **Spiking Neural Networks**(SNN) を Python で実装することを目標とする本です (ライブラリは基本的に **NumPy** と **Matplotlib** のみを用います). 単なる神経活動のシミュレーションでとどまらず, ネットワークの学習則まで実装することを目標とします.

初めになぜこの本を書こうと思ったかを説明しておきます. Spiking Neuron については既に和書・洋書共に優れた本がありますが, 近年の人工神経回路 (artificial neural networks; **ANN**) の成功を踏まえた上で解説している本はありません. SNN は神経系のシミュレーションのみならず, 近年では機械学習の応用も進んでいます. 実際 ANN の発展により SNN も発展しているのですが, 今一つ SNN の研究はとっつきにくく, 解説もほぼありません. そこでこの本では SNN を実際に実装しながら SNN を理解し, SNN の研究がどこまで進んできたか, 今後の課題は何か, ということについて, 計算論的神経科学の観点 (脳をシミュレーションするという観点[*1]) と, 機械学習への応用の観点から考える本になっています. 実装言語として Python を選んだのは, ANN を含む機械学習を行うのに最も用いられているためです. 正直に言えば Python は SNN に向いている言語ではありませんが[*2], ANN を知っている人なら扱えると思い選びました.

断っておきますが, 筆者は SNN の研究を長年やってきた, というわけではありません. この本は SNN の教科書ではなくまとめノートであるということを踏まえて読んでいただければと思います. 専門ではないがゆえに内容に誤りがあると思いますが, もし見つけられた場合は twitter 上ではありますが, `@tak_yamm` までご連絡していただければ幸いです.

[*1] 参考までですが, ANN と脳の対応については `https://github.com/takyamamoto/BNN-ANN-papers` という論文リストを作成しております.

[*2] C++ や Julia は向いている言語の例です.

0.1 Python とライブラリのバージョン

本書のコードは Python 3.7.3 で実行しました. また, 本書で用いた全てのライブラリのバージョンは次の通りです.

- numpy == 1.16.4
- matplotlib == 3.1.0
- tqdm == 4.32.2
- scipy == 1.3.0
- chainer == 6.0.0rc1

なお, 本書ではプログレス管理のために `tqdm` を用います[*3].

また, コードが冗長になるのを防ぐため, **本書における全てのコードは冒頭に**

```
import numpy as np
import matplotlib.pyplot as plt
from tqdm import tqdm
np.random.seed(seed=0)
```

を既に記述しているものとします.

0.2 GitHub でのコードの公開

本書で用いたコードは GitHub で公開しており, `https://github.com/takyamamoto/SNN-from-scratch-with-Python` から確認できます. また, 説明の際に対応するコードはその都度, 欄外に表示しています.

0.3 本書を読む上での前提知識

本書は時間の都合上, 日本語文献が十分にある前提知識に関しては省略しています. 対象読者として以下のことを理解している方を想定しています. ご了承ください.

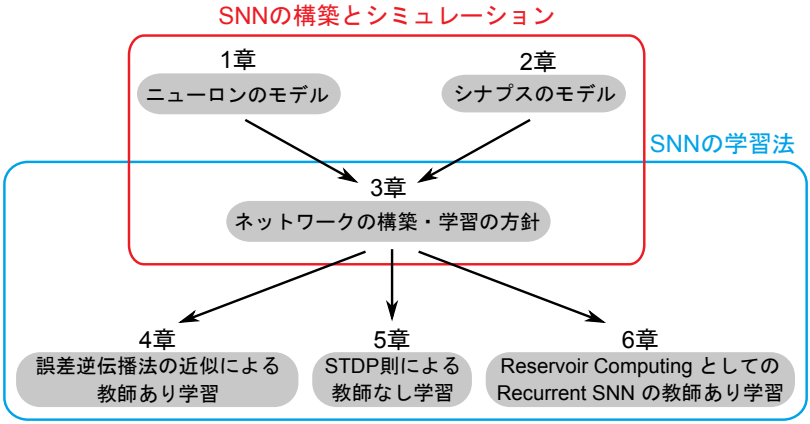
- 初歩的な微積分と線形代数
- 人工神経回路 (ANN) の基礎
- 神経生理学の基礎

[*3] `fastprogress` (https://github.com/fastai/fastprogress) 派の方は適宜置き換えてください.

目次

本書の各章の関係

本書の各章の関係は次の図のようになっています．参考にしてお読みいただければと思います．



第 1 章

ニューロンのモデル

ニューロンのモデル[*1](ニューロンの膜電位変化のモデル) は数多く考案されていますが[*2], 重要なモデルに絞って解説します. 解説するモデルは, 金字塔である **Hodgkin-Huxley モデル**, SNN によく用いられる **Leaky integrate-and-fire モデル**と **Izhikevich モデル**, 入力スパイクとして用いられる **Inter-spike interval モデル**です. さらに Spike-based ではないですが, **発火率 (firing rate) モデル**についても解説します.

1.1 Hodgkin-Huxley モデル

1.1.1 Hodgkin-Huxley モデルにおける膜の等価回路モデル

Hodgkin-Huxley モデル[*3](HH モデル) は, A.L. Hodgkin と A.F. Huxley によって 1952 年に考案されたニューロンの膜興奮を表すモデルです. 彼らはヤリイカの巨大神経軸索に対する**電位固定法** (voltage-clamp) を用いた実験を行い, 実験から得られた観測結果を元にモデルを構築しています.

HH モデルには等価な電気回路モデルがあり, **膜の並列等価回路モデル** (parallel conductance model) と呼ばれています. 膜の並列等価回路モデルでは, ニューロンの細胞膜をコンデンサ, 細胞膜に埋まっているイオンチャネルを可変抵抗 (動的に変化する抵抗) として置き換えます[*4]. **イオンチャネル** (ion channel) は特定のイオン (例えばナトリウ

[*1] シナプスもニューロンに含まれる構造なので, 分けるのは変な感じですが, 分けたほうが実装は楽なのでこうしています.

[*2] 他にどのようなモデルが考案されているかについては (Izhikevich, 2004) などを参照してください.

[*3] この節はどう書いても (宮川 & 井上, ニューロンの生物物理, 2013) の劣化とならざるを得ないので実装以外はそちらを読んでほしいです.

[*4] なお, 当時は Hodgkin と Huxley はイオンの通り道があるということは分かっていましたが, イオンチャネルの存在はまだ分かっていませんでした.

ムイオンやカリウムイオンなど) を選択的に通す[*5]膜輸送体の一種です. それぞれのイオンの種類において, 異なるイオンチャネルがあります (また同じイオンでも複数の種類のイオンチャネルがあります). また, イオンチャネルにはイオンの種類に応じて異なる**コンダクタンス** (抵抗の逆数で電流の「流れやすさ」を意味します) と**平衡電位** (equilibrium potential) があります[*6]. HH モデルでは, ナトリウム (Na^+) チャネル, カリウム (K^+) チャネル, 漏れ電流 (leak current) のイオンチャネルを仮定します. 漏れ電流のイオンチャネルは当時特定できなかったチャネルで, 膜から電流が漏れ出すチャネルを意味します. なお, 現在では漏れ電流の多くは Cl^- イオン (chloride ion) によることが分かっています.

それでは, 等価回路モデルを用いて電位変化の式を立ててみましょう. 図 1.1 において, C_m を細胞膜のキャパシタンス (膜容量), $I_m(t)$ を細胞膜を流れる電流 (外部からの入力電流), $I_{\mathrm{Cap}}(t)$ を膜のコンデンサを流れる電流, $I_{\mathrm{Na}}(t)$, $I_K(t)$ をそれぞれナトリウムチャネルとカリウムチャネルを通って膜から流出する電流, $I_{\mathrm{L}}(t)$ を漏れ電流とします. このとき,

$$I_m(t) = I_{\mathrm{Cap}}(t) + I_{\mathrm{Na}}(t) + I_{\mathrm{K}}(t) + I_{\mathrm{L}}(t) \tag{1.1}$$

という仮定をおきます. よって膜電位を $V(t)$ とすると, Kirchhoff の第二法則 (Kirchhoff's Voltage Law) より,

$$\underbrace{C_m \frac{dV(t)}{dt}}_{I_{\mathrm{Cap}}(t)} = I_m(t) - I_{\mathrm{Na}}(t) - I_{\mathrm{K}}(t) - I_{\mathrm{L}}(t) \tag{1.2}$$

となります.

Hodgkin と Huxley はチャネル電流 $I_{\mathrm{Na}}, I_K, I_{\mathrm{L}}$ が従う式を実験的に求めました.

$$I_{\mathrm{Na}}(t) = g_{\mathrm{Na}} \cdot m^3 h (V - E_{\mathrm{Na}}) \tag{1.3}$$

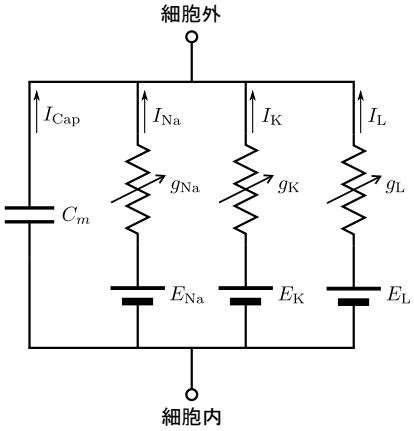
$$I_{\mathrm{K}}(t) = g_{\mathrm{K}} \cdot n^4 (V - E_{\mathrm{K}}) \tag{1.4}$$

$$I_{\mathrm{L}}(t) = g_{\mathrm{L}} (V - E_{\mathrm{L}}) \tag{1.5}$$

ただし, $g_{\mathrm{Na}}, g_{\mathrm{K}}$ はそれぞれ Na^+, K^+ の最大コンダクタンスです. g_{L} はオームの法則に従うコンダクタンスで, L コンダクタンスは時間的に変化はしません. また, m は Na^+ コ

[*5] イオンチャネルでは, あるイオンを通すイオンチャネルの中を, それより小さいイオンが通るということが起こらないようになっています. このことはイオンチャネルの分子構造の設計が非常に優れたものであることを示します.

[*6] イオンの流れは**濃度勾配**と**電位勾配**による駆動力 (driving force) に基づきます. 濃度勾配による駆動力を拡散 (diffusion), 電位勾配による駆動力をドリフト (drift) といいます. 細胞膜の電位変化により, イオンの細胞内への流入量と細胞外への流出量が等しくなった時 (つまり拡散とドリフトがつり合った時), 見かけ上イオンの流れが無くなります. このときの電位を**平衡電位 (equilibrium potential)** と呼びます. 平衡電位を境としてイオンの正味の流れの向きが変化するので, 平衡電位のことを**反転電位 (reversal potential)** とも呼びます.



▲ 図 1.1　HH モデルにおける膜の等価回路モデル

ンダクタンスの活性化パラメータ, h は Na^+ コンダクタンスの不活性化パラメータ, n は K^+ コンダクタンスの活性化パラメータであり, ゲートの開閉確率を表しています. よって, HH モデルの状態は V, m, h, n の 4 変数で表されます. これらの変数は以下の x を m, n, h に置き換えた 3 つの微分方程式に従います.

$$\frac{dx}{dt} = \alpha_x(V)(1-x) - \beta_x(V)x \tag{1.6}$$

ただし, V の関数である $\alpha_x(V)$, $\beta_x(V)$ は m, h, n によって異なり, 次の 6 つの式に従います.

$$\begin{aligned}
\alpha_m(V) &= \frac{0.1(25-V)}{\exp\left[(25-V)/10\right]-1}, & \beta_m(V) &= 4\exp\left(-V/18\right) \\
\alpha_h(V) &= 0.07\exp\left(-V/20\right), & \beta_h(V) &= \frac{1}{\exp\left[(30-V)/10\right]+1} \\
\alpha_n(V) &= \frac{0.01(10-V)}{\exp\left[(10-V)/10\right]-1}, & \beta_n(V) &= 0.125\exp\left(-V/80\right)
\end{aligned} \tag{1.7}$$

1.1.2　Hodgkin-Huxley モデルの実装

それでは, これまでに説明した式を用いて HH モデルを実装してみましょう. 定数は次のように設定します.

$$\begin{aligned}
C_m = 1.0, g_{\mathrm{Na}} = 120, g_{\mathrm{K}} = 36, g_{\mathrm{L}} = 0.3 \\
E_{\mathrm{Na}} = 50.0, E_{\mathrm{K}} = -77, E_{\mathrm{L}} = -54.387
\end{aligned}$$

基本的には以上の式を関数として定義し, 微分方程式のソルバー[*7]に渡すだけです. ソルバーとしては陽的 Euler 法または 4 次の Runge-Kutta 法[*8]を用います. また, 少し複雑になるのでここでは 1 ニューロンのみのシミュレーションを行います. なお, コードの書かれた Python ファイルの名称とパスは基本的に欄外に示します[*9]. まずは HH モデルのクラスを定義します.

```
class HodgkinHuxleyModel:
    def __init__(self, dt=1e-3, solver="RK4"):
        self.C_m  =   1.0 # 膜容量 (uF/cm^2)
        self.g_Na = 120.0 # Na+ の最大コンダクタンス (mS/cm^2)
        self.g_K  =  36.0 # K+ の最大コンダクタンス (mS/cm^2)
        self.g_L  =   0.3 # 漏れイオンの最大コンダクタンス (mS/cm^2)
        self.E_Na =  50.0 # Na+ の平衡電位 (mV)
        self.E_K  = -77.0 # K+ の平衡電位 (mV)
        self.E_L  = -54.387 #漏れイオンの平衡電位 (mV)

        self.solver = solver
        self.dt = dt

        # V, m, h, n
        self.states = np.array([-65, 0.05, 0.6, 0.32])
        self.I_m = None

    def Solvers(self, func, x, dt):
        # 4th order Runge-Kutta 法
        if self.solver == "RK4":
            k1 = dt*func(x)
```

[*7] SNN は基本的に微分方程式を解いてシミュレーションを行います. ソルバーとしては計算量の観点から陽的 Euler 法が主に用いられますが, 精度を上げたい場合には 4 次の Runge-Kutta 法などを用います. これ以降は Euler 法しか使用しません.

[*8] Runge-Kutta 法という文字を見るだけで興奮する人もいるらしい. Runge-Kutta という文字に応答する Runge-Kutta ニューロンでもあるのでしょうか.

[*9] コードは./SingleFileSimulations/Neurons/HH_single.py です. 実装において https://hodgkin-huxley-tutorial.readthedocs.io/en/latest/index.html を参考にしました. また複数ニューロンのシミュレーション例は ./SingleFileSimulations/Neurons/HH_model_multiple_neurons.py です.

```
            k2 = dt*func(x + 0.5*k1)
            k3 = dt*func(x + 0.5*k2)
            k4 = dt*func(x + k3)
            return x + (k1 + 2*k2 + 2*k3 + k4) / 6

        # 陽的 Euler 法
        elif self.solver == "Euler":
            return x + dt*func(x)
        else:
            return None

    # イオンチャネルのゲートについての 6 つの関数
    def alpha_m(self, V):
        return 0.1*(V+40.0)/(1.0 - np.exp(-(V+40.0) / 10.0))

    def beta_m(self, V):
        return 4.0*np.exp(-(V+65.0) / 18.0)

    def alpha_h(self, V):
        return 0.07*np.exp(-(V+65.0) / 20.0)

    def beta_h(self, V):
        return 1.0/(1.0 + np.exp(-(V+35.0) / 10.0))

    def alpha_n(self, V):
        return 0.01*(V+55.0)/(1.0 - np.exp(-(V+55.0) / 10.0))

    def beta_n(self, V):
        return 0.125*np.exp(-(V+65) / 80.0)

    # Na+ 電流 (uA/cm^2)
    def I_Na(self, V, m, h):
        return self.g_Na * m**3 * h * (V - self.E_Na)
```

```
    # K+ 電流 (uA/cm^2)
    def I_K(self, V, n):
        return self.g_K  * n**4 * (V - self.E_K)

    # 漏れ電流 (uA/cm^2)
    def I_L(self, V):
        return self.g_L * (V - self.E_L)

    # 微分方程式
    def dALLdt(self, states):
        V, m, h, n = states

        dVdt = (self.I_m - self.I_Na(V, m, h) \
                - self.I_K(V, n) - self.I_L(V)) / self.C_m
        dmdt = self.alpha_m(V)*(1.0-m) - self.beta_m(V)*m
        dhdt = self.alpha_h(V)*(1.0-h) - self.beta_h(V)*h
        dndt = self.alpha_n(V)*(1.0-n) - self.beta_n(V)*n
        return np.array([dVdt, dmdt, dhdt, dndt])

    def __call__(self, I):
        self.I_m = I
        states = self.Solvers(self.dALLdt, self.states, self.dt)
        self.states = states
        return states
```

まず, `__init__()` では定数の設定と初期化を行います. `self.states` は細胞膜の状態を表す 4 つの変数 (順に V, m, h, n) を格納する配列です. この `self.states` を各シミュレーションステップにおいて, `__call__()` により更新します[*10]. `self.solver` は微分方程式のソルバーを選択する変数で, `RK4`(4 次の Runge-Kutta 法) か `Euler`(Euler 法) が選べます. `Solvers()` は微分方程式のソルバーを定義する関数で, 上記のいずれかの手法を用いることができます.

シミュレーションにおいては `HodgkinHuxleyModel` のインスタンスを作成し, 刺激電

[*10] `__call__()` は Python においてクラスのインスタンスに引数を渡して呼び出すと実行されるメソッドです. つまり, インスタンスを関数のように扱うと呼び出されます.

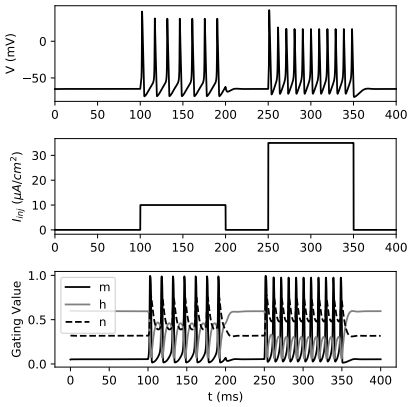
流を引数として渡します. ここで, 刺激電流 I_inj は矩形波の加減算により生成します.

```
dt = 0.01; T = 400 # (ms)
nt = round(T/dt) # シミュレーションステップ数
time = np.arange(0.0, T, dt)

 # 刺激電流 (uA/cm^2)
I_inj = 10*(time>100) - 10*(time>200) + 35*(time>250) - 35*(time>350)
HH_neuron = HodgkinHuxleyModel(dt=dt, solver="Euler")
X_arr = np.zeros((nt, 4)) # 記録用配列

for i in tqdm(range(nt)):
    X = HH_neuron(I_inj[i])
    X_arr[i] = X
```

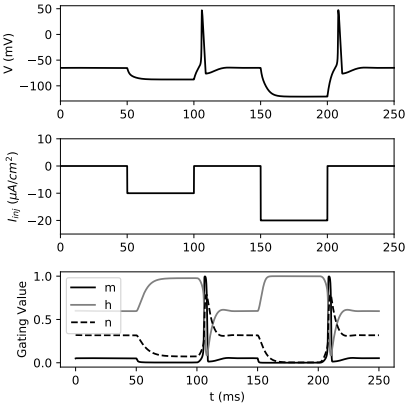
最後に結果を描画してみましょう.

```
plt.figure(figsize=(5, 5))
plt.subplot(3,1,1)
plt.plot(time, X_arr[:,0], color="k")
plt.ylabel('V (mV)'); plt.xlim(0, T)

plt.subplot(3,1,2)
plt.plot(time, I_inj, color="k")
plt.ylabel('$I_{inj}$ ($\\mu{A}/cm^2$)')
plt.xlim(0, T)

plt.subplot(3,1,3)
plt.plot(time, X_arr[:,1], 'k', label='m')
plt.plot(time, X_arr[:,2], 'gray', label='h')
plt.plot(time, X_arr[:,3], 'k', linestyle="dashed", label='n')
plt.xlabel('t (ms)'); plt.ylabel('Gating Value'); plt.legend(loc="upper left")
plt.show()
```

実行結果は図 1.2 のようになります.



▲ 図 1.2 Hodgkin Huxley モデルのシミュレーション. (上) 膜電位, (中) 印加電流, (下) ゲート変数



▲ 図 1.3 HH モデルによるアノーダルブレイクのシミュレーション. (上) 膜電位, (中) 印加電流, (下) ゲート変数

1.1.3 アノーダルブレイク

ニューロンは電流が流入することで膜電位が変化し, 膜電位がある一定の閾値を超えると発火が起こる, というのはニューロンの活動電位発生についての典型的な説明です. それでは HH モデルの膜電位閾値はどのくらいの値になるのでしょうか. 答えは「**膜電位閾値は一定ではない**」です. それを示す現象として**アノーダルブレイク** (anodal break, または anode break excitation; ABE) があります. アノーダルブレイクは, 過分極性の電流の印加を止めた際に膜電位が静止膜電位に回復するのみならず, さらに脱分極をして発火をするという現象です. HH モデルはアノーダルブレイクを再現できるため, シミュレーションによりどのような現象か確認してみましょう. これは前節で実装した HH モデルにおける入力電流を変更するだけで行えます[*11].

```
# 変更点
T = 250 # (ms)
I_inj = -10*(time>50) + 10*(time>100) - 20*(time>150) + 20*(time>200)
```

[*11] コードは./SingleFileSimulations/Neurons/HH_single_anodal_break.py です.

結果は図 1.3 のようになります.

なぜこのようなことが起こるか, というと過分極の状態から静止膜電位へと戻る際に Na^+ チャネルが活性化 (Na^+ チャネルの活性化パラメータ m が増加し, 不活性化パラメータ h が減少) し, 膜電位が脱分極することで再度 Na^+ チャネルが活性化する, というポジティブフィードバック過程 (**自己再生的過程**) に突入するためです (もちろん, この過程は通常の活動電位発生のメカニズムです). この際, 発火に必要な閾値が膜電位の低下に応じて下がった, ということもできるでしょう.

ということで膜電位閾値は一定ではありません. しかし, この後に紹介するモデルは簡略化のために if 文を用いて膜電位閾値を超えたから発火, というものもあります. 実際には違うということを頭の片隅に残しながら読み進めていただければと思います.

1.2 Leaky integrate-and-fire モデル

1.2.1 LIF モデルの単純な実装

生理学的なイオンチャネルの挙動は考慮せず, 入力電流を膜電位が閾値に達するまで時間的に積分するというモデルを **Integrate-and-fire (IF, 積分発火) モデル**といいます. さらに, IF モデルにおいて膜電位の漏れ (leak)[*12]も考慮したモデルを **Leaky integrate-and-fire (LIF, 漏れ積分発火) モデル**と呼びます. ここでは LIF モデルのみを取り扱います.

ニューロンの膜電位を $V_m(t)$, 静止膜電位を $V_{\rm rest}$, 入力電流[*13]を $I(t)$, 膜抵抗を R_m, 膜電位の時定数を τ_m $(= R_m \cdot C_m)$ とすると, 式は次のようになります[*14].

$$\tau_m \frac{dV_m(t)}{dt} = -(V_m(t) - V_{\rm rest}) + R_m I(t) \tag{1.8}$$

ここで, V_m が閾値 (threshold)[*15]$V_{\rm th}$ を超えると, 脱分極が起こり, 膜電位はピーク電位 $V_{\rm peak}$ まで上昇します. 発火後は再分極が起こり, 膜電位はリセット電位 $V_{\rm reset}$ まで低下するとします[*16]. 発火後, 一定の期間 $\tau_{\rm ref}$ の間は膜電位が変化しない[*17], とします (これを**不応期 (refractory time period)** と呼びます).

[*12] この漏れはイオンの拡散などによるものです.

[*13] シナプス入力による電流がどうなるかは, シナプスのモデルの項で扱います.

[*14] $(V_m(t) - V_{\rm rest})$ の部分は膜電位の基準を静止膜電位としたことにして, 単に $V_m(t)$ だけの場合もあります. また, 右辺の $RI(t)$ の部分は単に $I(t)$ とされることもあります. 同じ表記ですが, この場合の $I(t)$ はシナプス電流に比例する量, となっています (単位は mV).

[*15] th から始まるので文字 θ が使われることもあります.

[*16] リセット電位は静止膜電位と同じ場合もあれば, 過分極を考慮して静止膜電位より低めに設定することもあります.

[*17] 実装によっては不応期の間は膜電位の変化は許容するが発火は生じないようにすることもあります.

以上を踏まえて LIF モデルを実装してみましょう. 簡単のために, まずはクラスを用いずに 1 個のニューロンについてのシミュレーションをしてみます[*18]. まずは定数を定義します. なお, 時間の単位は秒に統一していますが, ミリ秒でも同じです. その場合は時定数などの時間の次元を持つ量を 1000 倍してください (後に紹介する Izhikevich モデルでは元の論文に合わせ, ミリ秒の単位を用いるので注意してください).

```
dt = 5e-5; T = 0.4 # (s)
nt = round(T/dt) # シミュレーションステップ数

tref = 2e-3 # 不応期 (s)
tc_m = 1e-2 # 膜時定数 (s)
vrest = -60 # 静止膜電位 (mV)
vreset = -65 # リセット電位 (mV)
vthr = -40 # 閾値電位 (mV)
vpeak = 30 #  ピーク電位 (mV)

t = np.arange(nt)*dt*1e3 # 時間 (ms)
I = 25*(t>50) - 25*(t>350) # 入力電流 (に比例する値)(mV)

# 初期化
v = vreset # 膜電位の初期値
tlast = 0 # 最後のスパイクの時間を記録する変数
v_arr = np.zeros(nt) # 膜電位を記録する配列
```

I は HH モデルのときと同じように矩形波を用いてパルス入力をしています. また, 入力電流ではなく入力電流に比例する量となっていますが, これは膜抵抗を乗じた後の値であると考えてください. それではメインとなる部分を書いてみましょう.

```
for i in tqdm(range(nt)):
    dv = (vrest - v + I[i]) / tc_m # 膜電位の変化量
    v = v + ((dt*i) > (tlast + tref))*dv*dt # 更新

    s = 1*(v>=vthr) # 発火の確認
```

[*18] コードは./SingleFileSimulations/Neurons/LIF_single.py です.

```
    tlast = tlast*(1-s) + dt*i*s # 発火時刻の更新
    v = v*(1-s) + vpeak*s # 発火している場合ピーク電位に更新
    v_arr[i] = v  # 膜電位の値を保存
    v = v*(1-s) + vreset*s # 発火している場合に膜電位をリセット
```

if 文を使って書かれることが多い部分を, if 文を使用せずに書いています[*19]. まず, `((dt*i) > (tlast + tref))` の部分は最後に発火した時刻 (`tlast`) に不応期 (`tref`) を足した時刻を現在の時刻 (`dt*i`) が上回っていれば, `v` に `dv*dt` を加算する, という意味です. 次に `(v>=vthr)` は Boolean 型となり, 膜電位が閾値 `vthr` を超えていれば `True`, そうでなければ `False` となります. これに 1 を乗じることで int 型となります[*20](1 が発火, 0 が未発火). その下は得られたスパイクの変数 `s` を用いて, 値の更新を行っています. 閾値を超えなかった場合 (`s=0`) は `(1-s)` が乗じられた値となり, 閾値を超えた場合 (`s=1`) は `s` が乗じられた値となります.

最後に `v_arr` に保存された膜電位変化を描画してみましょう.

```
plt.figure(figsize=(5, 3))
plt.plot(t, v_arr)
plt.xlim(0, t.max())
plt.xlabel('Time (ms)'); plt.ylabel('Membrane potential (mV)')
plt.show()
```

結果は図 1.4 のようになります.

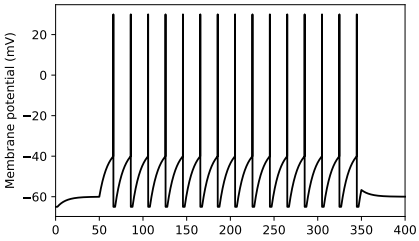
1.2.2 LIF モデルの class の実装

前節では変数を用いて LIF モデルを構成しました. 小さいモデルならこれでも良いのですが, モデルが複雑になると記述も煩雑になります. そこでこの節では class を用いて LIF モデルを記述します. とはいえ, 元になるのは前節のコードなので, 大きな違いはありません[*21]. このコードは 3 章以降で用いるので, `Neurons.py` に記述し, `Models` ディレクトリに保存しておきます.

[*19] これは Python が for ループを用いると遅いことによります. if 文を用いずにベクトルの計算で発火時の動態を記述することで「Python においては」高速に処理ができます.

[*20] 他には 0 を加算することでも Boolean 型から int 型への変換が可能です. `v` が配列ではなく, 1 変数の場合は `s = 1 if (v>=vthr) else 0` とする方が高速です.

[*21] コードは `./TrainingSNN/Models/Neurons.py` に含まれます.



▲ 図 1.4　LIF モデルの膜電位変化. 50 ms から 350 ms まで電流を印加している.

```
class CurrentBasedLIF:
    def __init__(self, N, dt=1e-4, tref=5e-3, tc_m=1e-2,
                 vrest=-60, vreset=-60, vthr=-50, vpeak=20):
        self.N = N
        self.dt = dt
        self.tref = tref
        self.tc_m = tc_m
        self.vrest = vrest
        self.vreset = vreset
        self.vthr = vthr
        self.vpeak = vpeak

        self.v = self.vreset*np.ones(N)
        self.v_ = None
        self.tlast = 0
        self.tcount = 0

    def initialize_states(self, random_state=False):
        if random_state:
            self.v = self.vreset + \
            np.random.rand(self.N)*(self.vthr-self.vreset)
        else:
            self.v = self.vreset*np.ones(self.N)
```

```
        self.tlast = 0
        self.tcount = 0

    def __call__(self, I):
        dv = (self.vrest - self.v + I) / self.tc_m
        v = self.v + (self.dt*self.tcount>(self.tlast+self.tref))*dv*self.dt

        s = 1*(v>=self.vthr) #発火時は 1, その他は 0 の出力

        self.tlast = self.tlast*(1-s) + self.dt*self.tcount*s # 発火時刻の更新
        v = v*(1-s) + self.vpeak*s # 閾値を超えると膜電位を vpeak にする
        self.v_ = v # 発火時の電位も含めて記録するための変数
        self.v = v*(1-s) + self.vreset*s  # 発火時に膜電位をリセット
        self.tcount += 1

        return s
```

名称が CurrentBasedLIF なのは，シナプス電流の記述の異なる別のモデル (ConductanceBasedLIF) があるためです (詳しくは第 2 章において説明します). N はニューロンの数を表し，多数のニューロンを同時にシミュレーションできるようになっています．initialize_states() は膜電位などをリセットしたいときに用います．__call__() はインスタンスを関数のように呼び出したときに実行される関数です．ネットワークの出力は s というスパイクを表す変数となっていますが，外部から膜電位変化を確認したい場合は，

```
neuron = CurrentBasedLIF(N=1)
v = neuron.v_
```

のようにします．また，self.tcount はシミュレーションステップを記録する変数です．発火時刻を記録し，不応期かどうかの確認に用います．

1.2.3 LIF モデルの F-I curve

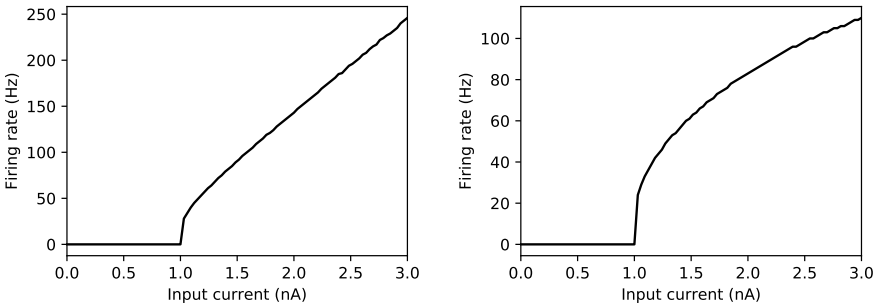
数値的計算による F-I curve の描画

この節では LIF モデルにおける入力電流の大きさに対して, 発火率の変化がどのように変化するかを考えます. 次のコードのように入力電流を徐々に増加させたときの発火率を見てみましょう[*22].

```
dt = 5e-5; T = 1; nt = round(T/dt)
tref = 5e-3; tc_m = 1e-2; vrest = 0; vreset = 0; vthr = 1

I_max = 3 # (nA)
N = 100 # N種類の入力電流
I = np.linspace(0, I_max, N) # 入力電流 (pA)
spikes = np.zeros((N, nt)) # スパイクの記録変数

for i in tqdm(range(N)):
    v = vreset; tlast = 0 # 初期化
    for t in range(nt):
        dv = (vrest - v + I[i]) / tc_m #  膜電位の導関数
        update = 1 if ((dt*t) > (tlast + tref)) else 0 # 不応期でないかの確認
        v = v + update*dv*dt # 膜電位の更新
        s = 1 if (v>=vthr) else 0 #発火時は 1, その他は 0 の出力
        tlast = tlast*(1-s) + dt*t*s # スパイク時刻の更新
        spikes[i, t] = s # 保存
        v = v*(1-s) + vreset*s # 膜電位のリセット

rate = np.sum(spikes, axis=1) / T # 発火率
plt.figure(figsize=(4, 3))
plt.plot(I, rate)
plt.xlabel('Input current (nA)'); plt.ylabel('Firing rate (Hz)')
plt.show()
```

[*22] コードは./SingleFileSimulations/Neurons/LIF_FI_curve_numerical.py です

結果は図 1.5 のようになります．このような曲線を **frequency-current (F-I) curve** (または neuronal input/output (I/O) curve) と呼びます．



▲ 図 1.5　LIF ニューロンの入力電流に対する発火率の関係 (F-I curve) を数値的に求めた結果．(左) 不応期がない場合．この場合は閾値付近以外は ReLU 関数のような挙動をします．また実際にこのような直線的な F-I curve を持つニューロンもあります (関連した考察は発火率モデルの項で行います)．(右) 不応期を 5 ms とした場合．さらに電流を強めると発火率は飽和 (saturation) します．

解析的計算による F-I curve の描画

ここまでは数値的なシミュレーションにより F-I curve を求めましたが, 以下では解析的に F-I curve の式を求めてみましょう．具体的には, 一定かつ持続的な入力電流を I としたときの LIF ニューロンの発火率 (firing rate) が

$$\text{rate} \sim \left(\tau_m \ln \frac{R_m I}{R_m I - V_{\text{th}}} \right) \tag{1.9}$$

と近似できることを示します．まず, $t = t_1$ にスパイクが生じたとします．このとき, 膜電位はリセットされるので $V_m(t_1) = 0$ です．$[t_1, t]$ における膜電位は LIF の式を積分することで得られます．

$$\tau \frac{dV_m(t)}{dt} = -V_m(t) + RI(t) \tag{1.10}$$

式 (1.10) を積分すると,

$$\int_{t_1}^{t} \frac{\tau dV_m}{RI - V_m} = \int_{t_1}^{t} dt \tag{1.11}$$

$$\ln(1 - \frac{V_m(t)}{RI}) = -\frac{t - t_1}{\tau} \quad (\because V_m(t_1) = 0) \tag{1.12}$$

$$\therefore \quad V_m(t) = RI \left[1 - \exp\left(-\frac{t - t_1}{\tau}\right)\right] \tag{1.13}$$

となります. $t > t_1$ における初めのスパイクが $t = t_2$ に生じたとすると, そのときの膜電位は $V_m(t_2) = V_{\rm th}$ です (実際には閾値以上となっている場合もありますが近似します). $t = t_2$ を上の式に代入して

$$V_{\rm th} = RI \left[1 - \exp\left(-\frac{t_2 - t_1}{\tau}\right)\right] \tag{1.14}$$

$$\therefore \quad T = t_2 - t_1 = \tau \ln \frac{RI}{RI - V_{\rm th}} \tag{1.15}$$

となります. ここで T は 2 つのスパイクの時間間隔です. $t_1 \leq t < t_2$ におけるスパイクは $t = t_1$ 時の 1 つなので, 発火率は $1/T$ となります. よって

$$\text{rate} \sim \left(\tau \ln \frac{RI}{RI - V_{\rm th}}\right) \tag{1.16}$$

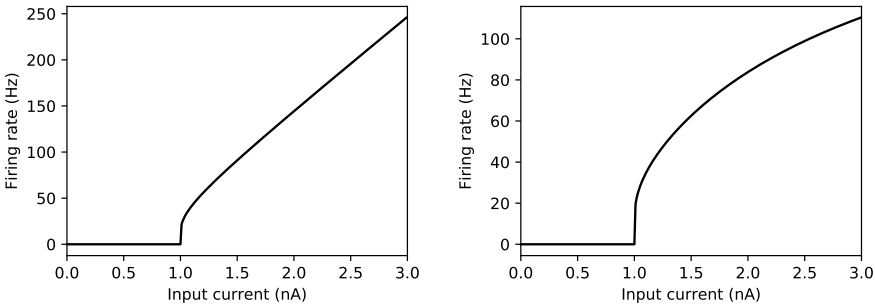
です. 不応期 $\tau_{\rm ref}$ を考慮すると, 持続的に入力がある場合は単純に $\tau_{\rm ref}$ だけ発火が遅れるので発火率は $1/(\tau_{\rm ref} + T)$ となります.

それでは式 (1.9) に基づいて F-I curve を描画してみましょう.

```
tc_m = 1e-2 # 膜時定数 (s)
R = 1 #膜抵抗
vthr = 1 # 閾値電位 (mV)
tref = 5e-3 # 不応期 (s)
I_max = 3 # 最大電流
I = np.arange(0, I_max, 0.01) #入力電流


rate = 1 / (tref + tc_m*np.log(R*I / (R*I - vthr)))
rate[np.isnan(rate)] = 0 # nan to 0

# 描画
plt.figure(figsize=(4, 3))
```

```
plt.plot(I, rate, color="k")
plt.xlabel('Input current (nA)'); plt.ylabel('Firing rate (Hz)')
plt.xlim(0, I_max)
plt.show()
```

なお, 閾値以下の場合は log の真数が負になるので `RuntimeWarning` が出ますが, 気にせずに実行しましょう. 結果は図 1.6 のようになります.



▲ 図 1.6　LIF ニューロンの入力電流に対する発火率の関係 (F-I curve) を解析的に求めた結果. (左) 不応期がない場合. (右) 不応期を 5 ms とした場合.

1.3 Izhikevich モデル

1.3.1　Izhikevich モデルの単純な実装

Izhikevich モデル (または **Simple model**) は (Izhikevich, 2003) で考案されたモデルです. HH モデルのような生理学的な知見に基づいたモデルは実際のニューロンの発火特性をよく再現できますが, 式が複雑化するために数学的な解析が難しくなります. また, 計算量が増えるために大規模なシミュレーションも困難となります. そこで, 生理学的な正しさには目をつぶり, 生体内でのニューロンの発火特性を再現するモデルが求められました. その特徴を持つのが Izhikevich モデルです (以下では Iz モデルと略します). Iz モデルは 2 変数しかない[*23]微分方程式ですが, 様々なニューロンの活動を模倣することが

[*23] if 文が入るためにシミュレーション上は簡易的ですが, 解析をする上では難しくなります. (Bernardo et al., 2008) を読むといいそうです.

できます.

$$C\frac{dv(t)}{dt} = k\left(v(t) - v_r\right)\left(v(t) - v_t\right) - u(t) + I \tag{1.17}$$

$$\frac{du(t)}{dt} = a\left\{b\left(v(t) - v_r\right) - u(t)\right\} \tag{1.18}$$

ここで, v と u が変数であり, v は膜電位 (membrane potential; 単位は mV), u は回復電流 (recovery current; 単位は pA)[*24]です. また, C は膜容量 (membrane capacitance; 単位は pF), v_r は静止膜電位 (resting membrane potential; 単位は mV), v_t は閾値電位 (instantaneous threshold potential; 単位は mV), a は回復時定数 (recovery time constant; 単位は ms^{-1}) の逆数 (これが大きいと u が元に戻る時間が短くなります), b は u の v に対する感受性 (共鳴度合い, resonance; 単位は pA/mV) です. k はニューロンのゲインに関わる定数で, 小さいと発火しやすくなります (単位は pA/mV).

Iz モデルの閾値の取り扱いは LIF モデルと異なり, HH モデルに近いです. LIF モデルでは閾値を超えた時に膜電位をピーク電位まで上昇させ (この過程は無くてもよいです), 続いて膜電位をリセットします. Iz モデルの閾値は v_t ですが, 膜電位のリセットは閾値を超えたかで判断せず, 膜電位 v がピーク電位 v_{peak} になったとき (または超えた時) に行います. そのため Iz モデルの実際の閾値は膜電位の挙動が変化する (発火状態に移行する), つまり分岐 (bifurcation) が生じる点であり, パラメータの閾値 v_t との間には差異があります.

さて, 膜電位がピーク電位 v_{peak} に達したとき (すなわち if $v \geq v_{\mathrm{peak}}$), u, v を次のようにリセットします[*25].

$$u \leftarrow u + d \tag{1.19}$$

$$v \leftarrow v_{\mathrm{reset}} \tag{1.20}$$

とします. ただし, v_{reset} は過分極を考慮して静止膜電位 v_r よりも小さい値とします. また, d はスパイク発火中に活性化される正味の外向き電流の合計を表し, 発火後の膜電位の挙動に影響します (単位は pA).

以上を踏まえて, シミュレーションの例と実装は次のようになります[*26]. ここで注意してほしいのが, 時間のスケールです. LIF モデルでは秒 (sec) のスケールでしたが, Iz モデ

[*24] ここでの「回復」というのは脱分極した後の膜電位が静止膜電位へと戻る, という意味です (対義語は activation で膜電位の上昇を意味します). u は v の導関数において v の上昇を抑制するように $-u$ で入っているため, u としては K^+ チャネル電流や Na^+ チャネルの不活性化動態などが考えられます.

[*25] バースト発火 (bursting) の挙動を表現するためには, 速い回復変数 (fast recovery variable) と遅い回復変数 (slow recovery variable) の 2 つが必要となります (従って膜電位も合わせて全部で 3 変数必要). 一方で, Iz モデルでは LIF モデルのような if 文によるリセットを用いているため, 速い回復変数が必要なく, 遅い回復変数 u のみでバースト発火を表現できます.

[*26] コードは`./SingleFileSimulations/Neurons/Izhikevich_single.py` です.

ルでは更新においてミリ秒のスケールを用います (LIF モデルの項で説明しましたが, これは元論文の (Izhikevich, 2003) に合わせています).

```
dt = 0.5; T = 500 # ms
nt = round(T/dt) # シミュレーションステップ数

# Regular spiking (RS) neurons
C = 100       # 膜容量 (pF)
a = 0.03      # 回復時定数の逆数 (1/ms)
b = -2        # u の v に対する共鳴度合い (pA/mV)
k = 0.7       # ゲイン (pA/mV)
d = 100       # 発火で活性化される正味の外向き電流 (pA)
vrest = -60   # 静止膜電位 (mV)
vreset = -50 # リセット電位 (mV)
vthr = -40    # 閾値電位 (mV)
vpeak = 35    #  ピーク電位 (mV)
t = np.arange(nt)*dt
I = 100*(t>50) - 100*(t>350) # 入力電流 (pA)

# 初期化 (膜電位, 膜電位 (t-1), 回復電流)
v = vrest; v_ = v; u = 0
v_arr = np.zeros(nt) # 膜電位を記録する配列
u_arr = np.zeros(nt) # 回復変数を記録する配列

# シミュレーション
for i in tqdm(range(nt)):
    dv = (k*(v - vrest)*(v - vthr) - u + I[i]) / C
    v = v + dt*dv # 膜電位の更新
    u = u + dt*(a*(b*(v_-vrest)-u)) # 膜電位の更新

    s = 1*(v>=vpeak) #発火時は 1, その他は 0 の出力

    u = u + d*s # 発火時に回復変数を上昇
    v = v*(1-s) + vreset*s # 発火時に膜電位をリセット
```

```
    v_ = v # v(t-1) <- v(t)

    v_arr[i] = v  # 膜電位の値を保存
    u_arr[i] = u  # 回復変数の値を保存

# 描画
t = np.arange(nt)*dt
plt.figure(figsize=(5, 5))
plt.subplot(2,1,1)
plt.plot(t, v_arr, color="k")
plt.ylabel('Membrane potential (mV)')

plt.subplot(2,1,2)
plt.plot(t, u_arr, color="k")
plt.xlabel('Time (ms)')
plt.ylabel('Recovery current (pA)')
plt.show()
```

結果は図 1.7 のようになります.

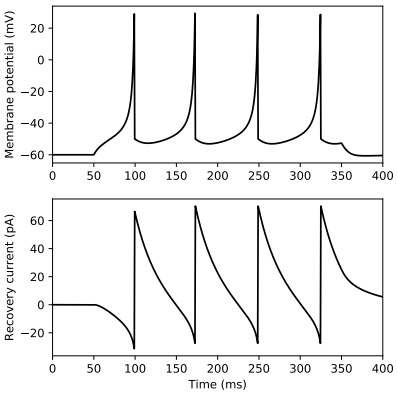
1.3.2 様々な発火パターンのシミュレーション

次に様々な発火パターンを模倣するように Iz モデルの定数を変化させてみましょう. Intrinsically Bursting (IB) ニューロンと Chattering (CH) ニューロン (または fast rhythmic bursting (FRB) ニューロン) のシミュレーションを行います. 基本的には定数を変えるだけです.

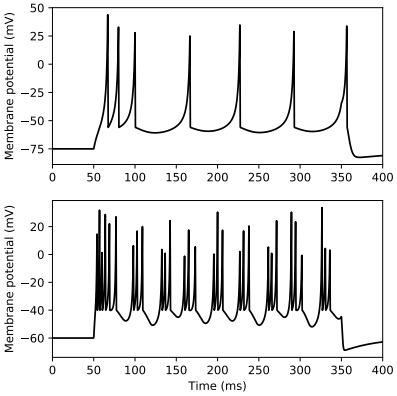
```
# Intrinsically Bursting (IB) neurons
C = 150; a = 0.01; b = 5; k =1.2; d = 130
vrest = -75; vreset = -56; vthr = -45; vpeak = 50;
I = 600*(t>50) - 600*(t>350)

# Chattering (CH) or fast rhythmic bursting (FRB) neurons
C = 50; a = 0.03; b = 1; k =1.5; d = 150
```

```
vrest = -60; vreset = -40; vthr = -40; vpeak = 35;
I = 600*(t>50) - 600*(t>350)
```

結果は図 1.8 のようになります.



▲ 図 1.7 Iz モデルにおける Regular spiking (RS) ニューロン. (上) 膜電位 v, (下) 回復電流 u



▲ 図 1.8 Iz モデルにおける Intrinsically Bursting (IB) ニューロン (上) と Chattering (CH) ニューロン (下) のシミュレーション

1.3.3 Izhikevich モデルの class の実装

最後に LIF モデルの際と同様に class を用いて Iz モデルの実装をしておきます[*27]. LIF モデルの場合と同様に `./Models/Neurons.py` に記述します.

```
class IzhikevichNeuron:
    def __init__(self, N, dt=0.5, C=250, a=0.01, b=-2,
                 k=2.5, d=200, vrest=-60, vreset=-65, vthr=-20, vpeak=30):
        self.N = N
        self.dt = dt
        self.C = C
```

[*27] コードは `./TrainingSNN/Models/Neurons.py` に含まれます.

```
        self.a = a
        self.b = b
        self.d = d
        self.k = k
        self.vrest = vrest
        self.vreset = vreset
        self.vthr = vthr
        self.vpeak = vpeak

        self.u = np.zeros(N)
        self.v = self.vrest*np.ones(N)
        self.v_ = self.v

    def initialize_states(self, random_state=False):
        if random_state:
            self.v = self.vreset + np.random.rand(self.N)\
                                 *(self.vthr-self.vreset)
        else:
            self.v = self.vrest*np.ones(self.N)
        self.u = np.zeros(self.N)

    def __call__(self, I):
        dv = (self.k*(self.v-self.vrest)*(self.v-self.vthr)-self.u+I)/self.C
        v = self.v + self.dt*dv
        u = self.u + self.dt*(self.a*(self.b*(self.v_-self.vrest)-self.u))

        s = 1*(v>=self.vpeak) #発火時は 1, その他は 0 の出力
        self.u = u + self.d*s
        self.v = v*(1-s) + self.vreset*s
        self.v_ = self.v
        return s
```

1.4 Inter-spike interval モデル

これまで紹介したモデルでは, 入力に対する膜電位などの時間変化に基づき発火が起こるかどうか, ということを考えてきました. この節では, 発火が生じるまでの過程を考慮せず, 発火の時間間隔 (**inter-spike interval, ISI**) の統計による現象論的モデルを考えます (これを **Inter-spike interval (ISI)** モデルと呼びます). ISI モデルは**点過程 (point process)** という統計的モデルに基づいており, 各モデルには ISI が従う分布の名称がついています. この節では, 使用頻度の高い**ポアソン過程 (Poisson process) モデル**, ポアソン過程モデルにおいて不応期を考慮した**死時間付きポアソン過程 (Poisson process with dead time, PPD) モデル**, 皮質の定常発火においてポアソン過程モデルよりも当てはまりがよいとされる**ガンマ過程 (Gamma process) モデル**について説明します.[*28]

なお, SNN において, ISI モデルは主に画像入力の際に**連続値からスパイク列への符号化 (encoding)** に用いられます.

1.4.1 ポアソン過程モデル

点過程とポアソン過程

時間に応じて変化する確率変数のことを**確率過程 (stochastic process)** といいます. さらに確率過程の中で, 連続時間軸上において離散的に生起する点事象の系列を**点過程 (point process)** といいます. スパイクは離散的に起こるので, 点過程を用いてモデル化ができるという話です.

ポアソン過程 (Poisson process) は点過程の 1 つです. ポアソン過程モデルはスパイクの発生をポアソン過程でモデル化したもので, このモデルによって生じるスパイクをポアソンスパイク (Poisson spike) と呼びます. ポアソン過程では, 時刻 t までに起こった点の数 $N(t)$ はポアソン分布に従います. すなわち, 点が起こる確率が強度 λ のポアソン分布に従う場合, 時刻 t までに事象が n 回起こる確率は $P[N(t)=n]=\dfrac{(\lambda t)^n}{n!}e^{-\lambda t}$ となります.

ポアソン過程において点が起こる回数がポアソン分布に従うことは, ポアソン過程という名称の由来です. これを定義とする場合もあれば, 次の 4 条件を満たす点過程をポアソン過程とするという定義もあります.

1. 時刻 0 における初期の点の数は 0 :
   $P[N(0)=0]=1$
2. $[t, t+\Delta t)$ に点が 1 つ生じる確率 :

[*28] この節は (島崎, “スパイク統計モデル入門”; Pachitariu, “Probabilistic models for spike trains of single neurons”) を主に参考にしました.

$P[N(t+\Delta t)-N(t)=1]=\lambda(t)\Delta t+o(\Delta t)$

3. 微小時間 Δt の間に点は 2 つ以上生じない：
   $P[N(t+\Delta t)-N(t)=2]=o(\Delta t)$
4. 任意の時点 $t_1<t_2<\cdots<t_n$ に対して，増分
   $N(t_2)-N(t_1), N(t_3)-N(t_2), \cdots, N(t_n)-N(t_{n-1})$ は互いに独立である．

ただし，$o(\cdot)$ は Landau の記号 (Landau の small o) であり，$o(x)$ は $x\to 0$ のとき，$o(x)/x\to 0$ となる微小な量を表します．ポアソン過程に従ってスパイクが生じるとする場合，条件 2 の強度関数 $\lambda(t)$ は**発火率**を意味します (また実装において有用)．条件 3 は不応期より短いタイムステップにおいては，1 つのシミュレーションステップにおいて 1 つしかスパイクは生じないということを表します．条件 4 はスパイクは独立に発生する，ということを意味します．また，これらの条件から $N(t)$ の分布は強度母数 $\lambda(t)$ のポアソン分布に従うことが示せます．

強度関数 (点がスパイクの場合，発火率) が $\lambda(t)=\lambda$ (定数) となる場合は点の時間間隔 (点がスパイクの場合，ISI) の確率変数 T が強度母数 λ の**指数分布**に従います．なお，指数分布の確率密度関数は確率変数を T とするとき，

$$f(t;\lambda)=\begin{cases}\lambda e^{-\lambda t} & (t\geq 0)\\ 0 & (t<0)\end{cases} \tag{1.21}$$

となります．このことは 4 条件と Chapman-Kolmogorov の式により求められますが，ややこしいので，$P[N(t)=n]=\dfrac{(\lambda t)^n}{n!}e^{-\lambda t}$ から導出できることを簡単に示します．指数分布の累積分布関数を $F(t;\lambda)$ とすると，

$$F(t;\lambda)=P(T<t)=1-P(T>t)=1-P(N(t)=0)=1-e^{-\lambda t} \tag{1.22}$$

となります．よって

$$f(t;\lambda)=\frac{dF(t;\lambda)}{dt}=\lambda e^{-\lambda t} \tag{1.23}$$

となります．

定常ポアソン過程

ここからポアソン過程によるスパイクのシミュレーションを実装していきます．実装方法には ISI が指数分布に従うことを利用したものと，ポアソン過程の条件 2 を利用したものの 2 通りがあります．実装は後者が楽で計算量も少ないですが，後のガンマ過程のために前者の実装を先に行います．

ISI が指数分布に従うことを利用してポアソン過程モデルの実装を行います．SNN のシミュレーションにおいてスパイク列 S はシミュレーションステップごとに発火し

ているかの $\{0,1\}$ 配列で保持しておくと楽に扱うことができます[*29]. 手順としては, まず ISI を指数分布に従う乱数とします. 次に ISI を累積することで発火時刻を得ます. 最後に発火時間を整数値に丸めて index とすることで $\{0,1\}$ のスパイク列が得られます. ISI の取得には `numpy.random.exponential()` を用います. ややこしいのですが, この関数は引数として `scale` と `size` を受け取りますが, この `scale` は指数分布の確率密度関数を $f(t;\frac{1}{\beta}) = \frac{1}{\beta}e^{-t/\beta}$ とした際の $\beta = 1/\lambda$ です (この時, 平均は β となります). よって発火率を `fr`(1/s), 単位時間を `dt`(s) としたときの ISI は `isi = np.random.exponential(1/(fr*dt))` として得ることができます. それでは実装してみましょう[*30].

```
dt = 1e-3; T = 1; nt = round(T/dt) # シミュレーション時間
n_neurons = 10 # ニューロンの数
fr = 30 # ポアソンスパイクの発火率 (Hz)
isi = np.random.exponential(1/(fr*dt), size=(round(nt*1.5/fr), n_neurons))
spike_time = np.cumsum(isi, axis=0) # ISI を累積
spike_time[spike_time > nt - 1] = 0 # nt を超える場合を 0 に
spike_time = spike_time.astype(np.int32) # float to int
spikes = np.zeros((nt, n_neurons)) # スパイク記録変数
for i in range(n_neurons):
    spikes[spike_time[:, i], i] = 1
spikes[0] = 0 # (spike_time=0) の発火を削除
print("Num. of spikes:", np.sum(spikes))
print("Firing rate:", np.sum(spikes)/(n_neurons*T))
# 描画
t = np.arange(nt)*dt
plt.figure(figsize=(5, 4))
for i in range(n_neurons):
    plt.plot(t, spikes[:, i]*(i+1), 'ko', markersize=2)
plt.xlabel('Time (s)'); plt.ylabel('Neuron index')
plt.xlim(0, T); plt.ylim(0.5, n_neurons+0.5)
plt.show()
```

[*29] もちろん発火時刻で記録しておく方がメモリを節約できますが, 計算量が多くなります.

[*30] コードは./SingleFileSimulations/ISI/poisson_process.py です.

結果は図 1.9 のようになります．図 1.9 は各ニューロンが発火したことを点で表しており，このような図を**ラスタープロット (raster plot)** といいます．

次にポアソン過程モデルの 2 番目の方法の実装を行います．こちらは λ を発火率とした場合，区間 $[t, t+\Delta t]$ の間にポアソンスパイクが発生する確率は $\lambda\Delta t$ となることを利用します．これはポアソン過程の条件ですが，ポアソン分布から導けることを簡単に示しておきます．事象が起こる確率が強度 λ のポアソン分布に従う場合，時刻 t までに事象が n 回起こる確率は $P[N(t)=n] = \dfrac{(\lambda t)^n}{n!}e^{-\lambda t}$ となります．よって，微小時間 Δt において事象が 1 回起こる確率は

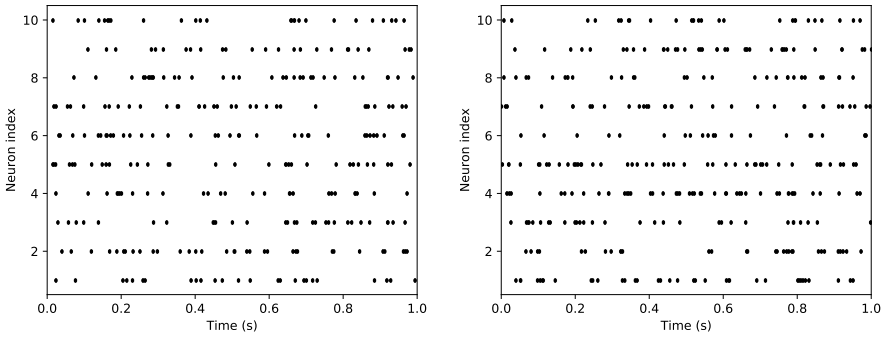
$$P[N(\Delta t)=1] = \frac{\lambda\Delta t}{1!}e^{-\lambda\Delta t} \simeq \lambda\Delta t + o(\Delta t) \tag{1.24}$$

となります．ただし，$e^{-\lambda\Delta t}$ についてはマクローリン展開による近似を行っています．このことから，一様分布 $U(0,1)$ に従う乱数 ξ を取得し，$\xi < \lambda dt$ なら発火 $(y=1)$，それ以外では $(y=0)$ となるようにすればポアソンスパイクを実装できます[*31]．

```
dt = 1e-3; T = 1; nt = round(T/dt) # シミュレーション時間
n_neurons = 10 # ニューロンの数
fr = 30 # ポアソンスパイクの発火率 (Hz)

# スパイク記録変数
spikes = np.where(np.random.rand(nt, n_neurons) < fr*dt, 1, 0)

print("Num. of spikes:", np.sum(spikes))
print("Firing rate:", np.sum(spikes)/(n_neurons*T))
# 描画
t = np.arange(nt)*dt
plt.figure(figsize=(5, 4))
for i in range(n_neurons):
    plt.plot(t, spikes[:, i]*(i+1), 'ko', markersize=2)
plt.xlabel('Time (s)'); plt.ylabel('Neuron index')
plt.xlim(0, T); plt.ylim(0.5, n_neurons+0.5)
plt.show()
```

結果は図 1.9 のようになります．

[*31] コードは./SingleFileSimulations/ISI/poisson_process_fast.py です．



▲ 図 1.9　ポアソン過程モデルによる 10 個のニューロンのスパイクの 1 秒間のシミュレーション (ラスタープロット). (左)ISI の累積により発火時刻を求める手法. (右)Δt 間の発火確率が $\lambda\Delta t$ であることを利用する方法. 結果はほぼ同じですが, 実行速度は右の方が速いです.

なお, ここでは全時間における発火をまとめて計算していますが, シミュレーションステップごとに発火の有無を計算することもできます. 前者は発火情報を保持するためのメモリが必要ですが, 計算時間は短くて済みます. 後者はメモリの節約になりますが, 計算時間は長くなります. そのため, これら 2 つの方法はメモリと計算時間のトレードオフとなります. また, 他には発火情報を疎行列 (sparse matrix) の形式で保持しておくとメモリの節約になると思われます.

非定常ポアソン過程

これまでの実装は発火率 λ が一定であるとする, **定常ポアソン過程 (homogeneous poisson process)** でしたが, ここからは発火率 $\lambda(t)$ が時間変化する, **非定常ポアソン過程 (inhomogeneous poisson process)** のシミュレーションを行います. とはいえ, 非定常ポアソン過程は定常ポアソン過程における発火率を, 時間についての関数で置き換えるだけで実装できます. 以下は $\lambda(t) = \sin^2(\alpha t)$(ただし α は定数) とした場合の実装です[*32].

```
dt = 1e-3; T = 1; nt = round(T/dt) # シミュレーション時間
n_neuron = 10 # ニューロンの数
t = np.arange(nt)*dt
```

[*32] コードは./SingleFileSimulations/ISI/inhomogeneous_poisson_process.py です.

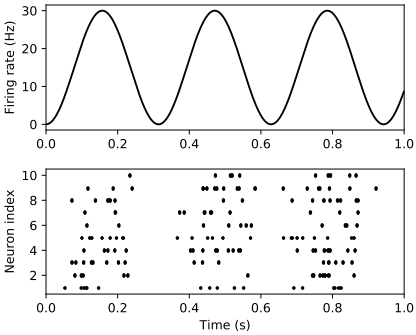
```
# ポアソンスパイクの発火率 (Hz)
fr = np.expand_dims(30*np.sin(10*t)**2, 1)

# スパイク記録変数
spikes = np.where(np.random.rand(nt, n_neuron) < fr*dt, 1, 0)

print("Num. of spikes:", np.sum(spikes))
# 描画
plt.figure(figsize=(5, 4))
plt.subplot(2,1,1)
plt.plot(t, fr[:, 0], color="k")
plt.ylabel('Firing rate (Hz)')
plt.xlim(0, T)

plt.subplot(2,1,2)
for i in range(n_neuron):
    plt.plot(t, spikes[:, i]*(i+1), 'ko', markersize=2,
             rasterized=True)
plt.xlabel('Time (s)'); plt.ylabel('Neuron index')
plt.xlim(0, T); plt.ylim(0.5, n_neuron+0.5)
plt.show()
```

結果は図 1.10 のようになります.

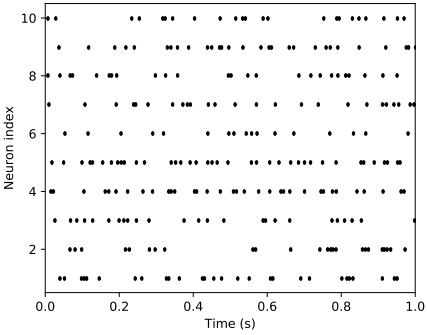
1.4.2 死時間付きポアソン過程モデル (PPD)

ポアソン過程は簡易的で有用ですが, 不応期を考慮していません. そのため, 時には生理的範疇を超えたバースト発火が起こる場合もあります. そこで, ポアソン過程において不応期のようなイベントの生起が起こらない**死時間 (dead time)**[*33] を考慮した**死時間付きポアソン過程 (Poisson process with dead time, PPD)** というモデルを導入します.

[*33] 例えば, ガイガー・カウンター (Geiger counter) などの放射線の検出器には放射線の到達を機器の物理的特性として検出できない時間 (つまり死時間) があります. そのため放射線の到達数がポアソン分布に従うとした場合, 放射線測定装置のモデルとして PPD が用いられます.



▲ 図 1.10　非定常ポアソン過程による 10 個のニューロンのスパイクの 1 秒間のシミュレーション. (上) 発火率 $\lambda(t)$ の時間変化, (下) ラスタープロット.



▲ 図 1.11　死時間付きポアソン過程 (PPD) による 10 個のニューロンのスパイクの 1 秒間のシミュレーション. 高頻度発火の場合に通常のモデルとの違いが明瞭となる.

実装においては LIF ニューロンの時と同じような不応期の処理をします. つまり, 現在が不応期かどうかを判断し, 不応期なら発火を許可しないようにします[*34].

```
dt = 1e-3; T = 1; nt = round(T/dt) # シミュレーション時間
n_neurons = 10 # ニューロンの数
tref = 5e-3 # 不応期 (s)
fr = 30 # ポアソンスパイクの発火率 (Hz)
spikes = np.zeros((nt, n_neurons)) #スパイク記録変数
tlast = np.zeros(n_neurons) # 発火時刻の記録変数
for i in range(nt):
    s = np.where(np.random.rand(n_neurons) < fr*dt, 1, 0)
    spikes[i] = ((dt*i) > (tlast + tref))*s
    tlast = tlast*(1-s) + dt*i*s # 発火時刻の更新

print("Num. of spikes:", np.sum(spikes))
print("Firing rate:", np.sum(spikes)/(n_neurons*T))
# 描画
```

[*34] コードは./SingleFileSimulations/ISI/PPD.py です.

```
t = np.arange(nt)*dt
plt.figure(figsize=(5, 4))
for i in range(n_neurons):
    plt.plot(t, spikes[:, i]*(i+1), 'ko', markersize=2)
plt.xlabel('Time (s)'); plt.ylabel('Neuron index')
plt.xlim(0, T); plt.ylim(0.5, n_neurons+0.5)
plt.show()
```

これの実行結果は図 1.11 のようになります.

1.4.3 ガンマ過程モデル

ガンマ過程 (gamma process) は点の時間間隔がガンマ分布に従うとするモデルです. ガンマ過程はポアソン過程よりも皮質における定常発火への当てはまりが良いとされています (Shinomoto et al., 2003; Maimon & Assad, 2009).

時間間隔の確率変数を T とした場合, ガンマ分布の確率密度関数は

$$f(t;k,\theta) = t^{k-1}\frac{e^{-t/\theta}}{\theta^k \Gamma(k)} \tag{1.25}$$

と表されます. ただし, $t > 0$ であり, 2 つの母数は $k, \theta > 0$ です. また, $\Gamma(\cdot)$ はガンマ関数であり,

$$\Gamma(k) = \int_0^\infty x^{k-1}e^{-x}\,dx \tag{1.26}$$

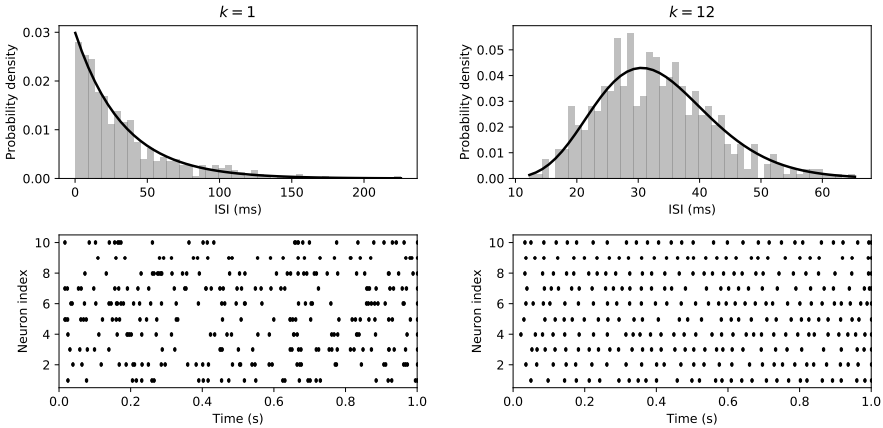
と定義されます. ガンマ分布の平均は $k\theta$ ですが, 発火率は ISI の平均の逆数なので, $\lambda = 1/k\theta$ となります. また, $k = 1$ のとき, ガンマ分布は指数分布となります. さらに k が正整数のとき, ガンマ分布はアーラン分布となります.

ガンマ過程モデルの実装はポアソン過程モデルの ISI を累積する手法と同様に書くことができ, `numpy.random.exponential()` を `numpy.random.gamma()` に置き換えるだけです (もちろん多少の修正は必要とします). また, ガンマ分布の描画にガンマ関数を用いるので, `import scipy.special as sps` を実行して `scipy.special` を import することで, ガンマ関数の関数 `sps.gamma()` を使用できるようにします[*35].

[*35] コードは./SingleFileSimulations/ISI/gamma_process.py です.

```
import scipy.special as sps
dt = 1e-3; T = 1; nt = round(T/dt) # シミュレーション時間
n_neurons = 10 # ニューロンの数
fr = 30 # ガンマスパイクの発火率 (Hz)
k = 12 # k=1 のときはポアソン過程に一致
theta = 1/(k*(fr*dt)) # fr = 1/(k*theta)
isi = np.random.gamma(shape=k, scale=theta, size=(int(nt*1.5/fr), n_neurons))
spike_time = np.cumsum(isi, axis=0) # ISI を累積
spike_time[spike_time > nt - 1] = -1 # nt を超える場合を 0 に
spike_time = spike_time.astype(np.int32) # float to int
spikes = np.zeros((nt, n_neurons)) # スパイク記録変数
for i in range(n_neurons):
    spikes[spike_time[:, i], i] = 1
spikes[0] = 0 # (spike_time=0) の発火を削除
print("Num. of spikes:", np.sum(spikes))
print("Firing rate:", np.sum(spikes)/(n_neurons*T))

t = np.arange(nt)*dt
plt.figure(figsize=(5, 5))
plt.subplot(2,1,1) # ISI のヒストグラム
count, bins, ignored = plt.hist(isi.flatten(), 50, density=True,
                                color="gray", alpha=0.5)
y = bins**(k-1)*(np.exp(-bins/theta) / (sps.gamma(k)*theta**k))
plt.plot(bins, y, linewidth=2, color="k")
plt.title('k='+str(k)); plt.xlabel('ISI (ms)')
plt.ylabel('Probability density')

plt.subplot(2,1,2) # ラスタープロット
for i in range(n_neurons):
    plt.plot(t, spikes[:, i]*(i+1), 'ko', markersize=2)
plt.xlabel('Time (s)'); plt.ylabel('Neuron index')
plt.xlim(0, T); plt.ylim(0.5, n_neurons+0.5)
plt.show()
```

結果は図 1.12 のようになります.



▲ 図 1.12　ガンマ過程モデルによる 10 個のニューロンのスパイクの 1 秒間のシミュレーション. 上段は ISI の分布 (横軸の単位が ms であることに注意). 下段はラスタープロット. 左の $k = 1$ の場合はポアソン過程モデルと同じです. 右は $k = 12$ とした場合です.

なお, 前述したようにガンマ過程モデルの方がポアソン過程モデルよりも皮質ニューロンのモデルとしては優れていますが, 入力画像のスパイク列への符号化 (encoding) をガンマ過程モデルにすることで SNN の認識精度が向上するかどうかはまだ十分に研究されていません. また, (Deger et al., 2012) では PPD やガンマ過程の重ね合わせによるスパイク列を生成するアルゴリズムを考案しています.

1.5 発火率モデル

発火率モデル (firing rate model) は Spike-based ではありませんが, 重要なモデルなので説明をしておきます. 簡潔に言えば, 発火率モデルではニューロンの活動をスパイクではなく発火率で表現します.

1.5.1 発火率

発火率モデルの前に**発火率 (firing rate)** について簡単に説明をしておきます. 発火率は, 単位時間あたりにニューロンが発火した回数のことです. 発火率を r, 時間窓 (time window) を T, 発火回数を

$$n = \int_0^T dt\ \delta(t - t_k) \quad (\delta\text{は Dirac のデルタ関数}, \quad t_k\text{はスパイクの発生時刻}) \tag{1.27}$$

とすると, 発火率は $r := n/T$ と表されます.

1.5.2 離散時間発火率モデル

発火率モデルは, 神経生理学者で外科医でもあった W. MuCulloch と, 論理学者であった W. Pitts によって考案された**形式ニューロン (formal neuron)** または **McCulloch-Pitts ニューロン**に端を発します (McCulloch & Pitts, 1943). さらに心理学者であった F. Rosenblatt により**パーセプトロン (perceptron)** が考案されました (Rosenblatt, 1958). 種々の修正はありますが, 離散時間発火率モデル[*36]は次のように定式化されます.

$$\boldsymbol{r} = f(W\boldsymbol{u} + \boldsymbol{b}) \tag{1.28}$$

ここで, $\boldsymbol{u} \in \mathbb{R}^{N_{\text{pre}}}$ はシナプス前細胞の発火率, $\boldsymbol{r} \in \mathbb{R}^{N_{\text{post}}}$ はシナプス後細胞の発火率, $\boldsymbol{b} \in \mathbb{R}^{N_{\text{post}}}$ はバイアス, W はシナプス強度を表す $N_{\text{post}} \times N_{\text{pre}}$ 行列です (ただし N_{pre} はシナプス前細胞の数, N_{post} はシナプス後細胞の数です). そして, $f(\cdot)$ は活性化関数で, シグモイド関数 (sigmoid) のような非線形関数 (nonlinear function) が使われます[*37]. 大

[*36] 一般に, 人工神経回路 (Artificial neural networks; ANN) というと発火率モデルを指します.

[*37] 非線形関数として昔からシグモイド関数が用いられてきたのは, ニューロンの発火特性に類似しているためです. ニューロンには不応期があるため, 入力刺激を大きくすると発火率は飽和 (saturation, 俗に言う「サチる」) します. このことは LIF ニューロンにおいて不応期を変化させ, F-I curve を描くことでも分かります. また, ReLU 関数は (non-leaky な)IF ニューロンの F-I curve とほぼ一致することも知られています. こうした飽和しない関数はシグモイド関数よりも生理学的ではない, とするのが一般的です. しかし, 個人的にシグモイド関数が ReLU 関数よりも生理学的であるというのはおかしいと考えています. まず, F-I curve は全てのニューロンにおいて同一ではありません. *in vivo* で発火率が飽和するニューロンもあれば, 飽和しない, 又はやや飽和 (ややサチ) 止まりのニューロンもあります. 人工的に電

事なこととして, **活性化関数は F-I curve を表します**. F-I curve については LIF ニューロンの節を参照してください.

1.5.3 連続時間発火率モデル

次に微分方程式で与えられる[*38], 連続時間発火率モデル (continuous-time firing rate model) は H.R.Wilson と J.D.Cowan によるモデル (**Wilson-Cowan model**) に基づきます (Wilson & Cowan, 1972). 連続時間発火率モデルの一種である, 再帰結合を持つ連続時間型 RNN(continuous-time recurrent neural network) の式は次のようになります.

$$\tau \frac{d\boldsymbol{r}}{dt} = -\boldsymbol{r} + f(W^{\mathrm{rec}}\boldsymbol{r} + W^{\mathrm{in}}\boldsymbol{u} + \boldsymbol{b}) \tag{1.29}$$

ただし, τ は発火率の時定数 (膜電位の時定数ではない), $\boldsymbol{u} \in \mathbb{R}^{N_{\mathrm{pre}}}$ はシナプス前細胞の発火率, $\boldsymbol{r} \in \mathbb{R}^{N_{\mathrm{post}}}$ はシナプス後細胞の発火率, $\boldsymbol{b} \in \mathbb{R}^{N_{\mathrm{post}}}$ はバイアスです. 他にはノイズが入力されることもあります (正則化の効果があります). また, $f(\cdot)$ は離散モデルと同様の活性化関数 (発火率応答関数) です. 離散時間モデルと比べると, 入力に対する応答が時定数 τ だけ遅れて生じます.

実装する際には式 (1.29) を 1 次 Euler 近似 (first-order Euler approximation) を用いてタイムステップ Δt で離散化します.

$$\boldsymbol{r}_t = (1-\alpha)\boldsymbol{r}_{t-1} + \alpha f(W^{\mathrm{rec}}\boldsymbol{r}_{t-1} + W^{\mathrm{in}}u_t + \boldsymbol{b}) \tag{1.30}$$

となります (離散化後は **leaky RNN** と呼ばれるモデルの一種となります). また, これは t を層の数としたときの ResNet(He et al., 2016) の形状に類似しており[*39], $(1-\alpha)\boldsymbol{r}_{t-1}$

気刺激をすると全てのニューロンの発火率は飽和しますが, その電気刺激が生理学的な範囲にあるとは限りません. 同様の意見の論文が (Bhumbra, 2018) であり, この論文は前述した理由から bionodal root unit (BRU) というやや飽和な活性化関数を提案しています. Introduction では『*in vivo* のニューロンは実験的には発火率は飽和するが, 必要な脱分極の大きさは生理学的範囲を超えている. これはコンピュータの桁限界のために ReLU が飽和する, ということと同じくらい有益な科学的考察ではない.』という意見が述べられています. この言葉は強すぎるので, 完全に同意をするわけではありませんが, いずれにせよシグモイド関数だから生理学的というのは慎重になるべきではないでしょうか.

[*38] Neural Ordinary Differential Equations (ODENet; Chen et al., 2018) も微分方程式で与えられる点は共通していますが, 連続時間発火率モデルは通常離散化した後, Backpropagation through time (BPTT) でパラメータの更新をしますが, ODENet では adjoint method という手法を用いてパラメータ更新を行います. ODENet が (連続時間)RNN よりも優れている点としては不均一なタイムステップの入力に対しても推論が高精度で行える, などがあります.

[*39] (Liao & Poggio, 2016) は, 重み共有した ResNet が RNN と等価であるということについて述べています. この論文では ResNet は視覚野における再帰的計算に似た処理を行うことで物体認識の精度を上げているのかもしれないと主張しています. 関連する論文として (Kar, et al., 2019) では画像刺激をされた再帰結合を持つ畳み込みニューラルネットワーク (Recurrent convolutional neural network) の応答がアカゲザル (Macaca mulatta) の IT 野の応答を通常の CNN よりもよく説明できるということが述べられています.

の部分はショートカット経路となっています. 実際, この形式ではショートカット経路により BPTT における勾配消失や発散が起こりにくくなります (他には, LSTM や GRU にあるようなゲート構造も勾配消失・発散問題を解決します).

この連続時間型 RNN (leaky RNN) のシミュレーションについては触れませんが, 近年の RNN のシミュレーションでは BPTT を用いて学習させるのが基本的なので, Python のディープラーニングのフレームワークにおける実装例 (Chainer-v6 による[*40]) を掲載します[*41].

```
import chainer
import chainer.functions as F
import chainer.links as L
from chainer import cuda
from chainer import Variable
xp = cuda.cupy
# import numpy as xp

class leakyRNN(chainer.Chain):
    def __init__(self, inp=32, mid=128, alpha=0.2, sigma_rec=0.1):
        super(leakyRNN, self).__init__()
        """
        Leaky RNN unit.

        Usage:
            >>> network = leakyRNN()
            >>> network.reset_state()
            >>> x = xp.ones((1, 32)).astype(xp.float32)
            >>> y = network(x)
            >>> y.shape
                (1, 128)
        """
```

[*40] Pytorch じゃないのかという声が上がりそうですが, 時間がなかったので過去に自分が Chainer でシミュレーションしたコードから引用しました.

[*41] コードは./SingleFileSimulations/Neurons/leakyRNN_chainer.py です.

```
        with self.init_scope():
            self.Wx = L.Linear(inp, mid) # feed-forward
            self.Wr = L.Linear(mid, mid, nobias=True) # recurrent

            self.inp = inp # 入力ユニット数
            self.mid = mid # 出力ユニット数
            self.alpha = alpha # tau / dt
            self.sigma_rec = sigma_rec # standard deviation of input noise

    def reset_state(self, r=None):
        self.r = r

    def initialize_state(self, shape):
        self.r = Variable(xp.zeros((shape[0], self.mid),
                                   dtype=self.xp.float32))

    def forward(self, x):
        if self.r is None:
            self.initialize_state(x.shape)

        z = self.Wr(self.r) + self.Wx(x)
        if self.sigma_rec is not None:
            z += xp.random.normal(0, self.sigma_rec,
                                  (x.shape[0], self.mid)) # Add noise
        r = (1 - self.alpha)*self.r + self.alpha*F.relu(z)

        self.r = r
        return r
```

ここでは cupy により GPU で計算するコードを計算しています．CPU で計算する場合は cupy を numpy で置き換えてください．

コラム：確率的シナプス電流のノイズによる表現

脳内はノイズ (neuronal noise)[a]に溢れており, それを考慮してシナプス入力にノイズを加える場合があります. 例として LIF モデルの入力にノイズが加わる場合を考えます. 正規分布 $\mathcal{N}(\tilde{\mu}, \tilde{\sigma}^2)$ に従うノイズ (Gaussian noise) を $\xi(t)$ とすると,

$$\tau_m \frac{dV_m(t)}{dt} = -(V_m(t) - V_{\rm rest}) + R_m I(t) + \xi(t) \tag{1.31}$$

となります. このような線形のドリフト項 $(-V_m(t))$ とガウシアンノイズ項のある**確率微分方程式** (stochastic differential equations) で与えられる確率過程を **Ornstein-Uhlenbeck (OU) 過程**と言います. $\xi(t)$ は標準正規分布 $\mathcal{N}(0,1)$ に従うノイズ (Gaussian white noise) を $\eta(t)$ とした場合, $\xi(t) = \tilde{\mu} + \tilde{\sigma}\eta(t)$ と表すこともできます.

ノイズ項 $\xi(t)$ が発火率 λ のポアソン過程に従う場合は, シナプス前細胞の数を $N_{\rm pre}$ とし, i 番目のシナプスにおけるシナプス強度に比例する定数を J_i とするとき, $\tilde{\mu} = \langle J_i \rangle N_{\rm pre} \cdot \lambda$, $\tilde{\sigma}^2 = \langle J_i^2 \rangle N_{\rm pre} \cdot \lambda$ と表せます. ただし, $\langle \cdot \rangle$ は平均を取ることを意味します. これを**拡散近似** (Diffusion approximation) と言います. このことは Campbell の定理により求められます.

式 1.31 をシミュレーションのために離散化する際には注意が必要です. 右辺をノイズ項のみ残すと

$$\tau_m \frac{dV_m(t)}{dt} = \xi(t) \tag{1.32}$$

となります. これをタイムステップ Δt により Euler 法で離散化すると,

$$V_m(t + \Delta t) = V_m(t) + \frac{1}{\tau_m}\xi_1(t) \tag{1.33}$$

となります (このように確率微分方程式を Euler 法で離散化する方法を **Euler–Maruyama 法**と呼びます[b]). ここで, Δt を半分の $\Delta t/2$ にする場合, 同様に

$$\begin{aligned} V_m(t + \Delta t) &= V_m(t + \Delta t/2) + \frac{1}{\tau_m}\xi_1(t) \\ &= V_m(t) + \frac{1}{\tau_m}\left[\xi_1(t) + \xi_2(t)\right] \end{aligned} \tag{1.34}$$

となります.

式 (1.33) と式 (1.34) のノイズ項の標準偏差はそれぞれ $\tilde{\sigma}/\tau_m$, $\sqrt{2}\tilde{\sigma}/\tau_m$ となります[c]. これはタイムステップの変化でノイズ項の標準偏差が変化することを意味しますが, これを避けるためにノイズ項を上手く係数倍することを考えます.

このためには, ノイズ項に $\sqrt{\Delta t}$ を乗じれば良いことが分かります. すなわち, 式 (1.33) を

$$V_m(t+\Delta t) = V_m(t) + \frac{\sqrt{\Delta t}}{\tau_m}\xi_1(t) \tag{1.35}$$

と修正すればよいです.

---

[a] Scholarpedia の “Neuronal noise” (`http://www.scholarpedia.org/article/Neuronal_noise`) を参照してください.

[b] 確率微分方程式のシミュレーションのための方法としては, 他に **Milstein 法**などがあります.

[c] 各ノイズは独立であるので $\xi_1(t) + \xi_2(t)$ の分散は $2\tilde{\sigma}$ となります.

第 2 章

シナプスのモデル

スパイクが生じたことによる膜電位変化は軸索を伝播し, **シナプス**という構造により, 次のニューロンへと興奮が伝わります. このときの伝達の仕組みとして, シナプスには**化学シナプス** (chemical synapse) と Gap junction による**電気シナプス** (electrical synapse) があります. 中枢神経系には両方存在しますが, 今回は化学シナプスのみを考えます. 化学シナプスの場合, シナプス前膜からの**神経伝達物質**の放出, シナプス後膜の受容体への神経伝達物質の結合, イオンチャネル開口による**シナプス後電流** (postsynaptic current; **PSC**) の発生, という過程が起こります[*1]. そのため, シナプス前細胞のスパイク列 (spike train) は次のニューロンにそのまま伝わるのではなく, ある種の時間的フィルターをかけられて伝わります[*2]. この章では, このようにシナプス前細胞で生じた発火が, シナプス後細胞の膜電位に与える過程のモデルについて説明します.

2.1 Current-based vs Conductance-based シナプス

具体的なシナプスのモデルの前に, この節ではシナプス入力 (synaptic drive) の 2 つの形式, **Current-based シナプス**と **Conductance-based シナプス**について説明します. 簡単に説明すると, Current-based シナプスは入力電流が変化するというモデルで, Conductance-based シナプスはイオンチャネルのコンダクタンス (電流の流れやすさ) が変化するというモデルです (cf. Cavallari et al., 2014).

以下では例として, 次の LIF ニューロンの方程式におけるシナプス入力を考えます.

$$\tau_m \frac{dV_m(t)}{dt} = -(V_m(t) - V_{\rm rest}) + R_m I_{\rm syn}(t) \tag{2.1}$$

[*1] かなり簡略化して書きましたが, 実際にはかなりの過程を含みます. しかし, これらの過程を全てモデル化するのは計算量がかなり大きくなるので, 基本的には簡易的な現象論的なモデルを用います.

[*2] このフィルターを**シナプスフィルター** (synaptic filter) と呼びます

とします. ただし, τ_m は膜電位の時定数, $V_m(t)$ は膜電位, $V_{\rm rest}$ は静止膜電位, R_m は膜抵抗です. 最後にシナプス入力の電流 $I_{\rm syn}(t)$ ですが[*3], ここが 2 つのモデルにおいて異なってきます.

2.1.1 Current-based シナプス

Current-based シナプスは単純に**入力電流が変化**するというモデルで, 簡略化したい場合によく用いられます. シナプス入力 $I_{\rm syn}(t)$ はシナプス効率 (synaptic efficacy)[*4]を $J_{\rm syn}$ とし (ただし単位は pA), シナプスの動態 (synaptic kinetics) を $s_{\rm syn}(t)$ とすると, 式 (2.2) のようになります. ただし, シナプスの動態とは, 前細胞に注目すれば神経伝達物質の放出量, 後細胞に注目すれば神経伝達物質の結合量やイオンチャネルの開口率を表します.

$$I_{\rm syn}(t) = \underbrace{J_{\rm syn} s_{\rm syn}(t)}_{\text{電流の変化}} \tag{2.2}$$

ただし, $s_{\rm syn}(t)$ は, 例えば次節で紹介する α 関数を用いる場合,

$$s_{\rm syn}(t) = \frac{t}{\tau_s} \exp\left(1 - \frac{t}{\tau_s}\right) \tag{2.3}$$

のようになります.

2.1.2 Conductance-based シナプス

Conductance-based シナプスはイオンチャネルの**コンダクタンスが変化**するというモデルです. 例えば, Hodgkin-Huxley モデルは Conductance-based モデルの 1 つです. このモデルの方が生理学的に妥当です[*5]. シナプス入力は $I_{\rm syn}(t)$ は次のようになります.

$$I_{\rm syn}(t) = \underbrace{g_{\rm syn} s_{\rm syn}(t)}_{\text{コンダクタンスの変化}} \cdot (V_{\rm syn} - V_m(t)) \tag{2.4}$$

ただし, $g_{\rm syn}$ はシナプスの最大コンダクタンス[*6](単位は nS), $V_{\rm syn}$ はシナプスの平衡電位 (単位は mV) を表します. これらも $J_{\rm syn}$ と同じく, シナプスにおける受容体の種類によって決まる定数です.

[*3] シナプス (synapse) 入力であることを明らかにするために syn と添え字をつけています.

[*4] シナプス強度 (Synaptic strength) とは違い, 受容体の種類 (GABA 受容体や AMPA 受容体, およびそのサブタイプなど) によって決まります.

[*5] 例えば抑制性シナプスは膜電位が平衡電位と比べて脱分極側にあるか, 過分極側にあるかで抑制的に働くか興奮的に働くかが逆転します. これは Current-based シナプスでは再現できません.

[*6] ただし, $s_{\rm syn}$ の最大値を 1 に正規化する場合です. 正規化は必須ではないので, 単なる係数と思うのがよいでしょう.

注意しなければならないこととして, $s_{\mathrm{syn}}(t) \leq 0$ としたとき, Current-based モデルにおける J_{syn} は正の値 (興奮性) と負の値 (抑制性) を取りますが, g_{syn} は正の値のみであるということがあります[*7]. Conductance-based モデルで興奮性と抑制性を決定しているのは, 平衡電位 V_{syn} です. 興奮性シナプスの平衡電位は高く, 抑制性シナプスの平衡電位は低いため, 膜電位を引いた符号はそれぞれ正と負になります.

Conductance-based シナプスを実装するためにはニューロンのモデルの方を変更する必要があります. 例として Conductance-based LIF モデルの class を用いた実装を示します[*8].

```
class ConductanceBasedLIF:
    def __init__(self, N, dt=1e-4, tref=5e-3, tc_m=1e-2,
                 vrest=-60, vreset=-60, vthr=-50, vpeak=20,
                 e_exc=0, e_inh=-100):
        self.N = N
        self.dt = dt
        self.tref = tref
        self.tc_m = tc_m
        self.vrest = vrest
        self.vreset = vreset
        self.vthr = vthr
        self.vpeak = vpeak

        self.e_exc = e_exc # 興奮性シナプスの平衡電位
        self.e_inh = e_inh # 抑制性シナプスの平衡電位

        self.v = self.vreset*np.ones(N)
        self.v_ = None
        self.tlast = 0
        self.tcount = 0

    def initialize_states(self, random_state=False):
```

*7 これはコンダクタンスが抵抗の逆数であり, 基本的に抵抗は正の値しか取らないことからも分かります. なお電子回路においては負性抵抗という, 素子の抵抗値が見かけ上, 負の値を取る場合もあります.

*8 コードは./TrainingSNN/Models/Neurons.py に含まれます.

```
        if random_state:
            self.v = self.vreset + \
                     np.random.rand(self.N)*(self.vthr-self.vreset)
        else:
            self.v = self.vreset*np.ones(self.N)
        self.tlast = 0
        self.tcount = 0

    def __call__(self, g_exc, g_inh):
        I_synExc = g_exc*(self.e_exc - self.v) # 興奮性入力
        I_synInh = g_inh*(self.e_inh - self.v) # 抑制性入力
        dv = (self.vrest - self.v + I_synExc + I_synInh) / self.tc_m
        v = self.v+((self.dt*self.tcount)>(self.tlast+self.tref))*dv*self.dt

        s = 1*(v>=self.vthr) # 発火時は 1, その他は 0 の出力

        self.tlast = self.tlast*(1-s) + self.dt*self.tcount*s # 発火時刻の更新
        v = v*(1-s) + self.vpeak*s # 閾値を超えると膜電位を vpeak にする
        self.v_ = v # 発火時の電位も含めて記録するための変数
        self.v = v*(1-s) + self.vreset*s  # 発火時に膜電位をリセット
        self.tcount += 1

        return s
```

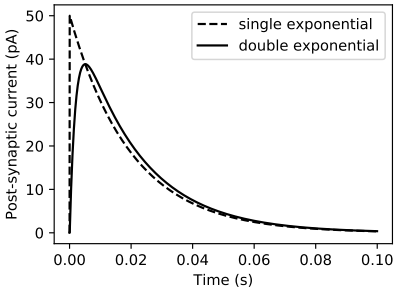
クラスの引数としては, Current-based LIF と比べて興奮性シナプスの平衡電位 e_exc と抑制性シナプスの平衡電位 e_inh の 2 つが増えています [*9]. このクラスのインスタンスは, 興奮性シナプスのコンダクタンスの重みづけ和 g_exc と, 抑制性シナプスのコンダクタンスの重みづけ和 g_inh の 2 入力を受け取ります (**いずれも正の値**). それぞれの入力は平衡電位と膜電位の差分に乗じられ, シナプス電流 I_synExc, I_synInh として膜電位の時間変化に加算されます.

[*9] ここでは大雑把にシナプスを興奮性と抑制性の 2 種類で分けていますが, 神経伝達物質の種類でより細かく分けることもできます.

2.2 指数関数型シナプスモデル (Exponential synapse model)

シナプスのモデルは複数ありますが, SNN で良く用いられるのが**指数関数型シナプスモデル (Exponential synapse model)** です. このモデルは生理学的な過程は無視していますが, シナプス後電流の挙動をよく再現します. 指数関数型シナプスモデルには 2 つの種類があり, **単一指数関数型モデル (Single exponential model)** と**二重指数関数型モデル (Double exponential model)** があります.

数式の説明の前にモデルの挙動を示します. 図 2.1 は 2 種類のモデルにおいて $t = 0$ でスパイクが生じてからのシナプス後電流の変化を示しています. ただし, 実際のシナプス後電流はこれに**シナプス強度** (Synaptic strength)[*10]を乗じて総和を取ったものとなります. シナプス強度については次々節で説明をします.



▲ 図 2.1　2 種類の指数関数型シナプスの動態. 点線は単一指数関数型シナプスで, 実線は二重指数関数型シナプスです. なおコードは数式の説明後に掲載します.

2.2.1　単一指数関数型モデル (Single exponential model)

シナプス前ニューロンにおいてスパイクが生じてからのシナプス後電流の変化はおおよそ指数関数的に減少する, というのが単一指数関数型モデル[*11]です. 式は次のようになります.

$$f(t) = \frac{1}{\tau_s} \exp\left(-\frac{t}{\tau_s}\right) \tag{2.5}$$

[*10] シナプス強度というのは便宜上の呼称で, 実際には神経伝達物質の種類や, その受容体の数など複数の要因によって決定されています. また, このシナプス強度はシナプス重みということもあります. これはどちらかと言えば機械学習の表現に引っ張られたものです. そのため, この本では重みという語も使います.

[*11] 薬学動態の静注 1 コンパートメントモデルと同じ式です.

この関数を時間的なフィルターとして, 過去の全てのスパイクについての総和を取ります[*12].

$$r(t) = \sum_{t_k<t} f\left(t - t_k\right) \tag{2.6}$$

ここで $r(t)$ は前節におけるシナプス動態 (s_{syn}) で, t_k はあるニューロンの k 番目のスパイクの発生時刻です. $t_k < t$ の意味は現在の時刻 t までに発生したスパイクについての和を取るという意味です.

別の表記法としてスパイク列に対する畳み込みを行うというものもあります. 畳み込み演算子を $*$ とし, シナプス前細胞のスパイク列を $S(t) = \sum_{t_k<t} \delta\left(t - t_k\right)$ とします (ただし, δ は Dirac の delta 関数です). このとき, $r(t) = f * S(t)$ と表すことができます. これは簡略な表記ができますが, 実際の計算は他と同じ手法を用います.

上の手法ではニューロンの発火時刻を記憶し, 時間毎に全てのスパイクについての和を取る必要があります. そこで, 実装する場合は次の等価な微分方程式を用います.

$$\frac{dr}{dt} = -\frac{r}{\tau_s} + \frac{1}{\tau_s}\sum_{t_k<t} \delta\left(t - t_k\right) \tag{2.7}$$

ここで τ_s はシナプスの時定数 (synaptic time constant) です. また, $\delta(\cdot)$ は Dirac の delta 関数です (ただし $\delta(0) = 1$ です). これを Euler 法で差分化すると

$$r(t + \Delta t) = \left(1 - \frac{\Delta t}{\tau_s}\right) r(t) + \frac{1}{\tau_s}\delta_{t,t_k} \tag{2.8}$$

となります. ここで δ_{t,t_k} は Kronecker の delta 関数で, $t = t_k$ のときに 1, それ以外は 0 となります. また減衰度として $(1 - \Delta t/\tau_d)$ の代わりに $\exp\left(-\Delta t/\tau_d\right)$ を用いる場合もあります.

2.2.2 二重指数関数型モデル (Double exponential model)

2 重の指数関数によりシナプス後電流の立ち上がりも考慮するのが, 二重指数関数型モデル (Double exponential model) です[*13]. $t = 0$ にシナプス前細胞が発火したときのシナプス後電流の時間変化の関数は次のようになります.

$$f(t) = A\left[\exp\left(-\frac{t}{\tau_d}\right) - \exp\left(-\frac{t}{\tau_r}\right)\right] \tag{2.9}$$

[*12] スパイクが生じてから, ある程度の時間が経過した後はそのスパイクの影響はないと見なせるので, 一定の時間までの総和を取るのがよいです.

[*13] 薬学動態の内服 1 コンパートメントモデルと同じ式です.

ただし, τ_r は立ち上がり時定数 (synaptic rise time constant), τ_d は減衰時定数 (synaptic decay time constant) です. τ_d は τ_s と同じく神経伝達物質の減少速度を決定しています. A は規格化定数で次のように表されます.

$$A = \frac{\tau_d}{\tau_d - \tau_r} \cdot \left(\frac{\tau_r}{\tau_d}\right)^{\frac{\tau_r}{\tau_r - \tau_d}} \tag{2.10}$$

規格化定数 A を乗じることで最大値が 1 となります. ただし, シミュレーションをする上で規格化をする場合は少ないです.

上記の式において, $\tau = \tau_r = \tau_d$ の場合は α **関数** (alpha function, alpha synapse) と呼びます (Rall, 1967). 式としては次のようになります.

$$\alpha(t) = \frac{t}{\tau} \exp\left(1 - \frac{t}{\tau}\right) \tag{2.11}$$

この式は二重指数関数型シナプスの式に単に代入するだけでは導出できません. これらの式の対応については後述します.

ここで, 二重指数関数型シナプスの式に対応する, 補助変数 h を用いた微分方程式を導入します.

$$\frac{dr}{dt} = -\frac{r}{\tau_d} + h \tag{2.12}$$

$$\frac{dh}{dt} = -\frac{h}{\tau_r} + \frac{1}{\tau_r \tau_d} \sum_{t_k < t} \delta\left(t - t_k\right) \tag{2.13}$$

単一指数関数型シナプスの場合と同様に Euler 法で差分化すると

$$r(t + \Delta t) = \left(1 - \frac{\Delta t}{\tau_d}\right) r(t) + h(t) \cdot \Delta t \tag{2.14}$$

$$h(t + \Delta t) = \left(1 - \frac{\Delta t}{\tau_r}\right) h(t) + \frac{1}{\tau_r \tau_d} \delta_{t, t_{jk}} \tag{2.15}$$

となります.

念のため, 微分方程式と元の式が一致することを確認しておきましょう. $t = 0$ のときにシナプス前細胞が発火したとし, それ以降の発火はないとします. このとき, $h(0) = 1/\tau_r \tau_d, r(0) = 0$ です. h についての微分方程式の解は

$$h(t) = h(0) \cdot \exp\left(-\frac{t}{\tau_r}\right) \tag{2.16}$$

となるので, これを r についての式に代入して

$$\frac{dr}{dt} = -\frac{r}{\tau_d} + h(0) \cdot \exp\left(-\frac{t}{\tau_r}\right) \tag{2.17}$$

これを解くには両辺に積分因子 $\exp(t/\tau_d)$ をかけてから積分をするか Laplace 変換をするかです. 今回は Laplace 変換を用いてみます. 右辺一項目を移行した後に両辺を Laplace 変換すると以下のようになります.

$$\mathcal{L}\left[\frac{dr}{dt} + r/\tau_d\right] = \mathcal{L}\left[h(0) \cdot \exp\left(-t/\tau_r\right)\right] \tag{2.18}$$

$$sF(s) - r(0) + \frac{1}{\tau_d}F(s) = \frac{h(0)}{s + 1/\tau_r} \tag{2.19}$$

$$F(s) = \frac{h(0)}{(s + 1/\tau_r)(s + 1/\tau_d)} \tag{2.20}$$

ただし $r(t)$ の Laplace 変換を $F(s)$ としました. ここで逆 Laplace 変換を行うと次のようになります.

$$r(t) = \mathcal{L}^{-1}(F(s)) \tag{2.21}$$

$$= \mathcal{L}^{-1}\left[\frac{h(0)}{(s + 1/\tau_r)(s + 1/\tau_d)}\right] \tag{2.22}$$

$$= \mathcal{L}^{-1}\left[\frac{h(0)}{1/\tau_r - 1/\tau_d}\left(\frac{1}{s + 1/\tau_d} - \frac{1}{s + 1/\tau_r}\right)\right] \tag{2.23}$$

$$= \frac{1}{\tau_d - \tau_r}\left[\exp(-t/\tau_d) - \exp(-t/\tau_r)\right] \tag{2.24}$$

この式の最大値 $r_{\max}$ を求めておきましょう. $r(t)$ を微分して 0 と置いた式の解 $t_{\max}$ を代入すれば求められます. 計算すると,

$$t_{\max} = \frac{\ln(\tau_d/\tau_r)}{1/\tau_r - 1/\tau_d}, \quad r_{\max} = \frac{1}{\tau_d} \cdot \left(\frac{\tau_r}{\tau_d}\right)^{\frac{\tau_r}{\tau_d - \tau_r}} \tag{2.25}$$

となります.

なお, α 関数の導出は逆 Laplace 変換をする前に $\tau = \tau_d = \tau_r$ とすればよく,

$$F_\alpha(s) = \frac{h(0)}{(s + 1/\tau)^2} \tag{2.26}$$

$$\alpha(t) = \frac{t}{\tau^2}\exp\left(-\frac{t}{\tau}\right) \tag{2.27}$$

となります. 若干の係数の違いはありますが, 同じ形の関数が導出されました.

2.2.3 指数関数型シナプスの単純な実装

それでは指数関数型シナプスの実装をしてみましょう. 結果は先に示した, 図 2.1 のようになります[*14].

[*14] コードは./SingleFileSimulations/Synapses/exponential_synapse.py です.

```
dt = 5e-5 # タイムステップ (sec)
td = 2e-2 # synaptic decay time (sec)
tr = 2e-3 # synaptic rise time (sec)
T = 0.1 # シミュレーション時間 (sec)
nt = round(T/dt) #  シミュレーションの総ステップ

# 単一指数関数型シナプス
r = 0 # 初期値
single_r = [] # 記録用配列
for t in range(nt):
    spike = 1 if t == 0 else 0
    single_r.append(r)
    r = r*(1-dt/td) + spike/td
    #r = r*np.exp(-dt/td) + spike/td

# 二重指数関数型シナプス
r = 0; hr = 0 # 初期値
double_r = [] # 記録用配列
for t in range(nt):
    spike = 1 if t == 0 else 0
    double_r.append(r)
    r = r*(1-dt/tr) + hr*dt
    hr = hr*(1-dt/td) + spike/(tr*td)
    #r = r*np.exp(-dt/tr) + hr*dt
    #hr = hr*np.exp(-dt/td) + spike/(tr*td)

time = np.arange(nt)*dt
plt.figure(figsize=(4, 3))
plt.plot(time, np.array(single_r), label="single exponential")
plt.plot(time, np.array(double_r), label="double exponential")
plt.xlabel('Time (s)'); plt.ylabel('Post-synaptic current (pA)')
plt.legend()
plt.show()
```

2.2.4 指数関数型シナプスの class の実装

次章以降で用いるので, 1 章におけるニューロンのモデルと同様に class を用いて指数関数型シナプスを定義しておきます[*15]. 各変数名は前小節での定義と同じです. 2 つのモデルをまとめて Synapses.py に記述し, Models ディレクトリに保存します.

```
class SingleExponentialSynapse:
    def __init__(self, N, dt=1e-4, td=5e-3):
        self.N = N
        self.dt = dt
        self.td = td
        self.r = np.zeros(N)

    def initialize_states(self):
        self.r = np.zeros(self.N)

    def __call__(self, spike):
        r = self.r*(1-self.dt/self.td) + spike/self.td
        self.r = r
        return r

class DoubleExponentialSynapse:
    def __init__(self, N, dt=1e-4, td=1e-2, tr=5e-3):
        self.N = N
        self.dt = dt
        self.td = td
        self.tr = tr
        self.r = np.zeros(N)
        self.hr = np.zeros(N)

    def initialize_states(self):
        self.r = np.zeros(self.N)
        self.hr = np.zeros(self.N)
```

[*15] コードは./TrainingSNN/Models/Synapses.py です.

```
    def __call__(self, spike):
        r = self.r*(1-self.dt/self.tr) + self.hr*self.dt
        hr = self.hr*(1-self.dt/self.td) + spike/(self.tr*self.td)
        self.r = r
        self.hr = hr
        return r
```

2.3 動力学モデル (Kinetic model)

この章以降では扱いませんが, 指数関数型シナプス以外のモデルとして, **動力学モデル** (Kinetic model) があります (Destexhe et al., 1994). モデルの振る舞いはほぼ同一ですが, 式の構成が少し異なります. 動力学モデルは HH モデルのゲート変数の式と類似した式で表されます. このモデルではチャネルが開いた状態 (Open) と閉じた状態 (Close), および神経伝達物質 (neurotransmitter) の放出状態 (T) の 2 つの要素に関する状態があります. また, 閉 $\rightarrow$ 開の反応速度を α, 開 $\rightarrow$ 閉の反応速度を β とします. ここで, これらを表す状態遷移の式は次のようになります.

$$\mathrm{Close} + \mathrm{T} \underset{\beta}{\overset{\alpha}{\rightleftharpoons}} \mathrm{Open} \tag{2.28}$$

ここで, シナプス動態を r とすると

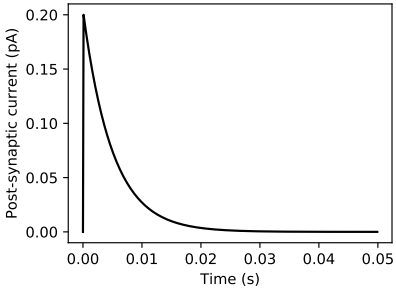
$$\frac{dr}{dt} = \alpha T(1-r) - \beta r \tag{2.29}$$

となります. ただし, T はシナプス前細胞が発火したときにインパルス的に 1 だけ増加するとします. $\alpha = 2000$, $\beta = 200$ とすると[*16], シナプス動態は図 2.2 のようになります. これを表示するコードは以下のようになります[*17].

```
dt = 1e-4; T = 0.05; nt = round(T/dt)
alpha = 1/5e-4; beta = 1/5e-3

r = 0; single_r = [] #記録用配列
for t in range(nt):
```

[*16] α, β は速度なので, 時定数の逆数であることに注意してください.
[*17] コードは./SingleFileSimulations/Synapses/kinetic_synapse.py です.

```
    spike = 1 if t == 0 else 0
    r += (alpha*spike*(1-r) - beta*r)*dt
    single_r.append(r)

time = np.arange(nt)*dt
plt.figure(figsize=(4, 3))
plt.plot(time, np.array(single_r), color="k")
plt.xlabel('Time (s)'); plt.ylabel('Post-synaptic current (pA)')
plt.show()
```



▲ 図 2.2 動力学モデル (Kinetic model) のシナプス動態の時間変化.

2.4 シナプス入力の重みづけ

ここまでは, シナプス前細胞と後細胞がそれぞれ 1 つずつである場合について考えていましたが, 実際には多数の細胞がネットワークを作っています. また, それぞれの入力は均等ではなく, 異なるシナプス強度 (Synaptic strength) を持ちます. この場合のシナプス入力の計算について述べておきます. ここで, シナプス前細胞が N_{pre} 個, シナプス後細胞が N_{post} 個あるとします. このとき**シナプス前過程に注目したシナプス動態**を $\boldsymbol{s}_{\mathrm{syn}} \in \mathbb{R}^{N_{\mathrm{pre}}}$, シナプス後細胞の入力電流を $\boldsymbol{I}_{\mathrm{syn}} \in \mathbb{R}^{N_{\mathrm{post}}}$, シナプス結合強度の行列[*18]を $W \in \mathbb{R}^{N_{\mathrm{post}} \times N_{\mathrm{pre}}}$ とすると, Current-based の場合は

$$\boldsymbol{I}_{\mathrm{syn}}(t) = W \boldsymbol{s}_{\mathrm{syn}} \tag{2.30}$$

[*18] この結合様式は全結合と呼ばれるもので, 第 3 章で説明します.

とします． ただし, シナプス強度にシナプス効率が含まれるとしました． また, Conductance-based の場合はシナプス後細胞の膜電位を $\boldsymbol{V}_m \in \mathbb{R}^{N_{\text{post}}}$ として,

$$\boldsymbol{I}_{\text{syn}}(t) = (V_{\text{syn}} - \boldsymbol{V}_m(t)) \odot W \boldsymbol{s}_{\text{syn}} \tag{2.31}$$

となります. ただし, $\odot$ は Hadamard 積です.

また, これらの式は順序を入れ替えることも可能です. シナプス前細胞でスパイクが生じたことを表すベクトルを $\boldsymbol{\delta}_{t,t_{\text{spike}}} \in \mathbb{R}^{N_{\text{pre}}}$ とします. ただし, t_{spike} は各ニューロンにおいてスパイクが生じた時刻です. $\boldsymbol{s}_{\text{syn}}$ は $\boldsymbol{\delta}_{t,t_{\text{spike}}}$ の関数であり, $\boldsymbol{s}_{\text{syn}}(\boldsymbol{\delta}_{t,t_{\text{spike}}})$ と表せます. ここで, このとき**シナプス後過程に注目したシナプス動態を** $\boldsymbol{s}'_{\text{syn}} \in \mathbb{R}^{N_{\text{post}}}$ とすると, Current-based の場合は

$$\boldsymbol{I}_{\text{syn}}(t) = \boldsymbol{s}'_{\text{syn}}(W\boldsymbol{\delta}_{t,t_{\text{spike}}}) \tag{2.32}$$

Conductance-based の場合は

$$\boldsymbol{I}_{\text{syn}}(t) = (V_{\text{syn}} - \boldsymbol{V}_m(t)) \odot \boldsymbol{s}'_{\text{syn}}(W\boldsymbol{\delta}_{t,t_{\text{spike}}}) \tag{2.33}$$

と表すことができます.

シナプス動態を前過程か後過程のどちらに注目したものとするかは, 実装によって様々です. シナプス入力の計算における中間の値を学習に用いるということもあるため, 単なる計算量の観点だけではどちらを選ぶかは決めることができません (計算量だけならシナプス変数に先に重み行列をかけた方がよい場合が多いです). 実装の中で異なってくるのは計算順序と保持するベクトルの要素数です. 同じ実装の中で 2 つとも用いる場合もあるので注意してください.

コラム：神経回路の汎用シミュレータ

SNN の理解のためには自分でコードを書いた方がよいですが, 効率的にシミュレーションをするには汎用シミュレータを用いるのが良いでしょう.

- NEURON (`http://www.neuron.yale.edu/neuron/`)
- GENESIS (GEneral NEural SImulation System, `http://genesis-sim.org`) 今回触れなかった**マルチコンパートメントモデル**などのシミュレーションをするには, こうしたシミュレータを用いた方が楽でしょう.
- NEST (The NEural Simulation Tool; `http://www.nest-simulator.org`) PyNEST という Python のモジュールも提供されています.
- BRIAN (`http://briansimulator.org/`) Python でのプログラミングに特化しており, チュートリアルも分かりやすいです. 現在は Brian2 が公開されています.
- PyNN (`https://neuralensemble.org/PyNN/`)
- SpykeTorch (`https://github.com/miladmozafari/SpykeTorch`)
- BindsNET (`https://github.com/Hananel-Hazan/bindsnet`)

SpykeTorch と BindsNET は Pytorch を用いて SNN を実装しています.

第 3 章

ネットワークの構築

第 1 章ではニューロンのモデルについて, 第 2 章ではシナプスのモデルについて学んできました. 第 3 章ではそれらのモデルを組み合わせたネットワークを構築してみます. また, 最後の節では SNN を学習させる意義とその方針について説明します.

3.1 ニューロン間の接続

ネットワークを構成するにはあるニューロンがどのニューロンに投射しているか, どのように活動が伝搬するかを記述する必要があります. この節ではニューロン同士の間の接続関係の記述の仕方について説明します.

3.1.1 全結合 (Full connection)

i 層目のニューロンが $i+1$ 層目のニューロンに全て繋がっていることを**全結合 (fully connected)** と言います. ただし, 全てが完全に繋がっているということではなく, 結合重みが 0 の場合は繋がっていないことを表します. なお, この結合様式は既に第 2 章で出てきています. 全結合は ANN では入力に重み行列を乗算し, バイアスを加算するようなアフィン変換で表されますが, SNN では入力に重み行列を乗算するだけの線形変換を用いることが主です.

単に重み行列を用意するだけでも (この本の内容に限るなら) 問題はありませんが, 重みを学習させる場合には `class` を用意しておくと取り扱いがしやすくなります. コードは次のようになります[*1]. このコードを `Connections.py` として `Models` ディレクトリ内に保存しておきましょう.

[*1] コードは./TrainingSNN/Models/Connections.py に含まれます.

```
class FullConnection:
    def __init__(self, N_in, N_out, initW=None):
        if initW is not None:
            self.W = initW
        else:
            self.W = 0.1*np.random.rand(N_out, N_in)

    def backward(self, x):
        return np.dot(self.W.T, x) #self.W.T @ x

    def __call__(self, x):
        return np.dot(self.W, x) #self.W @ x
```

3.1.2 2 次元の畳み込み (Convolution2D connection)

SNN では ANN の 1 つの結合形式である**畳み込み層** (convolutional layer) を含むことがあります．全結合が通常の ANN と同様であったように畳み込み層も全く同じ操作です．そのため, 今回実装はしないのですが, 行列計算ライブラリとして NumPy ではなく, Tensorflow や Pytorch, Chainer 等を使う場合には畳み込み層の関数が実装されているのでそれを使うとよいでしょう．

念のため, 2D 畳み込み層の出力テンソル ($H \times W \times C$ のテンソル, H, W はそれぞれ画像の高さと幅, C はチャネル数) の解釈について説明しておきます．まず, 1 つのチャネルは同種 (同系統の受容野を持つ) の $H \times W$ 個のニューロンの活動です．本来は「同種」ですが, 空間的な不変性により「同一」と見なし, 重み共有 (weight sharing, weight tying) をしてスライディングウィンドウ (sliding window) の操作をすることで, ニューロンを視野全体に複製 (要は 1 つのニューロンをコピペ) しています．実際の視覚野では近傍のニューロンの活動を受けることによる畳み込みはしていますが, 重み共有[*2]とスライディングウィンドウはしていない, ということです．

[*2] ただし, 類似の遺伝子発現による初期値共有はしているかもしれないですが.

3.1.3 遅延結合 (Delay connection)

実際のニューロンにおいて, シナプス前細胞での発火が瞬間的にシナプス後細胞に伝わるということはありません. これは**軸索遅延 (axonal delays)** や**シナプス遅延 (synaptic delay)** があるためです. ここでは発火情報の伝搬における遅延の実装について説明します. ただし, 全てのニューロンの遅延が等しいとした場合のみです[*3].

実装は通常の離散遅延微分方程式をシミュレーションする際と同様にします. まず, 行数はニューロンの数, 列数は遅延時間のシミュレーションステップ数と同じ長さとした行列を用意します. ステップごとに最後の行にあたるベクトルを出力し, 配列をずらした後[*4], 初めの行を新しい入力で更新します.

```
class DelayConnection:
    def __init__(self, N, delay, dt=1e-4):
        nt_delay = round(delay/dt) # 遅延のステップ数
        self.state = np.zeros((N, nt_delay))

    def __call__(self, x):
        out = self.state[:, -1] # 出力
        self.state[:, 1:] = self.state[:, :-1] # 配列をずらす
        self.state[:, 0] = x # 入力
        return out
```

このコードも `./Models/Connections.py` に記して保存しておきましょう.

次に, 遅延が正しく表現されているか確認してみましょう[*5]. まず, `Models` ディレクトリをパッケージとして認識させるために `__init__.py` という名称のファイルを作成し (何も書かれてなくてよいです), `Models` ディレクトリ内に保存します. 次に `Models` ディレクトリの親ディレクトリ内にこれから書くファイルを置きます. こうすることで `Models` ディレクトリ内のファイルから作成した `class` を import することができます. これは以降のコードでほぼ共通です.

さて, コードは次のようになります. 初めにモデルの import と定数の定義, モデルのインスタンスの作成, 記録用配列の定義を行っています.

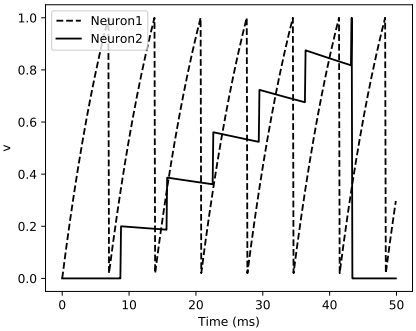
[*3] 遅延時間をバラバラにすると行列での取り扱いが難しくなり, for loop を用いる他にないと思うので省略します. 実装したい場合は C++ や Julia など for loop が速い言語を用いてください.

[*4] `np.roll` を用いるよりもこちらの方が速いです.

[*5] コードは `./TrainingSNN/example_using_delay_connection.py` です. また, この部分は Brian2 の tutorial を参考にしました.

```
from Models.Neurons import CurrentBasedLIF
from Models.Connections import DelayConnection

dt = 1e-4; T = 5e-2; nt = round(T/dt)

#モデルの定義
neuron1 = CurrentBasedLIF(N=1, dt=dt, tc_m=1e-2, tref=0,
                          vrest=0, vreset=0, vthr=1, vpeak=1)
neuron2 = CurrentBasedLIF(N=1, dt=dt, tc_m=1e-1, tref=0,
                          vrest=0, vreset=0, vthr=1, vpeak=1)
delay_connect = DelayConnection(N=1, delay=2e-3, dt=dt)

I = 2 # 入力電流
v_arr1 = np.zeros(nt); v_arr2 = np.zeros(nt) #記録用配列

for t in tqdm(range(nt)):
    # 更新
    s1 = neuron1(I)
    d1 = delay_connect(s1)
    s2 = neuron2(0.02/dt*d1)

    # 保存
    v_arr1[t] = neuron1.v_
    v_arr2[t] = neuron2.v_

time = np.arange(nt)*dt*1e3
plt.figure(figsize=(5, 4))
plt.plot(time, v_arr1, label="Neuron1", linestyle="dashed")
plt.plot(time, v_arr2, label="Neuron2")
plt.xlabel("Time (ms)"); plt.ylabel("v")
plt.legend(loc="upper left")
plt.show()
```

結果は図 3.1 のようになります.



▲ 図 3.1　ニューロン 1 からニューロン 2 へと 2 ms の遅延で発火が伝わる場合.

3.2 ランダムネットワーク

この節ではこれまでに実装した SNN の要素を組み合わせ, 重みがランダムなネットワーク (random network) を構成してみましょう. 作成するネットワークは 2 層から成り, 1 層目には 10 個の Poisson スパイクニューロン, 2 層目には 1 個の LIF ニューロンがあるとします. 1 層目のニューロンから 2 層目のニューロンへのシナプス結合には, 二重指数関数型シナプスを用います. 目標は 2 層目のニューロンの膜電位と入力電流, 1 層目のニューロンのラスタープロット (raster plot) を表示することです.

それではネットワークを構築してみましょう[*6]. まず, ニューロンとシナプスのクラスを `import` し, 各種定数, 入力のポアソンスパイク `x`, 結合重み `W`, ニューロンとシナプスのモデルの各インスタンス (`neurons`, `synapses`), 記録用の配列を定義します. 注意点として, 先ほどと同様に実行ファイルは `Models` ディレクトリの親ディレクトリ内に置くようにしましょう.

```
from Models.Neurons import CurrentBasedLIF
from Models.Synapses import DoubleExponentialSynapse


np.random.seed(seed=0)
```

[*6] コードは `./TrainingSNN/LIF_random_network.py` です.

```
dt = 1e-4; T = 1; nt = round(T/dt) # シミュレーション時間
num_in = 10; num_out = 1 # 入力 / 出力ニューロンの数

# 入力のポアソンスパイク
fr_in = 30 # 入力のポアソンスパイクの発火率 (Hz)
x = np.where(np.random.rand(nt, num_in) < fr_in * dt, 1, 0)
W = 0.2*np.random.randn(num_out, num_in) # ランダムな結合重み

# モデル
neurons = CurrentBasedLIF(N=num_out, dt=dt, tref=5e-3,
                          tc_m=1e-2, vrest=-65, vreset=-60,
                          vthr=-40, vpeak=30)
synapses = DoubleExponentialSynapse(N=num_out, dt=dt, td=1e-2, tr=1e-2)

# 記録用配列
current = np.zeros((num_out, nt))
voltage = np.zeros((num_out, nt))
```

次に, for ループ内でネットワークの流れを書き, シミュレーションを実行してみましょう.

```
# シミュレーション
neurons.initialize_states() # 状態の初期化
for t in tqdm(range(nt)):
    # 更新
    I = synapses(np.dot(W, x[t]))
    s = neurons(I)

    # 記録
    current[:, t] = I
    voltage[:, t] = neurons.v_
```

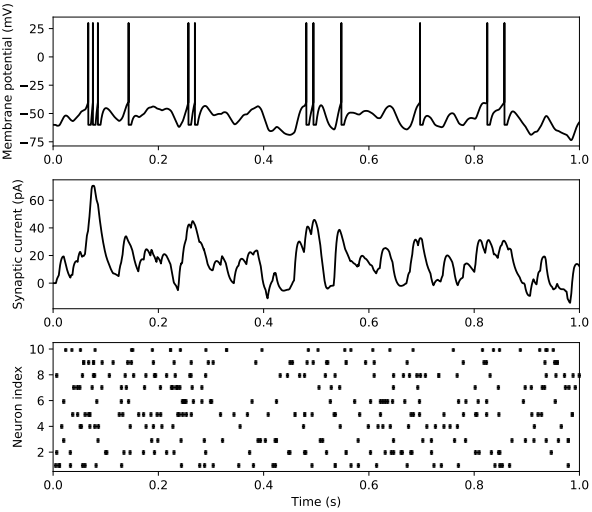
ここでは, 全結合は FullConnection クラスを用いず, 簡単のために np.dot(W, x[t]) で表し, synapses の出力はシナプス後電流とします. 第二章で述べたように synapses

の出力が何を意味するのか, すなわちシナプス前細胞の神経伝達物質の放出量なのか, シナプス後細胞のチャネルの開口頻度なのかは場合によって変わるので, 注意するようにしましょう. 今回の場合はシナプス後細胞に注目したモデルとなっています.

最後にシミュレーションの結果を描画してみましょう. 描画するのは前述したように 2 層目のニューロンの膜電位と入力電流, 1 層目のニューロンのラスタープロットです.

```
# 結果表示
t = np.arange(nt)*dt
plt.figure(figsize=(7, 6))
plt.subplot(3,1,1)
plt.plot(t, voltage[0], color="k")
plt.xlim(0, T)
plt.ylabel('Membrane potential (mV)')

plt.subplot(3,1,2)
plt.plot(t, current[0], color="k")
plt.xlim(0, T)
plt.ylabel('Synaptic current (pA)')

plt.subplot(3,1,3)
for i in range(num_in):
    plt.plot(t, x[:, i]*(i+1), 'ko', markersize=2)
plt.xlabel('Time (s)')
plt.ylabel('Neuron index')
plt.xlim(0, T)
plt.ylim(0.5, num_in+0.5)
plt.show()
```

これを実行した結果は図 3.2 のようになります.



▲ 図 3.2　ランダムネットワークの活動. (上)2 層目のニューロンの膜電位変化. (中) 2 層目のニューロンへの入力電流. (下) 1 層目のポアソンスパイクニューロンのラスタープロット.

3.3 SNN を訓練する

3.3.1　SNN の意義

DeepMind のアルファ碁が 2017 年に柯潔九段に勝利したとき, 同時に両者の消費電力 (エネルギー) が話題となりました. ヒトの脳の消費エネルギーは平常時で約 20W, 高負荷な思考時には約 21W 程度しか消費しない[*7]のに対し, アルファ碁 (2000CPU, 300GPU) の消費電力は 25 万 W であったそうです. この違いはどこにあるかといえば, アルゴリズムによる部分もありますが, ハードウェアによるところが大きいです (もちろん脳というハードウェアにおけるアルゴリズムも優れていると思いますが). そこで, 神経回路網を模した低消費電力のハードウェアとして **Neuromorphic Hardware** と呼ばれるチップが開発されています. 例を挙げると IBM の TrueNorth (Merolla et al., 2014), Intel の Loihi (Davies et al., 2018), Manchester 大学 (APT group) の SpiNNaker (Furber et al., 2014), Heidelberg 大学の BrainScaleS (Meier, 2015), Stanford 大学の NeuroGrid

[*7] ということで思考時と休止時では 1W 程度しか変わりません. この話は有名ですが, 実は自分は元の論文を知りません.

(Benjamin et al., 2014), INI Zurich の DYNAP (Moradi et al., 2018), BrainChip の Akida などがあります. 通常のコンピュータであれば SNN の方が計算量は大きいので, むしろ消費電力は高くなりますが, これらのチップは SNN に特化しており, 低消費電力で SNN をシミュレーションできます. Neuromorphic Hardware が低消費電力であるのは, いくつか理由があり (森江, 2019), 主に次のことが理由として挙げられます.

- ニューロンの出力 (すなわちスパイク) が 0 か 1 かの 2 値しかないため, オペアンプを使用する必要がなく, 高速かつ電力消費が少ないコンバータを使用することができる点.
- スパイクが到達した場合のみチップが駆動するようになっており (これを event-driven 型と呼びます) [*8], そのため従来のデジタル回路のようにクロック同期をする必要がなく, 非同期で動作できる点.

このようにハードウェアで実装する上では SNN は ANN にない利点があるため, SNN を高効率に学習させるアルゴリズムが研究されています.

単純に考えて ANN は発火率コーディング (rate coding) しか使えないのに対し, SNN は発火率コーディングに加え時間的コーディング (temporal coding) も用いることができるので, 将来的には ANN を超えてもよさそうなものです. しかし, SNN の学習は難しく, ANN を超える性能というところまでは至っていません (この原因は SNN は ANN のように誤差逆伝搬法という高効率なアルゴリズムが使えないということです). そのため, SNN の訓練と一口に言っても, これさえやっておけばよいというのはありません. 次節では SNN を訓練させる方針について紹介します.

3.3.2 SNN を訓練する 5 つの方針

現在のところ, SNN を訓練する方針は大きく分けて 5 つあります (Pfeiffer & Pfeil, 2018).

1. **ANN の 2 値化**
   2 値化された ANN(Binarized Neural Networks) により, SNN のように膜電位はシミュレーションしないものの, 高速かつ省メモリ化に ANN を実行できます. 出力の 2 値化 (0 or 1 または 1 or −1) の場合に限らず, 重みや勾配も量子化 (quantization) する研究もあります. なお, ここでの量子化とは出力などの値を少ない bit 数で表現することを意味します.
2. **ANN を SNN に変換**
   従来の ANN をそのまま学習させた後, 係数倍で補正して SNN に用いるという手

[*8] 通常のコンピュータであれば, 発火していないことを表す 0 の値も保持し続ける必要があります.

法があります.

3. **制約付きの ANN を SNN に変換 (constrain-then-train)**
   SNN の発火特性に一致するようにした制約付きの ANN を学習させ, 補正なしで直接重みを転用するという手法です.
4. **誤差逆伝搬法の近似による教師あり学習**
   ANN の学習に用いる誤差逆伝搬法 (backpropagation) を近似し, SNN に教師あり学習を行います.
5. **局所的な学習規則による教師なし学習**
   STDP (spike-timing dependent plasticity) などの学習規則で教師なし学習を行います.

1 番目の ANN の 2 値化の方法についてはこの本では触れません. (Hubara et al., 2016; Kim & Smaragdis, 2016; Rastegari et al., 2016; Severa et al., 2019) などを参照してください. 2 番目の ANN から SNN への変換は (Sengupta et al., 2019) などを参照してください.

3 番目の制約付き訓練については (Esser et al., 2015) を参照してください. 関連して, Noisy Softplus という ANN の活性化関数の提案があります (Liu et al., 2017). 入力にノイズが加わった LIF ニューロンの F-I curve を模倣するように設計されており, ANN で学習させた重みを SNN でそのまま用いることができます[*9].

残りの方針についてですが, この本では 4 番目の方針を第 4 章で, 5 番目の方針を第 5 章で扱います. さらに特殊なネットワークである Reservoir computing の学習を第 6 章で扱います.

[*9] Noisy Softplus は次のような関数です (k, σ がハイパーパラメータで, これらを大きくすることは, 元の LIF ニューロンにおける入力ノイズが大きくなる場合と対応します).

$$y = f_{ns}(x, \sigma) = k\sigma \log\left[1 + \exp\left(\frac{x}{k\sigma}\right)\right]$$

この微分は

$$\frac{\partial f_{ns}(x, \sigma)}{\partial x} = \frac{1}{1 + \exp\left(-\frac{x}{k\sigma}\right)}$$

となります. この関数は基本的な関数の組み合わせなので, Pytorch や Chainer ではライブラリの関数を用いて実装することができます.

第 4 章

誤差逆伝搬法の近似による教師あり学習

ANN は誤差逆伝搬法 (backpropagation) を用いてパラメータを学習することができますが, SNN は誤差逆伝搬法を直接使用することはできません. しかし, 誤差逆伝搬法の近似をすることで SNN を訓練することができるようになります. SNN を誤差逆伝搬法で訓練することは **SpikeProp 法** (Bohte et al., 2000) や **ReSuMe 法** (Ponulak, Kasiński, 2010) など多数の手法が考案されてきました (他の方針としては Lee et al. 2016; Huh & Sejnowski, 2018; Wu et al., 2018; Shrestha & Orchard, 2018; Tavanaei & Maida, 2019; Thiele et al., 2019; Comsa et al., 2019 など多数). この章の初めでは, 代表して SpikeProp 法の改善手法である **SuperSpike 法** (Zenke & Ganguli, 2018) の実装をしてみます.

4.1 SuperSpike 法

SuperSpike 法 (supervised learning rule for spiking neurons) はオンラインの教師あり学習で SpikeProp 法と同様にスパイク列を教師信号とし, そのスパイク列を出力するようにネットワークを最適化します (Zenke & Ganguli, 2018). SpikeProp 法と異なるのはスパイクの微分ではなく, 膜電位についての関数の微分を用いていることです. このため, 発火が生じなくても学習が進行します.

4.1.1 損失関数の導関数の近似

まず最小化したい損失関数 L から考えましょう. i 番目のニューロンの教師信号となるスパイク列 $\hat{S}_i$ に出力 S_i を近づけます (スパイク列は $S_i(t) = \sum_{t_k<t} \delta\left(t - t_i^k\right)$ と表され

ます)[*1]. SpikeProp 法ではこれらの二乗誤差を損失関数としていますが, SuperSpike 法ではそれぞれのスパイク列を二重指数関数フィルター α で畳み込みした後に二乗誤差を取ります.

$$L(t) = \frac{1}{2}\int_{-\infty}^{t} ds \left[\left(\alpha * \hat{S}_i - \alpha * S_i\right)(s)\right]^2 \tag{4.1}$$

ただし, $*$ は畳み込み演算子です. これは **van Rossum 距離** (van Rossum, 2001)[*2]を表します. 損失関数をこのように設定することで, SpikeProp と異なり, 完全にスパイク列が一致するまで誤差信号は 0 になりません. 損失関数 L を j 番目のシナプス前ニューロンから i 番目のシナプス後ニューロンへのシナプス強度 w_{ij} で微分すると, 次のようになります.

$$\frac{\partial L}{\partial w_{ij}} = -\int_{-\infty}^{t} ds \left[\left(\alpha * \hat{S}_i - \alpha * S_i\right)(s)\right] \left(\alpha * \frac{\partial S_i}{\partial w_{ij}}\right)(s) \tag{4.2}$$

目標はこの $\dfrac{\partial L}{\partial w_{ij}}$ を計算し, 確率的勾配降下法 (stochastic gradient discent; SGD) により $w_{ij} \leftarrow w_{ij} - r\dfrac{\partial L}{\partial w_{ij}}$ と最適化することです (ただし r は学習率). ここでの問題点は $\dfrac{\partial S_i}{\partial w_{ij}}$ の部分です. S_i は δ 関数を含むため, 微分すると発火時は ∞, 非発火時は 0 となり, 学習が進みません. そこで $S_i(t)$ を LIF ニューロンの膜電位[*3]$U_i(t)$ の非線形関数 $\sigma(U_i(t))$ で近似します. 非線形関数としては高速シグモイド関数 (fast sigmoid) $\sigma(x) = x/(1+|x|)$ を使用しています. ここまでの近似計算を纏めると

$$\frac{\partial S_i}{\partial w_{ij}} \approx \frac{\partial \sigma\left(U_i\right)}{\partial w_{ij}} = \sigma'\left(U_i\right)\frac{\partial U_i}{\partial w_{ij}} \tag{4.3}$$

となります. ただし, $\sigma'\left(U_i\right) = \left(1 + |\beta(U_i - \vartheta)|\right)^{-2}$ です. ϑ は LIF ニューロンの発火閾値で -50 mV とされています. β は係数で $(1\text{ mV})^{-1}$ です.

残った $\dfrac{\partial U_i}{\partial w_{ij}}$ の部分ですが, シナプス強度 w_{ij} の変化により j 番目のシナプス前ニューロンの発火 $S_j(t)$ が i 番目のシナプス後細胞の膜電位変化に与える影響が変化するという観点から, $\dfrac{\partial U_i}{\partial w_{ij}} \approx \epsilon * S_j(t)$ と近似します. ただし, ϵ は α と同じ二重指数関数フィルターです. また, これはシナプスでの神経伝達物質の濃度として解釈できるとされています.

[*1] 通常, 予測値に $\hat{}$ を付けることが多いですが, ここでは論文の表記に従って $\hat{S}$ を教師信号としています.

[*2] スパイク列の類似度を計算する手法としては他に Victor-Purpura 距離や, Schreiber *et al.* 類似度など, 数多く考案されています. (Dauwels et al., 2008) や Scholarpedia の "Measures of spike train synchrony"(`http://www.scholarpedia.org/article/Measures_of_spike_train_synchrony`) を参照してください.

[*3] これまで V や v を使っていましたが, 論文にあわせて U を用います.

ここまでの近似を用いると, 時刻 t におけるシナプス強度の変化率 $\dfrac{\partial w_{ij}}{\partial t}$ は

$$\frac{\partial w_{ij}}{\partial t} = -r\frac{\partial L}{\partial w_{ij}} \tag{4.4}$$

$$\approx r\int_{-\infty}^{t} ds \underbrace{\left[\alpha * \left(\hat{S}_i - S_i\right)(s)\right]}_{\text{誤差信号}} \quad \alpha * \left[\underbrace{\sigma'\left(U_i(s)\right)}_{\text{後細胞}}\underbrace{\left(\epsilon * S_j\right)(s)}_{\text{前細胞}}\right] \tag{4.5}$$

$$= r\int_{-\infty}^{t} ds \;\; e_i(s)\cdot\lambda_{ij}(s) \tag{4.6}$$

と表せます. ここで, $e_i(t) = \alpha * \left(\hat{S}_i - S_i\right)$, $\lambda_{ij}(t) = \alpha * \left[\sigma'\left(U_i(s)\right)\left(\epsilon * S_j\right)(s)\right]$ としました. $e_i(t)$ は誤差信号 (error signal) で, シナプス前細胞にフィードバックされます. $\lambda_{ij}(t)$ はシナプス適格度トレース (synaptic eligibility trace) を表します[*4].

4.1.2 離散化した重みの更新と RMaxProp

前項における $\dfrac{\partial w_{ij}}{\partial t}$ は時刻 t までの全ての誤差情報を積分していますが, 実装する上での利便性を考え, 時刻 $[t_k, t_{k+1}]$ の間の積分を用いて重みを更新します[*5].

$$\Delta w_{ij}^{k} = r_{ij}\int_{t_k}^{t_{k+1}} e_i(s)\lambda_{ij}(s)ds \tag{4.7}$$

ただし, r_{ij} は重み w_{ij} ごとの学習率です (これは後で説明します). さらに実装時には $t_b := t_{k+1} - t_k$ $(= 0.5\text{ s})$ とし, 0 で初期化されている配列 $[m_{ij}]$ をシミュレーションステップごとに

$$m_{ij} \leftarrow m_{ij} + g_{ij} \tag{4.8}$$

という式により更新します. ただし, $g_{ij} = e_i(t)\lambda_{ij}(t)$ です. t_b だけ経過すると,

$$w_{ij} \leftarrow w_{ij} + r_{ij}m_{ij}\cdot\Delta t \tag{4.9}$$

として重み w_{ij} を更新し, m_{ij} を 0 にリセットします[*6]. さらに更新時は重みに $-1 < w_{ij} < 1$ という制限をつけています.

[*4] これは遅延報酬問題 (distal reward problem) を解決していると説明されています. また, 生理学的には Ca^{2+} トランジェント (calcium transient) や関連するシグナル伝達カスケード (signaling cascade) として実現可能であるとされています.

[*5] これはミニバッチによる更新に類似しています.

[*6] Δt は元の論文には記載されていないですが, タイムステップの長さが変化しても良いようにするためにつけています.

学習率 r は全ての重みに対して同じものを用いても学習は可能ですが, 安定はしません. そこで, ANN の Optimizer の一種である **RMSprop** と類似した更新を行います.

まず, 新しく配列 $[v_{ij}]$ を用意します. ステップごとに

$$v_{ij} \leftarrow \max(\gamma v_{ij}, g_{ij}^2) \tag{4.10}$$

で更新します. ただし, γ はハイパーパラメータです (明確な値の記載がありませんが, 実験の結果から 0.8 程度の値がよいでしょう). この v_{ij} を用いて重みごとの学習率 r_{ij} を次のように定義します.

$$r_{ij} = \frac{r_0}{\sqrt{v_{ij}} + \varepsilon} \tag{4.11}$$

ただし, r_0 は学習係数, ε はゼロ除算を避けるための小さい値 (典型的には $\varepsilon = 10^{-8}$) です. 記載はありませんが, 学習係数の減衰 (learning rate decay) を行うと学習がよく進みました.

以上の更新法を著者らは **RMaxProp** と名付けています. なお, RMSprop の場合は g_{ij}^2 の移動平均を次式のように行います.

$$v_{ij} \leftarrow \gamma v_{ij} + (1-\gamma) \cdot g_{ij}^2 \tag{4.12}$$

4.1.3 誤差信号の逆伝搬について

出力層において誤差信号は $e_i(t) = \alpha * \left(\hat{S}_i - S_i\right)$ と計算されます. これを低次の層に逆伝搬すること, つまり l 層目の k 番目のニューロンの誤差信号 e_k を $l-1$ 層目の i 番目のニューロンに投射することを考えます. 対称なフィードバックをする場合, $W = [w_{ik}]$ の転置行列 $W^\intercal = [w_{ki}]$ を用いて,

$$e_i = \sum_k w_{ki} e_k \tag{4.13}$$

となります. ここで ANN の誤差逆伝搬のように, $l-1$ 番目の層の出力を引数とする活性化関数の勾配を乗じません.

この対称フィードバックは順伝搬の重みの転置行列を用いるため, 生物学的には妥当ではありません. そこで誤差逆伝搬法の対称な重みを使う問題を解消する手法として **Feedback alignment** (Lillicrap et al., 2016) があります[*7]. Feedback alignment では逆伝搬時に用いる重みをランダムに固定したものとします[*8]. このとき, ランダムな固定

[*7] Feedback alignment の発展については (Nøkland 2016; Akrout et al., 2019; Lansdell et al 2019) などを参照してください.

[*8] なぜ Feedback alignment が上手くいくかは, 2 層における証明 (Lillicrap et al., 2016) や多層における証明 (Nøkland 2016) を参考にしてください.

重みを $B = [b_{ki}]$ とすると, Feedback alignment の場合は

$$e_i = \sum_k b_{ki} e_k \tag{4.14}$$

となります. また, 重みを均一なものとする Uniform feedback による学習も紹介されています. この場合は,

$$e_i = \sum_k e_k \tag{4.15}$$

となります. 後の実装では Feedback alignment による学習も行います.

4.1.4 SuperSpike 法の実装

それでは SuperSpike 法を実装していきましょう. 今回は 3 層のネットワーク (ユニット数は順に 50, 4, 1) の出力ユニットが 100 ms ごとに発火するように訓練します. 訓練後の結果と全体のアルゴリズムは図 4.1 にまとめていますので, 適宜参照すると良いでしょう.

まず, `Models` ディレクトリの親ディレクトリに実行するファイルを作成します. 次に今回使うモデルを import しておきます.

```
from Models.Neurons import CurrentBasedLIF
from Models.Synapses import DoubleExponentialSynapse
from Models.Connections import FullConnection, DelayConnection
```

誤差信号と適格度トレースの実装

誤差信号と適格度トレースの計算を行う class を実装します. とはいえ, 二重指数関数型シナプスのコードに少し変更を加えるだけでよいです.

まず, 誤差信号では出力層のスパイク列 (`output_spike`) と教師信号のスパイク列 (`target_spike`) を引数とし, それらの差分を取ります. さらにこの時, 出力が $[-1, 1]$ となるように規格化を行います.

```
class ErrorSignal:
    def __init__(self, N, dt=1e-4, td=1e-2, tr=5e-3):
        self.dt = dt
        self.td = td
        self.tr = tr
```

```
        self.N = N
        self.r = np.zeros(N)
        self.hr = np.zeros(N)
        self.b = (td/tr)**(td/(tr-td)) # 規格化定数

    def initialize_states(self):
        self.r = np.zeros(self.N)
        self.hr = np.zeros(self.N)

    def __call__(self, output_spike, target_spike):
        r = self.r*(1-self.dt/self.tr) + self.hr/self.td*self.dt
        hr = self.hr*(1-self.dt/self.td)+(target_spike-output_spike)/self.b
        self.r = r
        self.hr = hr
        return r
```

次に, 適格度トレースはシナプス前細胞のシナプスフィルターをかけられたスパイク列と, シナプス後細胞の膜電位を引数とします. シナプス後細胞の膜電位は高速シグモイド関数の微分した式に代入され, Online STDP[*9]の計算のように列ベクトル (post の活動) と行ベクトル (pre の活動) の積を取ります.

```
lass EligibilityTrace:
    def __init__(self, N_in, N_out, dt=1e-4, td=1e-2, tr=5e-3):
        self.dt = dt
        self.td = td
        self.tr = tr
        self.N_in = N_in
        self.N_out = N_out
        self.r = np.zeros((N_out, N_in))
        self.hr = np.zeros((N_out, N_in))

    def initialize_states(self):
```

[*9] 5 章参照

```
        self.r = np.zeros((self.N_out, self.N_in))
        self.hr = np.zeros((self.N_out, self.N_in))

    def surrogate_derivative_fastsigmoid(self, u, beta=1, vthr=-50):
        return 1 / (1 + np.abs(beta*(u-vthr)))**2

    def __call__(self, pre_current, post_voltage):
        # (N_out, 1) x (1, N_in) -> (N_out, N_in)
        pre_ = np.expand_dims(pre_current, axis=0)
        post_ = np.expand_dims(
                self.surrogate_derivative_fastsigmoid(post_voltage),
                axis=1)
        r = self.r*(1-self.dt/self.tr) + self.hr*self.dt
        hr = self.hr*(1-self.dt/self.td)+np.dot(post_,pre_)/(self.tr*self.td)
        self.r = r
        self.hr = hr
        return r
```

定数とモデルの定義

それでは準備が終わったので, 定数とモデルのインスタンスを定義しましょう.

```
dt = 1e-4; T = 0.5; nt = round(T/dt)

t_weight_update = 0.5 #重みの更新時間
nt_b = round(t_weight_update/dt) #重みの更新ステップ

num_iter = 200 # 学習のイテレーション数

N_in = 50 # 入力ユニット数
N_mid = 4 # 中間ユニット数
N_out = 1 # 出力ユニット数

# 入力 (x) と教師信号 (y) の定義
```

```
fr_in = 10 # 入力の Poisson 発火率 (Hz)
x = np.where(np.random.rand(nt, N_in) < fr_in * dt, 1, 0)
y = np.zeros((nt, N_out))
y[int(nt/10)::int(nt/5), :] = 1 # T/5 に 1 回発火

# モデルの定義
neurons_1 = CurrentBasedLIF(N_mid, dt=dt)
neurons_2 = CurrentBasedLIF(N_out, dt=dt)
delay_conn1 = DelayConnection(N_in, delay=8e-4)
delay_conn2 = DelayConnection(N_mid, delay=8e-4)
synapses_1 = DoubleExponentialSynapse(N_in, dt=dt, td=1e-2, tr=5e-3)
synapses_2 = DoubleExponentialSynapse(N_mid, dt=dt, td=1e-2, tr=5e-3)
es = ErrorSignal(N_out)
et1 = EligibilityTrace(N_in, N_mid)
et2 = EligibilityTrace(N_mid, N_out)

connect_1 = FullConnection(N_in, N_mid,
                           initW=0.1*np.random.rand(N_mid, N_in))
connect_2 = FullConnection(N_mid, N_out,
                           initW=0.1*np.random.rand(N_out, N_mid))
#B = np.random.rand(N_mid, N_out) # Feedback alignment

r0 = 1e-3
gamma = 0.7

# 記録用配列
current_arr = np.zeros((N_mid, nt))
voltage_arr = np.zeros((N_out, nt))
error_arr = np.zeros((N_out, nt))
lambda_arr = np.zeros((N_out, N_mid, nt))
dw_arr = np.zeros((N_out, N_mid, nt))
cost_arr = np.zeros(num_iter)
```

ここで配列 B は Feedback alignment の際に用います.

シミュレーションの実装

for ループ内でモデルを構築し, nt_b ステップごとに重みの更新を行います. また, 最後の訓練イテレーション時に, 出力層の膜電位の時間変化などを記録しておきます.

```
for i in tqdm(range(num_iter)):
    if i%15 == 0:
        r0 /= 2 # 重み減衰

    # 状態の初期化
    neurons_1.initialize_states()
    neurons_2.initialize_states()
    synapses_1.initialize_states()
    synapses_2.initialize_states()
    delay_conn1.initialize_states()
    delay_conn2.initialize_states()
    es.initialize_states()
    et1.initialize_states()
    et2.initialize_states()

    m1 = np.zeros((N_mid, N_in))
    m2 = np.zeros((N_out, N_mid))
    v1 = np.zeros((N_mid, N_in))
    v2 = np.zeros((N_out, N_mid))
    cost = 0
    count = 0

    # one iter.
    for t in range(nt):
        # Feed-forward
        c1 = synapses_1(delay_conn1(x[t])) # input current
        h1 = connect_1(c1)
        s1 = neurons_1(h1) # spike of mid neurons

        c2 = synapses_2(delay_conn2(s1))
```

```
        h2 = connect_2(c2)
        s2 = neurons_2(h2)

        # Backward(誤差の伝搬)
        e2 = np.expand_dims(es(s2, y[t]), axis=1) / N_out
        e1 = connect_2.backward(e2) / N_mid
        # e1 = np.dot(B, e2) / N_mid

        # コストの計算
        cost += np.sum(e2**2)

        lambda2 = et2(c2, neurons_2.v_)
        lambda1 = et1(c1, neurons_1.v_)

        g2 = e2 * lambda2
        g1 = e1 * lambda1

        v1 = np.maximum(gamma*v1, g1**2)
        v2 = np.maximum(gamma*v2, g2**2)

        m1 += g1
        m2 += g2

        count += 1
        if count == nt_b:
            # 重みの更新
            lr1 = r0/np.sqrt(v1+1e-8)
            lr2 = r0/np.sqrt(v2+1e-8)
            dW1 = np.clip(lr1*m1*dt, -1e-3, 1e-3)
            dW2 = np.clip(lr2*m2*dt, -1e-3, 1e-3)
            connect_1.W = np.clip(connect_1.W+dW1, -0.1, 0.1)
            connect_2.W = np.clip(connect_2.W+dW2, -0.1, 0.1)

            # リセット
```

```
            m1 = np.zeros((N_mid, N_in))
            m2 = np.zeros((N_out, N_mid))
            v1 = np.zeros((N_mid, N_in))
            v2 = np.zeros((N_out, N_mid))
            count = 0

        # 保存
        if i == num_iter-1:
            current_arr[:, t] = c2
            voltage_arr[:, t] = neurons_2.v_
            error_arr[:, t] = e2
            lambda_arr[:, :, t] = lambda2

    cost_arr[i] = cost
    print("\n   cost:", cost)
```

なお, `r` は重みの係数ですが, これを減衰させる (つまり weight decay) するとパフォーマンスが上がったので, 今回冒頭に入れています. また, `lr` は更新時の値を用いていますが, これは ANN において入力ごとの勾配を加算し, 重みの更新はミニバッチ内の全ての要素に対して同じ学習率で行うということに対応します. また, Feedback alignment の場合は誤差逆伝搬に `e1 = np.dot(B, e2) / N_mid` を用います.

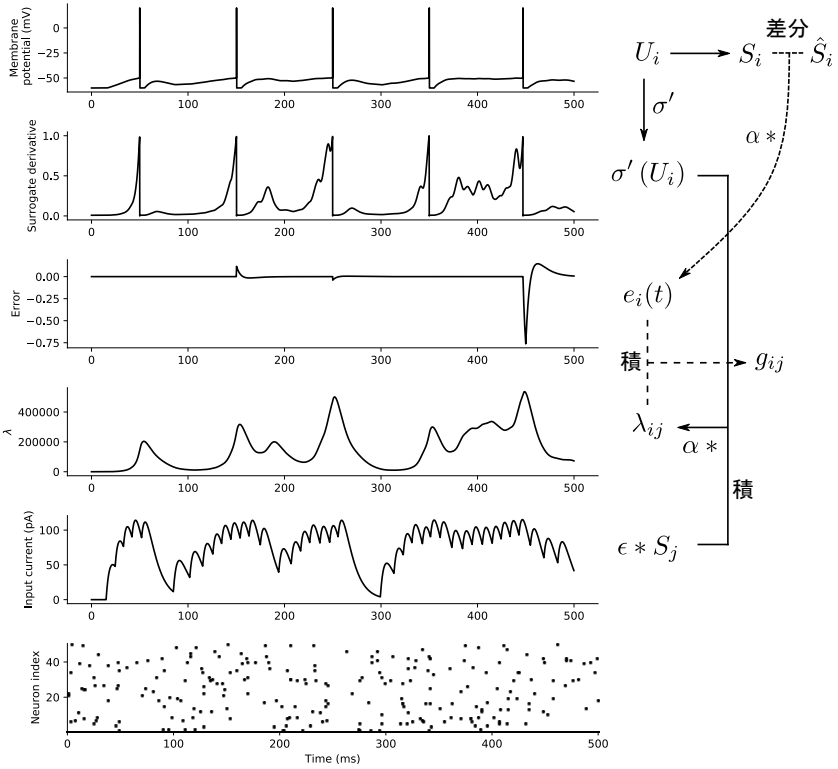
結果の描画

最後に結果を描画します. 描画するのは出力層の膜電位 U_i, 高速シグモイドによる膜電位の微分の近似 $\sigma'(U_i)$, 出力層における誤差信号 e_i, 適格度トレース λ_{ij}, 2 層目の $j = 0$ 番目のシナプス後電流 $\epsilon * S_j$, 入力のポアソンスパイク, 誤差関数の推移です.

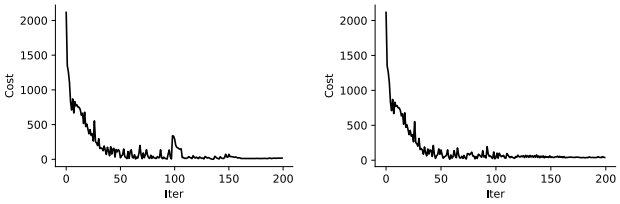
```
t = np.arange(nt)*dt*1e3


plt.figure(figsize=(8, 10))
plt.subplot(6,1,1)
plt.plot(t, voltage_arr[0])
plt.ylabel('Membrane\n potential (mV)')
plt.subplot(6,1,2)
```

```
plt.plot(t, et1.surrogate_derivative_fastsigmoid(u=voltage_arr[0]))
plt.ylabel('Surrogate derivative')
plt.subplot(6,1,3)
plt.plot(t, error_arr[0])
plt.ylabel('Error')
plt.subplot(6,1,4)
plt.plot(t, lambda_arr[0, 0], color="k")
plt.ylabel('$\lambda$')
plt.subplot(6,1,5)
plt.plot(t, current_arr[0], color="k")
plt.ylabel('Input current (pA)')
plt.subplot(6,1,6)
for i in range(N_in):
    plt.plot(t, x[:, i]*(i+1), 'ko', markersize=2)
plt.xlabel('Time (ms)'); plt.ylabel('Neuron index')
plt.xlim(0, t.max()); plt.ylim(0.5, N_in+0.5)
plt.show()

plt.figure(figsize=(4, 3))
plt.plot(cost_arr, color="k")
plt.xlabel('Iter'); plt.ylabel('Cost')
plt.show()
```

結果は図 4.1 のようになります．また, Feedback alignment の場合と比較した誤差関数の推移は図 4.2 のようになります．



▲ 図 4.1　SuperSpike 法の結果とアルゴリズム．上から出力層の膜電位 U_i, 高速シグモイドによる膜電位の微分の近似 $\sigma'(U_i)$, 出力層における誤差信号 e_i, 適格度トレース λ_{ij}, 2 層目の $j = 0$ 番目のシナプス後電流 $\epsilon * S_j$, 入力のポアソンスパイクを表します．



▲ 図 4.2　誤差関数の推移．(左) 対称フィードバックの場合．(右)Feedback alignment の場合

4.2 RNN としての SNN の BPTT を用いた教師あり学習

この節では発火率ベースの Recurrent neural networks (RNN) の一種のアーキテクチャとして Spiking neural networks を構成し, **Backpropagation Through Time (BPTT) 法**を用いて教師あり学習をする方法について説明します. これにより, Tensorflow や Pytorch, Chainer などの通常の ANN のフレームワークで SNN を学習させることができます. この節では, **Spiking Neural Unit** (SNU) と呼ばれる, LSTM や GRU のような状態 (state) を持つ RNN のユニットを紹介します (Woźniak et al., 2018). 他の類似の研究としては (Wu et al., 2018; Neftci et al., 2019) などを参照してください[*10].

Spiking Neural Unit (SNU) は次式で表される, Current-based LIF ニューロンが元となっています.

$$\tau \frac{dV_m(t)}{dt} = -V_m(t) + RI(t) \tag{4.16}$$

ただし, $\tau = RC$ です. ここでは静止膜電位を 0 としています[*11]. これを Euler 近似でタイムステップ Δt で離散化し,

$$V_{m,t} = \frac{\Delta t}{C} I_t + \left(1 - \frac{\Delta t}{\tau}\right) V_{m,t-1} \tag{4.17}$$

となります. V_m が一定の閾値 V_{th} を超えるとニューロンは発火し, 膜電位はリセットされて静止膜電位に戻ります. 閾値を超えると発火, ということを表すために次式で表される変数 y_t を導入し, ステップ関数により発火した際に $y_t = 1$ となるようにします.

$$y_t = f\left(V_{m,t} - V_{\mathrm{th}}\right) \tag{4.18}$$

ただし, $f(\cdot)$ はステップ関数で,

$$f(x) = \begin{cases} 1 & (x > 0) \\ 0 & (x \leq 0) \end{cases} \tag{4.19}$$

と表されます. さらに $y_{t-1} = 1$ なら膜電位がリセットされるように $(1 - y_{t-1})$ を膜電位 $V_{m,t-1}$ に乗じて膜電位を更新します.

$$V_{m,t} = \frac{\Delta t}{C} I_t + \left(1 - \frac{\Delta t}{\tau}\right) V_{m,t-1} \cdot (1 - y_{t-1}) \tag{4.20}$$

ここで, 膜電位を $V_{m,t} \to s_t$ とし, 入力電流を $I_t \to W x_t$ とします (ただし, x_t は入力, W は結合重みの行列). さらに以前の膜電位を保持する割合を表す変数として $l(\tau) = (1 - \frac{\Delta t}{\tau})$

[*10] 特に (Neftci et al., 2019) には Jupyter Notebook も用意されています (`https://github.com/fzenke/spytorch`). サーベイも詳しく参考になります.

[*11] 静止膜電位を考慮する場合は, 定数項 V_{rest} を加えるとよいです.

を定義します. このとき, SNU の状態を計算する式は

$$s_t = g\left(Wx_t + l(\tau) \odot s_{t-1} \odot (1 - y_{t-1})\right) \tag{4.21}$$
$$y_t = h(s_t + b) \tag{4.22}$$

となります. ただし, $g(\cdot)$ は ReLU 関数, $h(\cdot)$ はステップ関数です[*12]. このように LSTM のような状態 y_t を上手く設定することで, RNN のユニットとしてモデル化できています.

しかし, このモデルはステップ関数を含むため, このままでは学習できません. というのも, ステップ関数は微分すると Dirac のデルタ関数となり, 誤差逆伝搬できないためです. そこで Woźniak らはステップ関数の**疑似勾配** (pseudo-derivative) として tanh の微分を用いています. なお疑似勾配と同じ概念が, (Neftci et al., 2019) では**代理勾配** (Surrogate Gradient) と呼ばれています.

実装方法としてはステップ関数を新しく定義し, 逆伝搬時の勾配に tanh の微分などの関数を用いるようにします. コードは示しませんが[*13], Chainer で実装した結果を示します. この実装では 2 値化した MNIST データセットをポアソン過程モデルでエンコードし (これを Jittered MNIST と呼びます), 1 つの画像につき 10 ms (10 ステップ) の間, ネットワークにエンコードしたポアソンスパイクを入力します. ネットワークは 4 層 (ユニット数は順に 784-256-256-10) から構成され, 最後の層で最も発火率の高いユニットに対応するラベルを, 刺激画像の予測ラベルとします. 注意点として, このネットワークではシナプス入力を考えておらず (シナプスフィルターがなく), 重みづけされたスパイク列が直接次の層のユニットに伝わります.

その他, 論文の実装と変えたこととしては 4 点あります. 1 点目に, ReLU だと dying ReLU が起こっているようで学習がうまく進まなかったので, 活性化関数として Exponential Linear Unit (ELU) を代わりに用いました[*14]. 2 点目に, ステップ関数の疑似勾配を, tanh の微分では学習が進まなかったので, hard sigmoid のような関数の微分

$$f'(x) = \begin{cases} 1 & (-0.5 < x < 0.5) \\ 0 & (\text{otherwise}) \end{cases} \tag{4.23}$$

を用いました. 3 点目に, 損失関数を変更しました. 平均二乗誤差 (Mean squared error; MSE) だと学習が進まなかったので, 出力ユニットの全スパイク数の和を取り, Softmax をかけて, 正解ラベルとの交差エントロピー (cross entropy) を取りました. また出力ユニットの発火数を抑えるため, 代謝コスト (metabolic cost) を損失に加えました. これに

[*12] なお, $h(\cdot)$ をシグモイド関数とする soft-SNU も提案されています. この場合, 特に新しく関数を定義する必要はありません.

[*13] `https://github.com/takyamamoto/SNU_Chainer` を参考にしてください.

[*14] この変更は発火特性に影響を与えません.

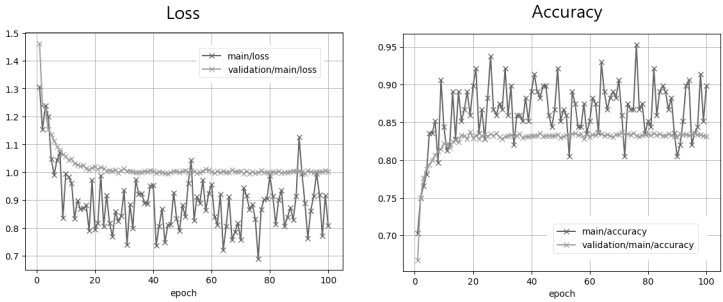
は正則化の効果もあります. 出力層の i 番目のユニットの出力を $y_t^{(i)}$ とすると, 代謝コスト C_{met} は

$$C_{\mathrm{met}} = \frac{10^{-2}}{N_t \cdot N_{\mathrm{out}}} \sum_{t=1}^{N_t} \sum_{i=1}^{N_{\mathrm{out}}} \left(y_t^{(i)} \right)^2 \tag{4.24}$$

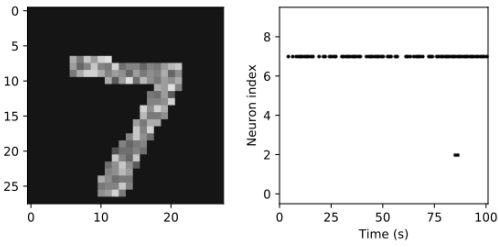
となります. ただし, N_t はシミュレーションの総ステップ数, N_{out} は出力ユニットの数 (今回だと 10 個) です. あまり大きくすると, 分類誤差よりも代謝コストの方が大きくなってしまうので低めに設定しました. 4 点目に, optimizer を Adam に変更しました.

この実装により 100 epoch 学習を行った結果を示します. 図は誤差と正解率の学習時における変化です.

この手法の欠点としてはナイーブに BPTT を実行するのであまりシミュレーションの時間ステップを長くできないということが挙げられます. ただし, 通常の ANN のフレームワークを用いることができるというのは大きな利点であると思います.



▲ 図 4.3 (左) 誤差の変化, (右) 正解率の変化. 100 epoch 目における validation の正解率は 83% 程度となりました.



▲ 図 4.4 (左) 入力のポアソンスパイクの時間軸における和. (右) 出力ユニットの活動. 7 番のニューロンがよく活動していることが分かります.

第 5 章

STDP 則による教師なし学習

5.1 STDP (Spike-timing-dependent plasticity) 則

5.1.1 Pair-based STDP 則

Spike-timing-dependent plasticity (STDP, スパイクタイミング依存性可塑性) はシナプス前細胞と後細胞の発火時刻の差によってシナプス強度が変化するという現象です (Markram et al. 1997; Bi & Poo 1998). 典型的な STDP 側は **Pair-based STDP 則**と呼ばれ, シナプス前細胞と後細胞の 2 つのスパイクのペアの発火時刻によって LTP(long-term potentiation, 長期増強) や LTD(long-term depression, 長期抑圧) が起こります. この節ではこの Pair-based STDP 則について説明します.

シナプス後細胞におけるスパイク (postsynaptic spike) の発生時刻 $t_{\rm post}$ とシナプス前細胞におけるスパイク (presynaptic spike) の発生時刻 $t_{\rm pre}$ の差を $\Delta t_{\rm spike} = t_{\rm post} - t_{\rm pre}$ とします[*1]. $\Delta t_{\rm spike}$ はシナプス前細胞, 後細胞の順で発火すれば正, 逆なら負となります. Pair-based STDP 則では, シナプス前細胞から後細胞へのシナプス強度 (w)[*2]の変化 Δw は $\Delta t_{\rm spike}$ に依存的に以下の式に従って変化します (Song et al,, 2000).

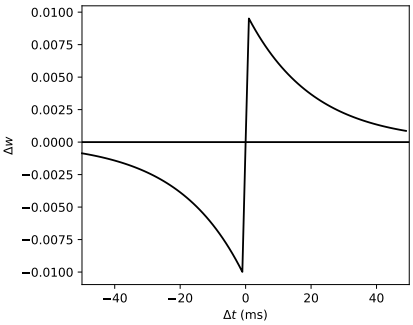
$$\Delta w = \begin{cases} A_+ \exp\left(-\dfrac{\Delta t_{\rm spike}}{\tau_+}\right) & (\Delta t_{\rm spike} > 0) \\ -A_- \exp\left(-\dfrac{|\Delta t_{\rm spike}|}{\tau_-}\right) & (\Delta t_{\rm spike} < 0) \end{cases} \tag{5.1}$$

A_+, A_- は正の定数, または重み依存的な関数 (後述) です. $\Delta t_{\rm spike} > 0$ のときは LTP が起こり, $\Delta t_{\rm spike} < 0$ のときは LTD が起こります. このタイプの STDP 則は **Hebbian**

[*1] $\Delta t_{\rm spike}$ の定義は元々は逆になっており, (Song et al,, 2000) では $\Delta t_{\rm spike} = t_{\rm pre} - t_{\rm post}$ としています. また, 添え字は離散時のタイムステップと混同しないために付けています.

[*2] シナプス強度 w に添え字をつけていませんが, この場合はシナプス前細胞と後細胞の 2 つの細胞しかないと仮定して考えています.

STDP と呼ばれ, Hebb 則[*3]に従うシナプス強度の変化が起こります[*4]. $A_+ = 0.01$, $A_-/A_+ = 1.05$, $\tau_+ = \tau_- = 20$ ms としたときの Δt_{spike} に対する Δw は図 5.1 のようになります.



▲ 図 5.1 STDP

以下は図 5.1 を描画するためのコードです[*5]

```
tau_p = tau_m = 20 #ms
A_p = 0.01
A_m = 1.05*A_p
dt = np.arange(-50, 50, 1) #ms
dw = A_p*np.exp(-dt/tau_p)*(dt>0) - A_m*np.exp(dt/tau_p)*(dt<0)

plt.figure(figsize=(5, 4))
plt.plot(dt, dw)
plt.hlines(0, -50, 50); plt.xlim(-50, 50)
plt.xlabel("$\Delta t$ (ms)"); plt.ylabel("$\Delta w$")
plt.show()
```

[*3] 「シナプス前細胞が発火してからシナプス後細胞が発火することによりシナプス結合が増強される」という法則です. 1949 年に Donald Hebb により提唱されました.

[*4] Hebb 則に従わない STDP もあり, 例えば LTP と LTD の挙動が逆のものを **Anti-Hebbian STDP** と呼びます (Bell et al., 1997 など参照).

[*5] コードは`./SingleFileSimulations/STDP/stdp.py` です.

5.1.2 オンライン STDP 則

単に 2 つのニューロンを考えるなら上で紹介した式でも良いのですが, ネットワーク全体を考えると実装は複雑になり効率的ではありません. また, スパイク発生時刻を記憶しておくことは生物学的に妥当ではありません. そこで, スパイク活動のトレース (trace) というローカル変数を用いて STDP を記述してみましょう.

$$\frac{dx_{\mathrm{pre}}}{dt} = -\frac{x_{\mathrm{pre}}}{\tau_+} + \sum_{t_{\mathrm{pre}}^{(i)}<t} \delta\left(t - t_{\mathrm{pre}}^{(i)}\right) \tag{5.2}$$

$$\frac{dx_{\mathrm{post}}}{dt} = -\frac{x_{\mathrm{post}}}{\tau_-} + \sum_{t_{\mathrm{post}}^{(j)}<t} \delta\left(t - t_{\mathrm{post}}^{(j)}\right) \tag{5.3}$$

とします. ただし, $t_{\mathrm{pre}}^{(i)}$ はシナプス前細胞の i 番目のスパイク, $t_{\mathrm{post}}^{(j)}$ はシナプス後細胞の i 番目のスパイクを意味します. また, x_{pre}, x_{post} はそれぞれシナプス前細胞, 後細胞のスパイクのトレースです. トレースはそれぞれの細胞においてスパイクが発生したときに 1 増加し[*6], それ以外では時定数 τ_+, τ_- で指数関数的に減少します. これは既に 1 章で説明した単一指数関数型シナプスと同じです. これらの生理学的解釈ですが, x_{pre} は NMDA 受容体のイオンチャネルの開口割合, x_{post} は**逆伝搬活動電位** (back-propagating action potential; bAP) [*7]や bAP によって活性化された電位依存性 Ca^{2+} チャネルによる Ca^{2+} の流入と捉えることができます (cf. “標準生理学”).

そして, $x_{\mathrm{pre}}, x_{\mathrm{post}}$ を用いて重みの更新式は

$$\frac{dw}{dt} = A_+ x_{\mathrm{pre}} \cdot \underbrace{\sum_{t_{\mathrm{post}}^{(j)}<t} \delta\left(t - t_{\mathrm{post}}^{(j)}\right)}_{\text{シナプス後細胞のスパイク}} - A_- x_{\mathrm{post}} \cdot \underbrace{\sum_{t_{\mathrm{pre}}^{(i)}<t} \delta\left(t - t_{\mathrm{pre}}^{(i)}\right)}_{\text{シナプス前細胞のスパイク}} \tag{5.5}$$

と表せます.

[*6] トレースの値域を $0 \leq x \leq 1$ に制限するため, スパイクが発生したとき 1 にリセットするという場合もあります (Morrison et al., 2008). その場合は

$$x(t + \Delta t) = \left(1 - \frac{\Delta t}{\tau}\right) x(t) \cdot (1 - \delta_{t,t'}) + \delta_{t,t'} \tag{5.4}$$

のようにします (t' はスパイクの発生時刻).

[*7] bAP は細胞体からシナプスのある樹状突起へと逆行性に伝搬する活動電位のことです. なお, 誤差逆伝搬法 (backpropagation) とは関係がありません.

これらの式を Euler 法によりタイムステップ Δt で離散化すると,

$$x_{\mathrm{pre}}(t+\Delta t) = \left(1 - \frac{\Delta t}{\tau_+}\right) \cdot x_{\mathrm{pre}}(t) + \delta_{t,t^{(i)}_{\mathrm{pre}}} \tag{5.6}$$

$$x_{\mathrm{post}}(t+\Delta t) = \left(1 - \frac{\Delta t}{\tau_-}\right) \cdot x_{\mathrm{post}}(t) + \delta_{t,t^{(j)}_{\mathrm{post}}} \tag{5.7}$$

$$w(t+\Delta t) = w(t) + A_+ x_{\mathrm{pre}} \cdot \delta_{t,t^{(j)}_{\mathrm{post}}} - A_- x_{\mathrm{post}} \cdot \delta_{t,t^{(i)}_{\mathrm{pre}}} \tag{5.8}$$

となります. ただし, $\delta_{t,t'}$ は Kronecker の delta 関数で, $t = t'$ のときに 1, それ以外は 0 となります. $\delta_{t,t'}$ は実装時においてスパイクが起こったときに 1, その他は 0 を取る変数を用いると良いでしょう.

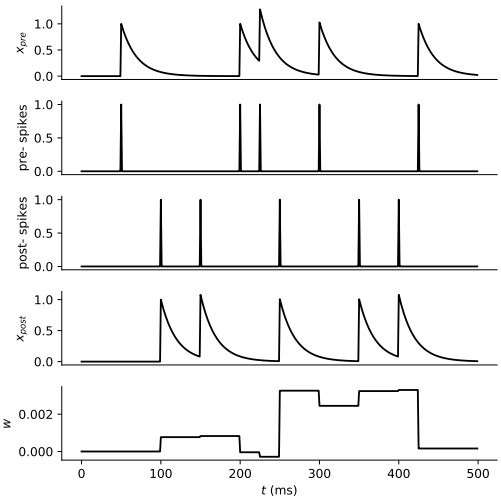
それでは, Online STDP を実装してみましょう[*8].

```
#定数
dt = 1e-3 #sec
T = 0.5 #sec
nt = round(T/dt)
tau_p = tau_m = 2e-2 #ms
A_p = 0.01; A_m = 1.05*A_p

#pre/postsynaptic spikes
spike_pre = np.zeros(nt); spike_pre[[50, 200, 225, 300, 425]] = 1
spike_post = np.zeros(nt); spike_post[[100, 150, 250, 350, 400]] = 1

#記録用配列
x_pre_arr = np.zeros(nt); x_post_arr = np.zeros(nt)
w_arr = np.zeros(nt)

#初期化
x_pre = x_post = 0 # pre/post synaptic trace
w = 0 # synaptic weight

#Online STDP
for t in range(nt):
```

[*8] コードは./SingleFileSimulations/STDP/stdp2.py です.

```
    x_pre = x_pre*(1-dt/tau_p) + spike_pre[t]
    x_post = x_post*(1-dt/tau_m) + spike_post[t]
    dw = A_p*x_pre*spike_post[t] - A_m*x_post*spike_pre[t]
    w += dw #重みの更新

    x_pre_arr[t] = x_pre
    x_post_arr[t] = x_post
    w_arr[t] = w

# 描画
time = np.arange(nt)*dt*1e3
def hide_ticks(): #上と右の軸を表示しないための関数
    plt.gca().spines['right'].set_visible(False)
    plt.gca().spines['top'].set_visible(False)
    plt.gca().yaxis.set_ticks_position('left')
    plt.gca().xaxis.set_ticks_position('bottom')
plt.figure(figsize=(6, 6))
plt.subplot(5,1,1)
plt.plot(time, x_pre_arr, color="k")
plt.ylabel("$x_{pre}$"); hide_ticks(); plt.xticks([])
plt.subplot(5,1,2)
plt.plot(time, spike_pre, color="k")
plt.ylabel("pre- spikes"); hide_ticks(); plt.xticks([])
plt.subplot(5,1,3)
plt.plot(time, spike_post, color="k")
plt.ylabel("post- spikes"); hide_ticks(); plt.xticks([])
plt.subplot(5,1,4)
plt.plot(time, x_post_arr, color="k")
plt.ylabel("$x_{post}$"); hide_ticks(); plt.xticks([])
plt.subplot(5,1,5)
plt.plot(time, w_arr, color="k")
plt.xlabel("$t$ (ms)"); plt.ylabel("$w$"); hide_ticks()
plt.show()
```

結果は図 5.2 のようになります.



▲ 図 5.2　オンライン STDP 則 : (1 段目) シナプス前細胞のスパイクトレース (2 段目) シナプス前細胞のスパイク (3 段目) シナプス後細胞のスパイク (4 段目) シナプス後細胞のスパイクトレース (5 段目) 重みの変化 (初期値 0)

行列を用いたオンライン STDP 則の実装

この節ではシナプス前細胞と後細胞の数を一般化し, 今まで 2 つのニューロン間で考えていた STDP 則を行列計算で実装する方法について説明します.

まず, シナプス前細胞, 後細胞がそれぞれ N_{pre}, N_{post} あるとします. また, Kronecker の delta 関数の代わりに, スパイクが起こったときに 1, その他は 0 の値を取る明示的な変数 $\boldsymbol{s}(t)$ を用いることにします. ここで, シナプス前細胞, 後細胞についてスパイク変数は $\boldsymbol{s}_{\mathrm{pre}} \in \mathbb{R}^{N_{\mathrm{pre}}}, \boldsymbol{s}_{\mathrm{post}} \in \mathbb{R}^{N_{\mathrm{post}}}$ であり, スパイクのトレースは $\boldsymbol{x}_{\mathrm{pre}} \in \mathbb{R}^{N_{\mathrm{pre}}}, \boldsymbol{x}_{\mathrm{post}} \in \mathbb{R}^{N_{\mathrm{post}}}$ です. さらにシナプスから後細胞へのシナプス強度を $N_{\mathrm{post}} \times N_{\mathrm{pre}}$ 行列の W とします. このとき, Online STDP 則は

$$\boldsymbol{x}_{\mathrm{pre}}(t+\Delta t) = \left(1 - \frac{\Delta t}{\tau_+}\right) \cdot \boldsymbol{x}_{\mathrm{pre}}(t) + \boldsymbol{s}_{\mathrm{pre}}(t) \tag{5.9}$$

$$\boldsymbol{x}_{\mathrm{post}}(t+\Delta t) = \left(1 - \frac{\Delta t}{\tau_-}\right) \cdot \boldsymbol{x}_{\mathrm{post}}(t) + \boldsymbol{s}_{\mathrm{post}}(t) \tag{5.10}$$

$$W(t+\Delta t) = W(t) + A_+ \boldsymbol{s}_{\mathrm{post}}(t)(\boldsymbol{x}_{\mathrm{pre}}(t))^\intercal - A_- \boldsymbol{x}_{\mathrm{post}}(t)(\boldsymbol{s}_{\mathrm{pre}}(t))^\intercal \tag{5.11}$$

と書けます. ただし, $^\intercal$ を転置記号とし, $\boldsymbol{x}$ を列ベクトル, $\boldsymbol{x}^\intercal$ を行ベクトルとしています.

これらを用いて Online STDP 則と元の STDP の式が一致しているかの確認をしてみましょう[*9]. タイムステップ dt を 1ms, シミュレーション時間 T は 50ms とし, シミュレーションステップ数 nt と同数のシナプス前細胞, 2 つのシナプス後細胞があるとします. それぞれのシナプス前細胞は dt だけずれて発火し[*10](つまり発火時刻の範囲は [0ms, 50ms]), 2 つの後細胞は $t = 0, 50\text{ms}$ に発火します. こうすることで, 発火の時間差として $[-50\text{ms}, 50\text{ms}]$ が生まれます.

```
dt = 1e-3; T = 5e-2 #sec
nt = round(T/dt)
N_pre = nt; N_post = 2
tau_p = tau_m = 2e-2 #ms
A_p = 0.01; A_m = 1.05*A_p

# pre/postsynaptic spikes
spike_pre = np.eye(N_pre) #単位行列で dt ごとに発火するニューロンを N 個作成
spike_post = np.zeros((N_post, nt))
spike_post[0, -1] = spike_post[1, 0] = 1

# 初期化
x_pre = np.zeros(N_pre)
x_post = np.zeros(N_post)
W = np.zeros((N_post, N_pre))

for t in range(nt):
    # 1 次元配列 -> 縦ベクトル or 横ベクトル
    spike_pre_ = np.expand_dims(spike_pre[:, t], 0) # (1, N)
    spike_post_ = np.expand_dims(spike_post[:, t], 1) # (2, 1)
    x_pre_ = np.expand_dims(x_pre, 0) # (1, N)
    x_post_ = np.expand_dims(x_post, 1) # (2, 1)

    # Online STDP
```

[*9] コードは./SingleFileSimulations/STDP/stdp3.py です.

[*10] この場合, シナプス前細胞のスパイクを表す spike_pre 行列として nt 次単位行列を与えればよいです.

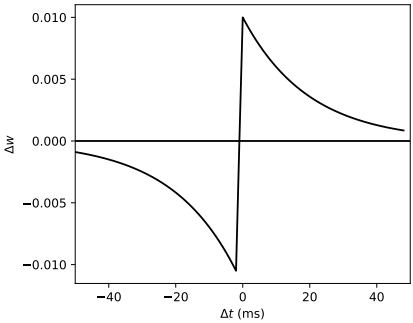
```
    dW = A_p*np.matmul(spike_post_, x_pre_)
    dW -= A_m*np.matmul(x_post_, spike_pre_)
    W += dW

    # Update
    x_pre = x_pre*(1-dt/tau_p) + spike_pre[:, t]
    x_post = x_post*(1-dt/tau_m) + spike_post[:, t]

# 結果
delta_w = np.zeros(nt*2-1) # スパイク時間差 = 0ms が重複
delta_w[:nt] = W[0, :]; delta_w[nt:] = W[1, 1:]

# 描画
time = np.arange(-T, T-dt, dt)*1e3
plt.figure(figsize=(5, 4))
plt.plot(time, delta_w[::-1])
plt.hlines(0, -50, 50)
plt.xlabel("$\Delta t$ (ms)")
plt.ylabel("$\Delta w$")
plt.xlim(-50, 50)
plt.show()
```

このコードを実行すると図 5.3 のようになります.

5.1.3 重み依存的な STDP

生理学的にはシナプス強度 w には $w_{\min} < w < w_{\max}$ というような制限 (bound) が存在すると考えられます[*11]. 多くの場合では $w_{\min} = 0$ となっているので, 以下では $w \in [0, w_{\max}]$ となる場合を考えます. また, 前節までは正の定数としていた A_+, A_- を重み依存的な関数であるとします (つまり $A_\pm = A_\pm(w)$ とします).

[*11] 受容体の数が際限なく増加したり減少したりすることはないと考えられるためです. もし LTP によりシナプス強度が限りなく増大した場合, シナプス後細胞の発火頻度が高くなり, 実際には発火を誘発していないシナプス前細胞とのシナプス強度も大きくなってしまいます. LTP の暴走を防ぐための機構の 1 つとして**恒常性可塑性** (homeostatic scaling) または**シナプススケーリング** (synaptic scaling) と呼ばれる現象があります.



▲ 図 5.3　Online STDP

重みの制限には**ソフト制限 (soft bound)** と**ハード制限 (hard bound)** があります (Gerstner & Kistler, 2002, Chapter 11). ソフト制限は, 重みが上界 (または下界) に近づくにつれ重みの変化が小さくなるというものです.

$$A_+(w) = \eta_+ \cdot (w_{\max} - w) \tag{5.12}$$
$$A_-(w) = \eta_- \cdot w \tag{5.13}$$

ここで η_+, η_- は正の値で, **学習率 (learning rate)** を意味します.

次に, ハード制限は重みが上限 (または下限) に達した際に, 重みが増加 (または減少) しないというものです. Heaviside の階段関数 $\Theta(x)$ (ただし $x < 0$ で $\Theta(x) = 0$, $x \geq 0$ で $\Theta(x) = 1$) を用いて

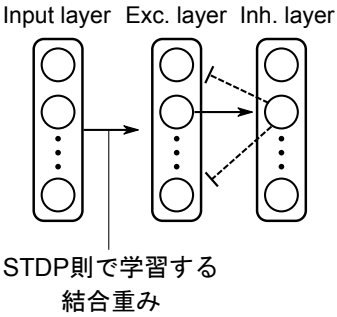
$$A_+(w) = \eta_+ \cdot \Theta(w_{\max} - w) \tag{5.14}$$
$$A_-(w) = \eta_- \cdot \Theta(-w) \tag{5.15}$$

となります.

5.2 STDP 則と 2 層 WTA ネットワークによる教師なし学習

この節では STDP 則と **Winner-take-all (WTA)** の機構を用いた**自己組織化マップ** (self-organizing map, SOM) による教師なし学習の研究について紹介します (Diehl & Cook, 2015)[*12]. Diehl らは, 提案するモデルにより, MNIST データセットにおいて教師なし学習で, テストデータに対して 95% の予測精度を出しています. 現在でも Diehl らの研究を発展させて, Convolutional SNN の学習に応用する研究が進んでいます.

WTA (Winner-take-all) というのはネットワーク内のニューロンが互いに抑制しあう[*13]ことで, 最も発火率の高いニューロン以外は抑制されて発火しないようになる, という機構です[*14]. WTA の機構による学習は**競合学習** (Competitive learning) と呼んだりします. Diehl らが提案したネットワークは 2 層から成り立っており, 1 層目は入力層 (28×28 =784 ニューロン) で, MNIST データセットの画像 1 画素に 1 つのニューロンが対応します. このとき, 画像はポアソン過程モデルでスパイク列に符号化 (encoding) されます. 2 層目は興奮性ニューロンおよび同数の抑制性ニューロンから成ります (図 5.4). 1 つの興奮性ニューロンは 1 つの抑制性ニューロンに投射し, 抑制性ニューロンは自分に入力したニューロン以外の興奮性ニューロンを抑制します. こうすることで側抑制の仕組みが実装されています.



▲ 図 5.4 (Diehl & Cook, 2015) で提案されたネットワーク. 1 層目が入力層, 2 層目が興奮性ニューロン層と抑制性ニューロン層から構成されます. 興奮性ニューロンと抑制性ニューロンの間の結合重みは固定で, 入力層から興奮性ニューロン層へ投射される結合重みを STDP 則で学習します.

[*12] Brian を用いた実装は `https://github.com/peter-u-diehl/stdp-mnist`, Brian2 を用いた実装は `https://github.com/zxzhijia/Brian2STDPMNIST` で公開されています (ただし Python2).

[*13] これを**側抑制** (lateral inhibition) と言います.

[*14] 例えば ANN で用いられる Softmax や Max-pooling の操作は WTA のメカニズムのモデル化の 1 つといえます.

5.2.1 ニューロンとシナプスのモデル

ニューロンとシナプスのモデルとしては Conductance-based モデルを用いています.

ニューロンのモデル

興奮性ニューロンと抑制性ニューロンの膜電位 V は次の式に従います.

$$\tau \frac{dV}{dt} = (E_{\text{rest}} - V) + g_e (E_{\text{exc}} - V) + g_i (E_{\text{inh}} - V) \tag{5.16}$$

これは通常の Conductance-based モデルと同じ式ですが, ネットワークの発火率を一定に保つため, 興奮性ニューロンの膜電位の発火閾値は発火のたびに $\theta = 0.05\text{mV}$ 上昇するとします. 発火の無い場合は時定数 10^7 ms で減衰します. 著者実装では 350 ms 間, 画像をエンコードしたスパイク列を入力し, 変数をリセットするために 150 ms 間, 何も入力しないブランク期間を設定しています. この間に膜電位等は全てリセットされますが, 発火閾値のみ, 減衰するだけに留まります. さらに, 興奮性ニューロンの発火数が少ない場合には, 入力のポアソンスパイクの発火率を上げ, 再度同じ画像を提示します.

また, 興奮性ニューロンの膜電位の時定数としては生理学的に逸脱した 100 ms という値を用いています. これは時定数を大きくすることで, 多くのスパイク入力を積分することができ, ノイズによる影響を減らすことができると説明されています.

まず, 準備としてこのようなニューロンを実装しておきます[*15]. コード自体は 2 章で紹介した `Conductance-based LIF` に修正を加えたものです. 後で使用するために, コードは `./Models/Neurons.py` に記述して保存しておきます.

```
class DiehlAndCook2015LIF:
    def __init__(self, N, dt=1e-3, tref=5e-3, tc_m=1e-1,
                 vrest=-65, vreset=-65, init_vthr=-52, vpeak=20,
                 theta_plus=0.05, theta_max=35, tc_theta=1e4,
                 e_exc=0, e_inh=-100):
        self.N = N
        self.dt = dt
        self.tref = tref
        self.tc_m = tc_m
        self.vreset = vreset
        self.vrest = vrest
```

[*15] コードは `./TrainingSNN/Models/Neurons.py` に含まれます.

```
        self.init_vthr = init_vthr
        self.theta = np.zeros(N)
        self.theta_plus = theta_plus
        self.theta_max = theta_max
        self.tc_theta = tc_theta
        self.vpeak = vpeak

        self.e_exc = e_exc # 興奮性シナプスの平衡電位
        self.e_inh = e_inh # 抑制性シナプスの平衡電位

        self.v = self.vreset*np.ones(N)
        self.vthr = self.init_vthr
        self.v_ = None
        self.tlast = 0
        self.tcount = 0

    def initialize_states(self, random_state=False):
        if random_state:
            self.v = self.vreset + \
                     np.random.rand(self.N)*(self.vthr-self.vreset)
        else:
            self.v = self.vreset*np.ones(self.N)
        self.vthr = self.init_vthr
        self.theta = np.zeros(self.N)
        self.tlast = 0
        self.tcount = 0

    def __call__(self, g_exc, g_inh):
        I_synExc = g_exc*(self.e_exc - self.v)
        I_synInh = g_inh*(self.e_inh - self.v)
        dv = (self.vrest - self.v + I_synExc + I_synInh) / self.tc_m
        v = self.v+((self.dt*self.tcount)>(self.tlast+self.tref))*dv*self.dt

        s = 1*(v>=self.vthr) #発火時は 1, その他は 0 の出力
```

```
        # 閾値の更新
        theta = (1-self.dt/self.tc_theta)*self.theta + self.theta_plus*s
        self.theta = np.clip(theta, 0, self.theta_max)
        self.vthr = self.theta + self.init_vthr

        self.tlast = self.tlast*(1-s) + self.dt*self.tcount*s
        v = v*(1-s) + self.vpeak*s #閾値を超えると膜電位を vpeak にする
        self.v_ = v #発火時の電位も含めて記録するための変数
        self.v = v*(1-s) + self.vreset*s  #発火時に膜電位をリセット
        self.tcount += 1

        return s
```

Conductance-based LIF に追加した部分は閾値の更新に関する部分です．実際に発火を判断する閾値は `self.theta` と `self.init_vthr` を加算した `self.vthr` です．このうち, 変動するのは `self.theta` の項ですが, これはスパイクトレースと同様に発火が起こると上昇し, それ以外では減衰します．ただし, 閾値が上がりすぎることを防ぐため, 上限を `self.theta_max` として `np.clip()` により制限しています．

シナプスのモデル

シナプスには Conductance-based の単一指数関数型シナプス (single exponential synapse) を用いています．スパイクが入るとコンダクタンス g は w だけあがり, その他では以下のように減衰します

$$\tau_x \frac{dg_x}{dt} = -g_x \tag{5.17}$$

ただし, $x = e, i$ で, それぞれ興奮性, 抑制性を意味する添え字です．なお, この実装自体は第 2 章で述べたものと変わりません．

5.2.2 興奮性ニューロンのラベリング

このネットワークには出力層がなく, 通常とは異なる形式で画像を分類しています．まず, 訓練時において興奮性ニューロンの活動を全て記録します．次に, 画像に元々されていたラベルを用いて, 興奮性の各ニューロンの各ラベルの画像への平均発火数を計算します．このとき, 各ニューロンにおいて最も発火数が多かったラベルを求め, そのラベルを各ニューロンに割り当てます．そして, 割り当てたラベルを用い, 入力画像に対する各ラベ

ルに割り当てられたニューロンの平均発火数を求めます．このとき平均発火数が最も高いニューロン群のラベルが，入力画像の予測ラベルとなります．また，推論時には訓練時においてニューロンに割り当てたラベルを用います．

それではニューロンへラベルを割り当てる関数 (assign_labels) を実装してみましょう[*16]． assign_labels は 5 つの変数を引数に取ります． spikes は，サイズが (n_samples, n_neurons) の 2 次元配列で，各サンプルにおいて各興奮性ニューロンが何回発火したかを記録したものです．labels はサイズが (n_samples,) の配列で，各サンプルの教師ラベルを表します．n_labels はラベルの数です．今回は MNIST なので 10 個です．rates はサイズが (n_samples, n_neurons) の配列で，各興奮性ニューロンの各ラベルに対する発火率を表します．rates は 2 度目の計算以降に引数として渡すと，alpha:1 の割合で過去の rates との指数平均を取ります．

```
def assign_labels(spikes, labels, n_labels, rates=None, alpha=1.0):
    n_neurons = spikes.shape[1]

    if rates is None:
        rates = np.zeros((n_neurons, n_labels)).astype(np.float32)

    # 時間の軸でスパイク数の和を取る
    for i in range(n_labels):
        # サンプル内の同じラベルの数を求める
        n_labeled = np.sum(labels == i).astype(np.int16)

        if n_labeled > 0:
            # label == i のサンプルのインデックスを取得
            indices = np.where(labels == i)[0]

            # label == i に対する各ニューロンごとの平均発火率を計算
            #(前回の発火率との移動平均)
            rates[:, i] = alpha*rates[:, i] + \
                          (np.sum(spikes[indices], axis=0)/n_labeled)
```

[*16] 以降のコードは特に明記が無い限り ./TrainingSNN/LIF_WTA_STDP_MNIST.py に含まれます．また一部，BindsNET の実装を参考にしています．

```
    sum_rate = np.sum(rates, axis=1)
    sum_rate[sum_rate==0] = 1
    # クラスごとの発火頻度の割合を計算する
    proportions = rates / np.expand_dims(sum_rate, 1) # (n_neurons, n_labels)
    proportions[proportions != proportions] = 0  # Set NaNs to 0

    # 最も発火率が高いラベルを各ニューロンに割り当てる (n_neurons,)
    assignments = np.argmax(proportions, axis=1).astype(np.uint8)

    return assignments, proportions, rates
```

ここで assignments はサイズが (n_neurons,) の配列で, 各ニューロンに割り当てられたラベルを表します. これを用いて, 各サンプルに対してラベルを予測する関数 prediction を実装しましょう.

```
def prediction(spikes, assignments, n_labels):
    n_samples = spikes.shape[0]

    # 各サンプルについて各ラベルの発火率を見る
    rates = np.zeros((n_samples, n_labels)).astype(np.float32)

    for i in range(n_labels):
        # 各ラベルが振り分けられたニューロンの数
        n_assigns = np.sum(assignments == i).astype(np.uint8)
        if n_assigns > 0:
            # 各ラベルのニューロンのインデックスを取得
            indices = np.where(assignments == i)[0]

            # 各ラベルのニューロンのレイヤー全体における平均発火数を求める
            rates[:, i] = np.sum(spikes[:, indices], axis=1) / n_assigns

    # レイヤーの平均発火率が最も高いラベルを出力
    return np.argmax(rates, axis=1).astype(np.uint8) # (n_samples, )
```

prediction は 3 つの引数, spikes, assignments, n_labels を取ります. これらはそれぞれ先ほど説明した同名の配列と同じです. そして, 各ラベルに紐づけられたニューロンごとに平均発火数を計算し, 最も平均発火数の多かったラベルを入力サンプルのラベルとします.

5.2.3 MNIST データセットのスパイク列への変換

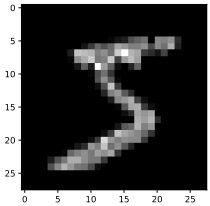
入力として用いるために, MNIST データセットをスパイク列へ変換する関数を書きましょう. MNIST データセットを読み込む関数を別に書いても良いですが, 煩雑となるため, ANN のライブラリに付属するデータ読み込み関数を利用してみます. ここでは例として Chainer の chainer.datasets.get_mnist() を用いますが, 他のライブラリでも大きな修正をすることなく実装できると思います.

```
def online_load_and_encoding_dataset(dataset, i, dt, n_time, max_fr=32,
                                     norm=140):
    fr_tmp = max_fr*norm/np.sum(dataset[i][0])
    fr = fr_tmp*np.repeat(np.expand_dims(dataset[i][0],
                                         axis=0), n_time, axis=0)
    input_spikes = np.where(np.random.rand(n_time, 784) < fr*dt, 1, 0)
    input_spikes = input_spikes.astype(np.uint8)
    return input_spikes
```

ここで, 各入力におけるスパイクの数が等しくなるように正規化を行っています. この関数の使用例と, スパイク列への変換が正しく行われているかを確認するコードは次のようになります.

```
import chainer
dt = 1e-3; t_inj = 0.350; nt_inj = round(t_inj/dt)
train, _ = chainer.datasets.get_mnist() # Chainer による MNIST データの読み込み
input_spikes = online_load_and_encoding_dataset(dataset=train, i=0,
                                                dt=dt, n_time=nt_inj)
# 描画
plt.imshow(np.reshape(np.sum(input_spikes, axis=0), (28, 28)), cmap="gray")
plt.show()
```

ここでスパイク列を時間的に加算し, 28×28 の配列に変換した後に描画をしています. 結果は図 5.5 のようになります.



▲ 図 5.5　スパイク列に変換した MNIST データの例 (5).

5.2.4　ネットワークの構築

それではネットワークを実装してみましょう. 長いですが, それぞれの部分はこれまでの実装の組み合わせです. ただし, 著者実装におけるハイパーパラメータでは正常に学習が進まなかったので, 各ハイパーパラメータの値は変更しています[*17].

```
class DiehlAndCook2015Network:
    def __init__(self, n_in=784, n_neurons=100, wexc=2.25, winh=0.85,
                 dt=1e-3, wmin=0.0, wmax=5e-2, lr=(1e-2, 1e-4),
                 update_nt=100):
        self.dt = dt
        self.lr_p, self.lr_m = lr
        self.wmax = wmax
        self.wmin = wmin
        self.n_neurons = n_neurons
        self.n_in = n_in
        self.norm = 0.1
        self.update_nt = update_nt
```

[*17] そもそも論文にハイパーパラメータが書いておらず, 著者実装のコードを読むしかありません.

```
        # Neurons
        self.exc_neurons = DiehlAndCook2015LIF(n_neurons, dt=dt, tref=5e-3,
                                               tc_m=1e-1, vrest=-65,
                                               vreset=-65, init_vthr=-52,
                                               vpeak=20, theta_plus=0.05,
                                               theta_max=35, tc_theta=1e4,
                                               e_exc=0, e_inh=-100)

        self.inh_neurons = ConductanceBasedLIF(n_neurons, dt=dt, tref=2e-3,
                                               tc_m=1e-2, vrest=-60,
                                               vreset=-45,
                                               vthr=-40, vpeak=20,
                                               e_exc=0, e_inh=-85)
        # Synapses
        self.input_synapse = SingleExponentialSynapse(n_in, dt=dt, td=1e-3)
        self.exc_synapse = SingleExponentialSynapse(n_neurons, dt=dt, td=1e-3)
        self.inh_synapse = SingleExponentialSynapse(n_neurons, dt=dt, td=2e-3)

        self.input_synaptictrace = SingleExponentialSynapse(n_in, dt=dt,
                                                            td=2e-2)
        self.exc_synaptictrace = SingleExponentialSynapse(n_neurons, dt=dt,
                                                          td=2e-2)

        # Connections (重みの初期化)
        initW = 1e-3*np.random.rand(n_neurons, n_in)
        self.input_conn = FullConnection(n_in, n_neurons,
                                         initW=initW)
        self.exc2inh_W = wexc*np.eye(n_neurons)
        self.inh2exc_W = (winh/n_neurons)*(np.ones((n_neurons, n_neurons)) \
                         - np.eye(n_neurons))

        self.delay_input = DelayConnection(N=n_neurons, delay=5e-3, dt=dt)
        self.delay_exc2inh = DelayConnection(N=n_neurons, delay=2e-3, dt=dt)
```

```
        self.g_inh = np.zeros(n_neurons)

        self.tcount = 0

        self.s_in_ = np.zeros((self.update_nt, n_in))
        self.s_exc_ = np.zeros((n_neurons, self.update_nt))
        self.x_in_ = np.zeros((self.update_nt, n_in))
        self.x_exc_ = np.zeros((n_neurons, self.update_nt))

    # スパイクトレースのリセット
    def reset_trace(self):
        self.s_in_ = np.zeros((self.update_nt, self.n_in))
        self.s_exc_ = np.zeros((self.n_neurons, self.update_nt))
        self.x_in_ = np.zeros((self.update_nt, self.n_in))
        self.x_exc_ = np.zeros((self.n_neurons, self.update_nt))
        self.tcount = 0

    # 状態の初期化
    def initialize_states(self):
        self.exc_neurons.initialize_states()
        self.inh_neurons.initialize_states()
        self.delay_input.initialize_states()
        self.delay_exc2inh.initialize_states()
        self.input_synapse.initialize_states()
        self.exc_synapse.initialize_states()
        self.inh_synapse.initialize_states()

    def __call__(self, s_in, stdp=True):
        # 入力層
        c_in = self.input_synapse(s_in)
        x_in = self.input_synaptictrace(s_in)
        g_in = self.input_conn(c_in)

        # 興奮性ニューロン層
```

```
        s_exc = self.exc_neurons(self.delay_input(g_in), self.g_inh)
        c_exc = self.exc_synapse(s_exc)
        g_exc = np.dot(self.exc2inh_W, c_exc)
        x_exc = self.exc_synaptictrace(s_exc)

        # 抑制性ニューロン層
        s_inh = self.inh_neurons(self.delay_exc2inh(g_exc), 0)
        c_inh = self.inh_synapse(s_inh)
        self.g_inh = np.dot(self.inh2exc_W, c_inh)

        if stdp:
            # スパイク列とスパイクトレースを記録
            self.s_in_[self.tcount] = s_in
            self.s_exc_[:, self.tcount] = s_exc
            self.x_in_[self.tcount] = x_in
            self.x_exc_[:, self.tcount] = x_exc
            self.tcount += 1

            # Online STDP
            if self.tcount == self.update_nt:
                W = np.copy(self.input_conn.W)
                # post に投射される重みが均一になるようにする
                W_abs_sum = np.expand_dims(np.sum(np.abs(W), axis=1), 1)
                W_abs_sum[W_abs_sum == 0] = 1.0
                W *= self.norm / W_abs_sum
                # STDP 則
                dW = self.lr_p*(self.wmax-W)*np.dot(self.s_exc_, self.x_in_)
                dW -= self.lr_m*W*np.dot(self.x_exc_, self.s_in_)
                clipped_dW = np.clip(dW / self.update_nt, -1e-3, 1e-3)
                self.input_conn.W = np.clip(W + clipped_dW,
                                            self.wmin, self.wmax)
                self.reset_trace() # スパイク列とスパイクトレースをリセット
        return s_exc
```

ここで__call__() 関数は入力スパイク列 s_in と STDP 則による学習をするかどうかの Boolean 変数である stdp の 2 つの値を引数とします．stdp が True のとき，入力層と興奮性ニューロン層のスパイク列，スパイクトレースが記録され，self.tcount が self.update_nt となったときに STDP 則による重みの更新が行われます．重みの更新の前に，興奮性ニューロンへ投射される重みの合計が self.norm となるように正規化をしています．また，ここでの STDP 則は重み依存的なものとしています．なお，重みの更新時に重みが self.wmin と self.wmax 範囲となるように np.clip() で制限をしています．

5.2.5 STDP 則による学習と結果の表示

実際にシミュレーションをする部分を書いていきます．タイムステップは 1 ms とし，350 ms の間は画像を変換したスパイク列を入力，150 ms の間何も入力しない，というようにします．また，興奮性・抑制性ニューロンの数を 100 個とし，訓練データのサンプル数を 10000，エポック数を 30 とします．

なお，学習後にネットワークの評価をするために，このファイル内の関数を呼び出します．そのため，この部分は外部から呼び出されないように if __name__ == '__main__': と記述した中に書いておきます．

```
if __name__ == '__main__':
    # 350ms 画像入力，150ms 入力なしでリセットさせる（膜電位の閾値以外）
    dt = 1e-3 # タイムステップ（sec）
    t_inj = 0.350 # 刺激入力時間（sec）
    t_blank = 0.150 # ブランク時間（sec）
    nt_inj = round(t_inj/dt)
    nt_blank = round(t_blank/dt)

    n_neurons = 100 # 興奮性/抑制性ニューロンの数
    n_labels = 10 # ラベル数
    n_epoch = 30 # エポック数

    n_train = 10000 # 訓練データの数
    update_nt = nt_inj # STDP 則による重みの更新間隔

    # Chainer による MNIST データの読み込み
```

```
    train, _ = chainer.datasets.get_mnist()
    labels = np.array([train[i][1] for i in range(n_train)]) # ラベルの配列

    # ネットワークの定義
    network = DiehlAndCook2015Network(n_in=784, n_neurons=n_neurons,
                                      wexc=2.25, winh=0.85,
                                      dt=dt, wmin=0.0, wmax=2e-5,
                                      lr=(1e-2, 1e-4),
                                      update_nt=update_nt)

    network.initialize_states() # ネットワークの初期化

    spikes = np.zeros((n_train, n_neurons)).astype(np.uint8) # スパイク記録変数
    accuracy_all = np.zeros(n_epoch) # 訓練精度を記録する変数
    blank_input = np.zeros(784) # ブランク入力
    init_max_fr = 32 # 初期のポアソンスパイクの最大発火率

    # 結果を保存するディレクトリ
    results_save_dir = "./LIF_WTA_STDP_MNIST_results/"
    os.makedirs(results_save_dir, exist_ok=True) # ディレクトリが無ければ作成

    # Simulation
    for epoch in range(n_epoch):
        for i in tqdm(range(n_train)):
            max_fr = init_max_fr
            while(True):
                # 入力スパイクをオンラインで生成
                input_spikes = online_load_and_encoding_dataset(train, i, dt,
                                                                nt_inj,
                                                                max_fr)
                spike_list = [] # サンプルごとにスパイクを記録するリスト
                # 画像刺激の入力
                for t in range(nt_inj):
                    s_exc = network(input_spikes[t], stdp=True)
```

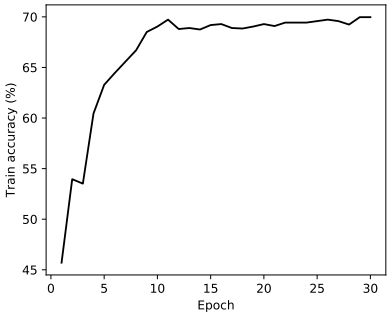
```
                spike_list.append(s_exc)

            # スパイク数を記録
            spikes[i] = np.sum(np.array(spike_list), axis=0)

            # ブランク刺激の入力
            for _ in range(nt_blank):
                _ = network(blank_input, stdp=False)

            # スパイク数を計算
            num_spikes_exc = np.sum(np.array(spike_list))
            # スパイク数が 5 より大きければ次のサンプルへ
            if num_spikes_exc >= 5:
                break
            else: # スパイク数が 5 より小さければ入力発火率を上げて再度刺激
                max_fr += 16

    # ニューロンを各ラベルに割り当てる
    if epoch == 0:
        assignments, proportions, rates = assign_labels(spikes, labels,
                                                        n_labels)
    else:
        assignments, proportions, rates = assign_labels(spikes, labels,
                                                        n_labels, rates)
    print("Assignments:\n", assignments)

    # スパイク数の確認（正常に発火しているか確認）
    sum_nspikes = np.sum(spikes, axis=1)
    mean_nspikes = np.mean(sum_nspikes).astype(np.float16)
    print("Ave. spikes:", mean_nspikes)
    print("Min. spikes:", sum_nspikes.min())
    print("Max. spikes:", sum_nspikes.max())

    # 入力サンプルのラベルを予測する
```

```
        predicted_labels = prediction(spikes, assignments, n_labels)

        # 訓練精度を計算
        accuracy = np.mean(np.where(labels==predicted_labels, 1, 0))
        print("epoch :", epoch, " accuracy :", accuracy)
        accuracy_all[epoch] = accuracy

        # 学習率の減衰
        network.lr_p *= 0.75
        network.lr_m *= 0.75

        # 重みの保存（エポック毎）
        np.save(results_save_dir+"weight_epoch"+str(epoch)+".npy",
                network.input_conn.W)
    # 結果
    plt.figure(figsize=(5,4))
    plt.plot(np.arange(1, n_epoch+1), accuracy_all*100,
             color="k")
    plt.xlabel("Epoch")
    plt.ylabel("Train accuracy (%)")
    plt.savefig(results_save_dir+"accuracy.png")

    # パラメータの保存
    np.save(results_save_dir+"assignments.npy", assignments)
    np.save(results_save_dir+"weight.npy", network.input_conn.W)
    np.save(results_save_dir+"exc_neurons_theta.npy",
            network.exc_neurons.theta)
```

シミュレーション部は入れ子状のループとなっています．`while` によるループは，サンプルを入力した時に興奮性ニューロン全体の発火数が 5 を超えない限り，入力の発火率を増加させて再度スパイク列を入力する，ということを行います．

各エポック終了時には刺激時の興奮性ニューロンの発火情報から，興奮性ニューロンの各ラベルへの割り当てと各サンプルのラベルの予測，および訓練精度の計算を行います．最後に STDP 則における学習率を減衰させます．

学習が終了した後は訓練精度の変化と各ニューロンに割り当てたラベル (assignments), 入力層から興奮性ニューロンへの結合重み, 興奮性ニューロンの閾値を保存しておきます. なお, 学習時における訓練精度の変化は図 5.6 のようになりました.



▲ 図 5.6　訓練時の精度の変化. 最終的な訓練精度は 69.97% となりました.

5.2.6　テストデータによる評価

学習後のネットワークを学習に用いなかったテストデータで評価してみましょう. [*18]. ほぼネットワークの学習の際に記述したコードを流用するだけです. まず, 学習後の重みを読み込んで, テストデータ (1000 サンプル) を入力し, それに対するスパイクを記録します. 次に訓練時に行った各ニューロンのラベルへの割り当てを用いて各サンプルのラベルを予測し, 実際のラベルと比較し, 推論精度を計算します.

```
import chainer
from LIF_WTA_STDP_MNIST import online_load_and_encoding_dataset, prediction
from LIF_WTA_STDP_MNIST import DiehlAndCook2015Network

# 350ms 画像入力, 150ms 入力なしでリセットさせる (膜電位の閾値以外)
dt = 1e-3 # タイムステップ (sec)
t_inj = 0.350 # 刺激入力時間 (sec)
t_blank = 0.150 # ブランク時間 (sec)
```

[*18] コードは./TrainingSNN/LIF_WTA_STDP_MNIST_evaluation.py です.

```
nt_inj = round(t_inj/dt)
nt_blank = round(t_blank/dt)

n_neurons = 100 #興奮性/抑制性ニューロンの数
n_labels = 10 #ラベル数

n_train = 1000 # 訓練データの数
update_nt = nt_inj # STDP 則による重みの更新間隔

_, test = chainer.datasets.get_mnist() # Chainer による MNIST データの読み込み
labels = np.array([test[i][1] for i in range(n_train)]) # ラベルの配列

# 結果が保存されているディレクトリ
results_save_dir = "./LIF_WTA_STDP_MNIST_results/"

# ネットワークの定義
network = DiehlAndCook2015Network(n_in=784, n_neurons=n_neurons,
                                  wexc=2.25, winh=0.85, dt=dt)
network.initialize_states() # ネットワークの初期化

# 重みと閾値の load
network.input_conn.W = np.load(results_save_dir+"weight.npy")
network.exc_neurons.theta = np.load(results_save_dir+"exc_neurons_theta.npy")
network.exc_neurons.theta_plus = 0 # 閾値が上昇しないようにする

#スパイクを記録する変数
spikes = np.zeros((n_train, n_neurons)).astype(np.uint8)
blank_input = np.zeros(784) # ブランク入力
init_max_fr = 32 # 初期のポアソンスパイクの最大発火率

for i in tqdm(range(n_train)):
    max_fr = init_max_fr
    while(True):
        # 入力スパイクをオンラインで生成
```

```
        input_spikes = online_load_and_encoding_dataset(test, i, dt,
                                                        nt_inj, max_fr)
        spike_list = [] # サンプルごとにスパイクを記録するリスト
        # 画像刺激の入力
        for t in range(nt_inj):
            s_exc = network(input_spikes[t], stdp=False)
            spike_list.append(s_exc)

        spikes[i] = np.sum(np.array(spike_list), axis=0) # スパイク数を記録

        # ブランク刺激の入力
        for _ in range(nt_blank):
            _ = network(blank_input, stdp=False)

        num_spikes_exc = np.sum(np.array(spike_list)) # スパイク数を計算
        if num_spikes_exc >= 5: # スパイク数が 5 より大きければ次のサンプルへ
            break
        else: # スパイク数が 5 より小さければ入力発火率を上げて再度刺激
            max_fr += 16

# 入力サンプルのラベルを予測する
assignments = np.load(results_save_dir+"assignments.npy")
predicted_labels = prediction(spikes, assignments, n_labels)

# 訓練精度を計算
accuracy = np.mean(np.where(labels==predicted_labels, 1, 0))
print("Test accuracy :", accuracy)
```

実行した結果, `Test accuracy :  0.635` となり, テストデータ (1000 サンプル) での精度は 63.5% となりました. 100 個のニューロンを学習させた際の論文における精度が約 85% であるので, それに比べるとかなり低い値です. これには学習データを少なくしている, ハイパーパラメータを著者実装とは異なるものにしているということなどが原因であると考えられます.

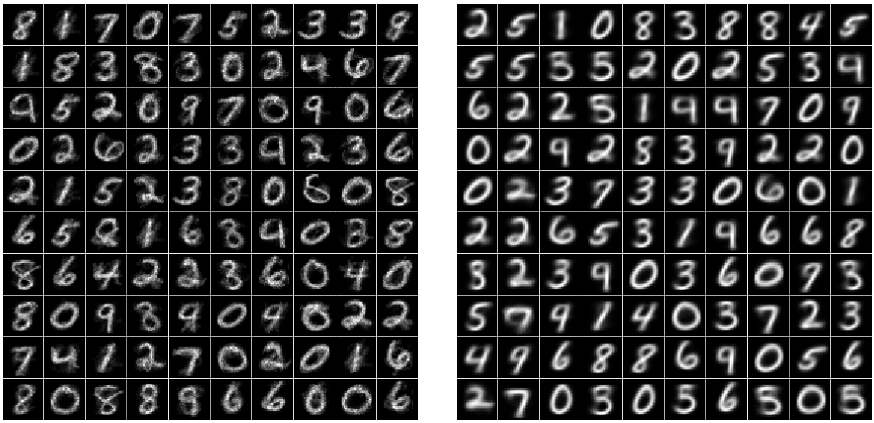
5.2.7 興奮性ニューロンの受容野の描画

最後に, 学習後の入力層から興奮性ニューロン層へのシナプス重みを描画してみましょう[*19]. 各興奮性ニューロンへ投射する 784 個のシナプス重みを 28×28 に reshape して描画するだけです.

```
epoch = 29
n_neurons = 100

# 結果が保存されているディレクトリ
results_save_dir = "./LIF_WTA_STDP_MNIST_results/"
input_conn_W = np.load(results_save_dir+"weight_epoch"+str(epoch)+".npy")
reshaped_W = np.reshape(input_conn_W, (n_neurons, 28, 28))

# 描画
fig = plt.figure(figsize=(6,6))
fig.subplots_adjust(left=0, right=1, bottom=0, top=1,
                    hspace=0.05, wspace=0.05)
row = col = np.sqrt(n_neurons)
for i in tqdm(range(n_neurons)):
  ax = fig.add_subplot(row, col, i+1, xticks=[], yticks=[])
  ax.imshow(reshaped_W[i], cmap="gray")
plt.savefig("weights_"+str(epoch)+".png")
plt.show()
```

結果は図 5.7 のようになります.

[*19] コードは./TrainingSNN/LIF_WTA_STDP_MNIST_visualize_weights.py です.



▲ 図 5.7　学習後の 100 個の興奮性ニューロンの重みを描画したもの．各数字に対応するフィルターが生まれていることが分かります．(左) 1 エポック終了時の重み．受容野が生じつつあるのが分かります．(右) 30 エポック終了時．一番上の段の 10 個のニューロンをラベルに割り当てた結果は “2,5,1,0,8,3,8,8,4,5” となっており，受容野の見た目と一致しています．

第 6 章

Reservoir Computing としての Recurrent SNN の教師あり学習

この章では Reservoir Computing としての Recurrent SNN と, それを学習するための FORCE 法について解説します.

6.1 Reservoir Computing

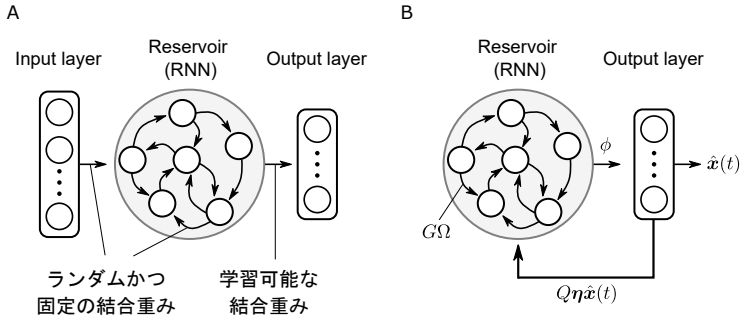
Reservoir Computing は, RNN[*1]のモデルの一種です. 一般の RNN が全ての結合重みを学習するのに対し, Reservoir Computing では RNN のユニット間の結合重みはランダムに初期化して固定し, **出力の結合重みだけを学習**します. そのため, Reservoir Computing は学習するパラメータが少なく, 学習も高速に行えます (もちろん関数の表現力は一般の RNN の方が優れています).

Reservoir というのは溜め池 (貯水池) を意味します. Reservoir Computing では, まず入力信号をランダムな固定重みにより高次空間の信号に変換し, Reservoir RNN(信号の溜め池) に保持します. そして, Reservoir RNN のユニットの活動として保持された情報を学習可能な重みにより線形変換し, 出力とします. このとき, ネットワークの出力が教師信号と一致するように出力重みを更新します.

6.2 FORCE 法と Recurrent SNN への適用

Reservoir Computing における教師あり学習の手法の 1 つとして, **FORCE 法**と呼ばれるものがあります. **FORCE** (First-Order Reduced and Controlled Error) 法は (Sussillo & Abbott, 2009) で提案された学習法で, 元々は発火率ベースの RNN に対す

[*1] ここでは発火率モデルについての RNN について述べています.



▲ 図 6.1 (A)Reservoir Computing の一般的なモデル. 入力と中間にはランダムに固定された重みを用い, 出力のみ学習可能となっています. (B)FORCE 法で用いるモデルの 1 つ. 7.3 節以降でこのモデルの実装を行います.

るオンラインの学習法です (具体的な方法については次節で解説します). さらに (Nicola & Clopath, 2017) は FORCE 法が Recurrent SNN の学習に直接的に使用できる, ということを示しました. この章では Nicola らの手法を用いて Reservoir Computing としての Recurrent SNN の教師あり学習を行います.

6.3 Recurrent SNN に正弦波を学習させる

今回は Recurrent SNN のニューロンの活動をデコードしたものが正弦波となるように (すなわち正弦波を教師信号として) 訓練することを目標とします. 先になりますが, 結果は図のようになります.

6.3.1 ネットワークの構造と教師信号

ネットワークの構造は図のようになっています. ネットワークには特別な入力があるわけではなく, 再帰的な入力によって活動が持続しています (膜電位の初期値をランダムにしているため開始時に発火するニューロン[*2]があり, またバイアス電流もあります).

まず, Reservoir ニューロンの数を N とし, 出力の数を N_{out} とします. i 番目のニューロンの入力はバイアス電流を I_{Bias} として,

$$I_i = s_i + I_{\text{Bias}} \tag{6.1}$$

[*2] ここでの「ニューロン」はこれ以後も含め, Reservoir のユニットを指します.

と表されます. ただし, s_i は

$$s_i = \sum_{j=1}^{N} \omega_{ij} r_j \tag{6.2}$$

として計算されます. r_j が j 番目のニューロンの出力 (シナプスフィルターをかけられたスパイク列), ω_{ij} は j 番目のニューロンから i 番目のニューロンへの結合重みを意味します.

次にニューロンの活動 r_j を重み $\phi \in \mathbb{R}^{N \times N_{\text{out}}}$ で線形にデコードし, その出力 $\hat{\boldsymbol{x}}(t)$ を教師信号 $\boldsymbol{x}(t)$ に近づけます. すなわち,

$$\hat{\boldsymbol{x}}(t) = \sum_{j=1}^{N} \boldsymbol{\phi}_j r_j = \phi^{\intercal} \boldsymbol{r} \tag{6.3}$$

とします. ただし, $^{\intercal}$ を転置記号とし, $\boldsymbol{x}$ を列ベクトル, $\boldsymbol{x}^{\intercal}$ を行ベクトルとします. また, $\boldsymbol{\phi}_j \in \mathbb{R}^{N_{\text{out}}}$ です.

ここから少しややこしいのですが, ネットワークの重み $\Omega = [\omega_{ij}] \in \mathbb{R}^{N \times N}$ は

$$\omega_{ij} = G\omega_{ij}^0 + Q\boldsymbol{\eta}_i^{\intercal} \boldsymbol{\phi}_j \tag{6.4}$$

となっています. ω_{ij}^0 は固定された再帰重みです. G, Q は定数で, $\eta = [\boldsymbol{\eta}_i^{\intercal}] \in \mathbb{R}^{N \times N_{\text{out}}}$ は -1 か 1 に等確率に決められた行列です. よって学習するパラメータは ϕ のみです. よってバイアスを抜いた入力電流 s_i は次のように分割できます.

$$s_i = \sum_{j=1}^{N} \omega_{ij} r_j \tag{6.5}$$

$$= \sum_{j=1}^{N} \left(G\omega_{ij}^0 + Q\boldsymbol{\eta}_i^{\intercal} \boldsymbol{\phi}_j \right) r_j \tag{6.6}$$

$$= Q\boldsymbol{\eta}_i^{\intercal} \hat{\boldsymbol{x}}(t) + \sum_{j=1}^{N} G\omega_{ij}^0 r_j \tag{6.7}$$

6.3.2 固定重みの初期化

固定された結合重み ω_{ij}^0 は $\mathcal{N}(0, (Np)^{-1})$ の正規分布からランダムサンプリングした値です (N はニューロンの数, p は定数). ただし, 各ニューロンが投射される重みの平均が 0 になるようにスケーリングします.

6.3.3 RLS 法による重みの更新

FORCE 法は **RLS フィルタ** (recursive least squares filter, 再帰的最小二乗法フィルタ) という**適応フィルタ** (adaptive filter) の一種を学習するアルゴリズムを, RNN の学習に適応したものです[*3]. FORCE 法は, 誤差を

$$\boldsymbol{e}(t) = \hat{\boldsymbol{x}}(t) - \boldsymbol{x}(t) = \phi(t - \Delta t)^\intercal \boldsymbol{r}(t) - \boldsymbol{x}(t) \tag{6.8}$$

とした場合[*4], 出力重み ϕ を次の式で更新します.

$$\phi(t) = \phi(t - \Delta t) - P(t)\boldsymbol{r}(t)\boldsymbol{e}(t)^\intercal \tag{6.9}$$

$$P(t) = P(t - \Delta t) - \frac{P(t - \Delta t)\boldsymbol{r}(t)\boldsymbol{r}(t)^\intercal P(t - \Delta t)}{1 + \boldsymbol{r}(t)^\intercal P(t - \Delta t)\boldsymbol{r}(t)} \tag{6.10}$$

また, これらの初期値を $\phi(0) = 0, P(0) = I_N \lambda^{-1}$ とします. ただし, I_N は N 次の単位行列で, λ は正則化のための定数です.

6.3.4 FORCE 法の実装

それでは FORCE 法の実装をしてみましょう[*5]. Reservoir ネットワークは 2000 個の LIF ニューロンで構成されているとします. また出力ユニットの個数は 1 です. まず, 各種定数と教師信号を定義します.

```
N = 2000  # ニューロンの数
dt = 5e-5 # タイムステップ（s）
tref = 2e-3 # 不応期（s）
tc_m = 1e-2 #  膜時定数（s）
vreset = -65 # リセット電位（mV）
vrest = 0 # 静止膜電位（mV）
vthr = -40 # 閾値電位（mV）
vpeak = 30 # ピーク電位（mV）
BIAS = -40 # 入力電流のバイアス（pA）
```

[*3] なお, (Sussillo & Abbott, 2009) では Delta 則を用いることで, RLS 法を用いない重みの更新則も紹介されています.

[*4] 実際にはこれは真の誤差ではなく, **事前誤差** (apriori error) と呼ばれるものです. 真の誤差は $\phi(t)^\intercal \boldsymbol{r}(t) - \boldsymbol{x}(t)$ と表されます.

[*5] コードは `./TrainingSNN/LIF_FORCE_sinewave.py` です. ModelDB において公開されている MATLAB のコード (`https://senselab.med.yale.edu/ModelDB/ShowModel.cshtml?model=190565`) を参考にしました.

```
td = 2e-2; tr = 2e-3 # シナプスの時定数 (s)
alpha = dt*0.1
P = np.eye(N)*alpha
Q = 10; G = 0.04

T = 15 # シミュレーション時間 (s)
tmin = round(5/dt) # 重み更新の開始ステップ
tcrit = round(10/dt) # 重み更新の終了ステップ
step = 50 # 重み更新のステップ間隔
nt = round(T/dt) # シミュレーションステップ数
zx = np.sin(2*np.pi*np.arange(nt)*dt*5) # 教師信号
```

次にニューロンとシナプスを定義します.

```
from Models.Neurons import CurrentBasedLIF
from Models.Synapses import DoubleExponentialSynapse

# ニューロンとシナプスの定義
neurons = CurrentBasedLIF(N=N, dt=dt, tref=tref, tc_m=tc_m,
                          vrest=vrest, vreset=vreset, vthr=vthr, vpeak=vpeak)
neurons.v = vreset + np.random.rand(N)*(vpeak-vreset) # 膜電位の初期化

synapses_out = DoubleExponentialSynapse(N, dt=dt, td=td, tr=tr)
synapses_rec = DoubleExponentialSynapse(N, dt=dt, td=td, tr=tr)

# 再帰重みの初期値
p = 0.1 # ネットワークのスパース性
OMEGA = G*(np.random.randn(N,N))*(np.random.rand(N,N)<p)/(np.sqrt(N)*p)
for i in range(N):
    QS = np.where(np.abs(OMEGA[i,:])>0)[0]
    OMEGA[i,QS] = OMEGA[i,QS] - np.sum(OMEGA[i,QS], axis=0)/len(QS)
```

シナプスのインスタンスとして synapses_out, synapses_rec があります. 実は synapses_out だけでも良いのですが, 高速化のために 2 つ用意しています. また,

OMEGA はランダムに生成した後に, シナプス後細胞に投射される重みの平均が 0 となるようにスケーリングをしています.

次に各種変数の初期化と, 記録用変数を定義します.

```
# 変数の初期値
k = 1 # 出力ニューロンの数
E = (2*np.random.rand(N, k) - 1)*Q
PSC = np.zeros(N).astype(np.float32) # シナプス後電流
JD = np.zeros(N).astype(np.float32) # 再帰入力の重み和
z = np.zeros(k).astype(np.float32) # 出力の初期化
Phi = np.zeros(N).astype(np.float32) #  学習される重みの初期値

# 記録用変数
REC_v = np.zeros((nt,10)).astype(np.float32) # 膜電位の記録変数
current = np.zeros(nt).astype(np.float32) # 出力の電流の記録変数
tspike = np.zeros((4*nt,2)).astype(np.float32) # スパイク時刻の記録変数
ns = 0 # スパイク数の記録変数
```

それではシミュレーションのメインの部分を書いていきましょう.

```
for t in tqdm(range(nt)):
    I = PSC + np.dot(E, z) + BIAS # シナプス電流
    s = neurons(I) # 中間ニューロンのスパイク

    index = np.where(s)[0] # 発火したニューロンの index
    len_idx = len(index) # 発火したニューロンの数
    if len_idx > 0:
        JD = np.sum(OMEGA[:, index], axis=1)
        tspike[ns:ns+len_idx,:] = np.vstack((index, 0*index+dt*t)).T
        ns = ns + len_idx # スパイク数の記録

    PSC = synapses_rec(JD*(len_idx>0)) # 再帰的入力電流
    #PSC = np.dot(OMEGA, r) # 遅い
    r = synapses_out(s) # 出力電流（神経伝達物質の放出量）
```

```
    r = np.expand_dims(r,1) # (N,) -> (N, 1)
    z = np.dot(Phi.T, r) # デコードされた出力
    err = z - zx[t] # 誤差

    # FORCE 法 (RLS) による重み更新
    if t % step == 1:
        if t > tmin:
            if t < tcrit:
                cd = np.dot(P, r)
                Phi = Phi - np.dot(cd, err.T)
                P = P - np.dot(cd, cd.T) / (1.0 + np.dot(r.T, cd)

    current[t] = z # デコード結果の記録
    REC_v[t] = neurons.v_[:10] # 膜電位の記録
```

途中で少し不思議に思われるようなことをしています.
`PSC = synapses_rec(JD*(len_idx>0))` の部分 (とその少し上) ですが, これはデコードに用いる `r` を行列変換するよりも発火した結合重みの和を取り, 再帰入力のシナプス後細胞のモデルに入力した方が速いという理由によります. `t` が一定のステップの範囲にある場合は FORCE 法により学習を実行します. 最後に各種変数を記録しています.

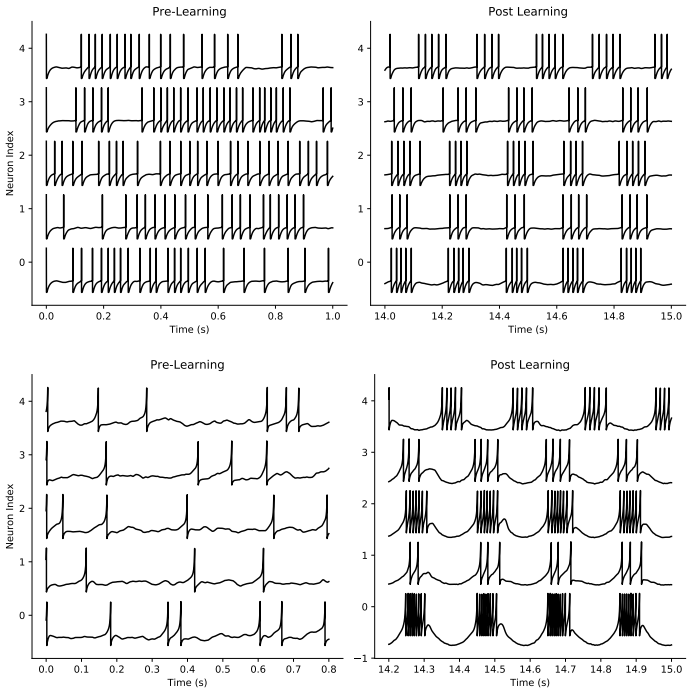
それでは学習後の結果を表示しましょう. 初めに発火数と発火率を表示し, 次に学習前と学習後の 5 つのニューロンの膜電位, 最後に学習前/中間と学習後のデコード結果を描画します (なお, この本に記載はしていないですがコードには重みの固有値の描画も付けています).

```
TotNumSpikes = ns
M = tspike[tspike[:,1]>dt*tcrit,:]
AverageRate = len(M)/(N*(T-dt*tcrit))
print("\n")
print("Total number of spikes : ", TotNumSpikes)
print("Average firing rate(Hz): ", AverageRate)
step_range = 20000
plt.figure(figsize=(10, 5))
plt.subplot(1,2,1)
```

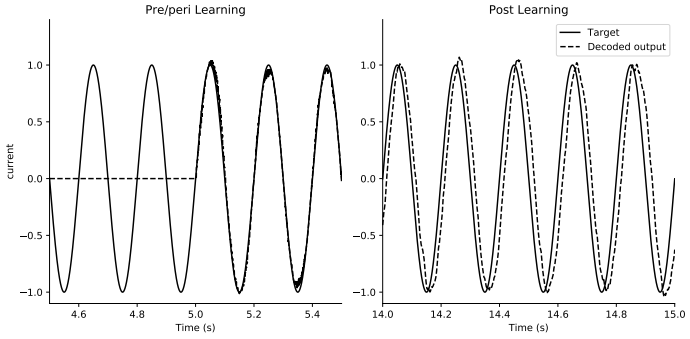
```
for j in range(5):
    plt.plot(np.arange(step_range)*dt,
             REC_v[:step_range, j]/(50-vreset)+j, color="k")
plt.title('Pre-Learning')
plt.xlabel('Time (s)'); plt.ylabel('Neuron Index')
plt.subplot(1,2,2)
for j in range(5):
    plt.plot(np.arange(nt-step_range, nt)*dt,
             REC_v[nt-step_range:, j]/(50-vreset)+j,
             color="k")
plt.title('Post Learning'); plt.xlabel('Time (s)')
plt.show()

plt.figure(figsize=(10,5))
plt.subplot(1,2,1)
plt.plot(np.arange(nt)*dt, zx, label="Target", color="k")
plt.plot(np.arange(nt)*dt, current, label="Decoded output",
         linestyle="dashed", color="k")
plt.xlim(4.5,5.5); plt.ylim(-1.1,1.4)
plt.title('Pre/peri Learning')
plt.xlabel('Time (s)'); plt.ylabel('current')
plt.subplot(1,2,2)
plt.title('Post Learning')
plt.plot(np.arange(nt)*dt, zx, label="Target", color="k")
plt.plot(np.arange(nt)*dt, current, label="Decoded output",
         linestyle="dashed", color="k")
plt.xlim(14,15); plt.ylim(-1.1,1.4)
plt.xlabel('Time (s)'); plt.legend(loc='upper right')
plt.show()
```

結果は図 6.2, 6.3 のようになります．また, 同様のシミュレーションを Izhikevich ニューロンで行った[*6]結果も示しています．

[*6] コードは./TrainingSNN/Izhikevich_FORCE_sinewave.py です．



▲ 図 6.2　FORCE 法による学習前 (左) と学習後 (右) の発火率の変化.　(上)LIF ニューロン, (下) を Izhikevich ニューロン



▲ 図 6.3　FORCE 法による学習前 (左) と学習後 (右) のデコード結果の変化 (LIF ニューロンの場合). 教師信号は実線, デコード結果は破線で示している.

6.3.5 鳥の鳴き声の再現と海馬の記憶と再生

Nicola らは教師信号として正弦波以外にも Van der Pol 方程式や Lorenz 方程式の軌道を用いて実験しています. さらに教師信号としてベートーヴェンの歓喜の歌 (Ode to joy) や鳥の鳴き声を用いても学習可能であったようです.

話は少しずれますが, 小鳥の運動前野である **HVC** には連鎖的に結合したニューロン群が存在します. これはリズムを生み出すための計時に関わっているといわれています[*7]. カナリアの HVC ニューロンを実験的に損傷 (ablation) させると歌が歌えなくなるという実験がありますが, 同様に SNN の HVC パターンを ablation すると学習した歌が再生できなくなったようです. このような計時に関わるパターンを **HDTS**(high-dimentional temporal signal) と Nicola らは呼んでいます. HDTS を学習させた後に歓喜の歌を学習させると, HDTS がない場合よりも短い時間かつ高精度で学習できたようです.

さらに HDTS を外部入力とし, 同時に映像を学習させる, という実験もしています (HDTS を内的に学習させる場合も行っています). ネットワークは記録した映像を実時間で再生することができましたが, 外部信号の HDTS を加速させることで圧縮再生が可能だったそうです. さらに HDTS を逆にすると, 逆再生もできたそうです.

ニューロンの発火のタスク依存的な圧縮は実験的に観察されています (例えば Euston, et al., 2007). 空間的な課題 (箱の中に入れて探索させるなど) をラットにさせると, 課題中に記憶された場所細胞の順序だった活動は, ラットの睡眠中に圧縮再生されるという実験結果があります. その圧縮比は 5.4〜8.1 だったそうですが, この比率は SNN が映像を大きな損失なく再生できる圧縮比とほぼ同じであったようです. Nicola らはさらに進んで SNN を用いて海馬における急速圧縮学習[*8]の機構における介在細胞の働きについての研究も行っています (Nicola & Clopath, 2019).

6.4 RLS 法の導出

ここからは RLS 法の導出を行います (cf. Haykin, 2002). RLS 法では次の損失関数 $C \in \mathbb{R}^{N_{\text{out}}}$ を最小化するような重み $\phi = [\boldsymbol{\phi}_j] \in \mathbb{R}^{N \times N_{\text{out}}}$ を求めます. シミュレーション

[*7] 時間を認知するための感覚器官は存在しないため, 様々な時間感覚は脳によって生み出されています. HVC ニューロンの話を含む, 脳における時間認知の神経機構については, 一般向けの本ではありますが,
Dean Buonomano 著, 村上郁也 翻訳. (2018). “脳と時間”. 森北出版.
を読まれることをお勧めします.

[*8] 海馬では情報がリップル波 (sharp wave-ripples, SPW-R) 中に, 情報を圧縮した形式で, かつ順方向・逆方向で再生されます (Buzsáki, 2015). SPW-R は皮質から海馬への記憶の移行に関わっていると言われています.

時間を T とすると, C は

$$C = \int_0^T (\hat{\boldsymbol{x}}(t) - \boldsymbol{x}(t))^2 \mathrm{d}t + \lambda \phi^\intercal \phi \tag{6.11}$$

です. ただし, $\hat{\boldsymbol{x}}(t), \boldsymbol{x}(t) \in \mathbb{R}^{N_{\text{out}}}$ です.

さて, 式の C を最小化するような ϕ を数値的に求めるためには, 損失関数の近似が必要です. まず, タイムステップ Δt で C を離散化します. さらに n ステップ目における重み $\phi(n)$ により, $\hat{\boldsymbol{x}}(i) \simeq \phi(n)^\intercal \boldsymbol{r}(i)$ と近似します. このとき, n ステップ目の損失関数 $C(n)$ は

$$C(n) \simeq \sum_{i=0}^{n} (\hat{\boldsymbol{x}}(i) - \boldsymbol{x}(i))^2 + \lambda \phi(n)^\intercal \phi(n) \tag{6.12}$$

$$\simeq \sum_{i=0}^{n} (\phi(n)^\intercal \boldsymbol{r}(i) - \boldsymbol{x}(i))^2 + \lambda \phi(n)^\intercal \phi(n) \tag{6.13}$$

となります. ここで L2 正則化 (ridge) 付きの (通常の) 最小二乗法の**正規方程式** (normal equation) により, $C(n)$ を最小化する $\phi(n)$ は

$$\phi(n) = \left[\sum_{i=0}^{n} (\boldsymbol{r}(i)\boldsymbol{r}(i)^\intercal + \lambda I_N) \right]^{-1} \left[\sum_{i=0}^{n} \boldsymbol{r}(i)\boldsymbol{x}(i)^\intercal \right] \tag{6.14}$$

$$= P(n)\psi(n) \tag{6.15}$$

となります[*9]. ただし,

$$P(n)^{-1} = \sum_{i=0}^{n} (\boldsymbol{r}(i)\boldsymbol{r}(i)^\intercal + \lambda I_N) \quad \left(= \int_0^T \boldsymbol{r}(t)\boldsymbol{r}(t)^\intercal \mathrm{d}t + \lambda I_N \right) \tag{6.16}$$

$$\psi(n) = \sum_{i=0}^{n} \boldsymbol{r}(i)\boldsymbol{x}(i)^\intercal \tag{6.17}$$

です. $P(n)$ は $\boldsymbol{r}(n)$ の相関行列の時間積分と係数倍した単位行列の和の逆行列となっています. また,

$$P(n)^{-1} = P(n-1)^{-1} + \boldsymbol{r}(n)\boldsymbol{r}(n)^\intercal \tag{6.18}$$

となります. ここで, **逆行列の補助定理** (Matrix Inversion Lemma, または Sherman-Morrison-Woodbury Identity) より,

$$X = A + BCD \tag{6.19}$$

$$\Rightarrow X^{-1} = A^{-1} - A^{-1}B(C^{-1} + DA^{-1}B)^{-1}DA^{-1} \tag{6.20}$$

[*9] 重み ϕ で C を微分し, 勾配が 0 となるときの方程式の解です.

となるので, $X = P(n)^{-1}, A = P(n-1)^{-1}, B = \boldsymbol{r}(n), C = I_N, D = \boldsymbol{r}(n)^\intercal$ とすると,

$$P(n) = P(n-1) - \frac{P(n-1)\boldsymbol{r}(n)\boldsymbol{r}(n)^\intercal P(n-1)}{1 + \boldsymbol{r}(n)^\intercal P(n-1)\boldsymbol{r}(n)} \tag{6.21}$$

が成り立ちます (右辺 2 項目の分母はスカラーとなります). さらに

$$\begin{aligned}
\psi(n) &= \psi(n-1) + \boldsymbol{r}(n)\boldsymbol{x}(n)^\intercal && (6.22)\\
&= P(n-1)^{-1}\phi(n-1) + \boldsymbol{r}(n)\boldsymbol{x}(n)^\intercal && (6.23)\\
&= \left\{P(n)^{-1} - \boldsymbol{r}(n)\boldsymbol{r}(n)^\intercal\right\}\phi(n-1) + \boldsymbol{r}(n)\boldsymbol{x}(n)^\intercal && (6.24)
\end{aligned}$$

となります. 式 (6.22) から式 (6.23) へは

$$\phi(n) = P(n)\psi(n) \Rightarrow \psi(n) = P(n)^{-1}\phi(n) \tag{6.25}$$

であること, 式 (6.23) から式 (6.24) へは式 (6.18) により,

$$P(n-1)^{-1} = P(n)^{-1} - \boldsymbol{r}(n)\boldsymbol{r}(n)^\intercal \tag{6.26}$$

であることを用いています. よって,

$$\begin{aligned}
\phi(n) &= P(n)\psi(n)\\
&= P(n)\left[\left\{P(n)^{-1} - \boldsymbol{r}(n)\boldsymbol{r}(n)^\intercal\right\}\phi(n-1) + \boldsymbol{r}(n)\boldsymbol{x}(n)^\intercal\right]\\
&= \phi(n-1) - P(n)\boldsymbol{r}(n)\boldsymbol{r}(n)^\intercal\phi(n-1) + P(n)\boldsymbol{r}(n)\boldsymbol{x}(n)^\intercal\\
&= \phi(n-1) - P(n)\boldsymbol{r}(n)\left[\boldsymbol{r}(n)^\intercal\phi(n-1) - \boldsymbol{x}(n)^\intercal\right]\\
&= \phi(n-1) - P(n)\boldsymbol{r}(n)\boldsymbol{e}(n)^\intercal && (6.27)
\end{aligned}$$

となります. 式 (6.22) と式 (6.27) を連続時間での表記法にすると, 式 (6. 9,10) の更新式となります.

参考文献

本書を書く上で参考にした文献です．本文中で引用を明記していない文献も含みます．

生理学全般

小澤司，本間研一，大森治紀，他, 編. (2014). “標準生理学 (第 8 版)”. 医学書院.

Hall, J.E. (2015). “*Guyton and Hall Textbook of Medical Physiology* (13th ed.)”. Saunders. 〔John E. Hall 著, 石川義弘, 岡村康司, 尾仲達史, 河野憲二, 他，翻訳．(2018). “ガイトン生理学 (原著第 13 版)”. エルゼビア・ジャパン株式会社.〕

神経生理学

宮川博義, 井上雅司, (2013). “ニューロンの生物物理 (第 2 版)”. 丸善出版.

Kandel, E.R., Schwartz, J.H., Jessell. T.M., Siegelbaum, S.A., and Hudspeth, A.J. (2013). “*Principles of Neural Science* (5th ed.)”. New York:McGraw-Hill Medical. 〔Eric R. Kandel 他著, 金澤一郎, 宮下保司 監修. (2014). “カンデル神経科学”. メディカルサイエンスインターナショナル.〕

計算論的神経科学

甘利俊一, 深井朋樹, 他, 偏. (2009). “脳の計算論”. 東京大学出版会.

国里愛彦, 片平健太郎, 沖村宰, 山下祐一. (2019). “計算論的精神医学: 情報処理過程から読み解く精神障害”. 勁草書房.

田中宏和. (2019). “計算論的神経科学: 脳の運動制御・感覚処理機構の理論的理解へ”. 森北出版.

Dayan, P., and Abbott, L. (2001). “*Theoretical Neuroscience*”. The MIT Press.

Gerstner, W., and Kistler, W.M. (2002). “*Spiking Neuron Models*”. Cambridge University Press. (`https://icwww.epfl.ch/~gerstner/BUCH.html`)

Gerstner, W., Kistler, W.M., Naud, R., and Paninski, L. (2014). “*Neuronal Dynamics:*

From Single Neurons to Networks and Models of Cognition". Cambridge University Press. (`https://neuronaldynamics.epfl.ch/`).

Izhikevich, E. (2007). "*Dynamical Systems in Neuroscience: The Geometry of Excitability and Bursting*". The MIT Press.

Maass, O., and Bishop, C. (2001). "*Pulsed Neural Networks*". The MIT Press.

Deep spiking neural networks に関する総論

酒見悠介, 森野佳生. (2019). "スパイキングニューラルネットワークにおける深層学習". 生産研究. **71**(2), 159-167.

Illing, B., Gerstner, W., and Brea, J. (2019). "Biologically plausible deep learning – but how far can we go with shallow networks?". *Neural Networks*. **118**, 90-101.

Pfeiffer, M., and Pfeil, T. (2018). "Deep learning with spiking neurons:opportunities and challenges". *Front. Neurosci.* **12**, 774.

Tavanaei, A. (2019). "Deep Learning in Spiking Neural Networks". *Neural Networks*. **111**, 47–63.

1 章

島崎秀昭. "スパイク統計モデル入門". (`http://www.neuralengine.org/res/book/book.html`).

Bernardo, M., Budd, C., Champneys, A.R., and Kowalczyk, P. (2008). "*Piecewise-smooth Dynamical Systems*". Springer.

Bhumbra, G. (2018). "Deep learning improved by biological activation functions". *arXiv* [Preprint]. arXiv:1804.11237.

Chen, R., Rubanova, Y., Bettencourt, J., and Duvenaud, D. (2018). "Neural Ordinary Differential Equations". In *Advances in Neural Information Processing Systems*.

Deger, M., Helias, M., Boucsein, C., and Rotter, S. (2012). "Statistical properties of superimposed stationary spike trains". *Journal of computational neuroscience*, **32**(3), 443-463.

He, K., Zhang, X., Ren, S., and Sun, J. (2016). "Deep Residual Learning for Image Recognition". In *CVPR*.

Izhikevich, E. (2004). "Which Model to Use for Cortical Spiking Neurons?". *IEEE*. **15**(5), 1063-70.

Kar, K., Kubilius, J., Schmidt. K., Issa. E.B., and DiCarlo, J.J. (2019). "Evidence that recurrent circuits are critical to the ventral stream' s execution of core object

recognition behavior". *Nat. Neurosci.* **22**(6), 974-983.

Pachitariu, M., Brody, C.D., Jun, P.D.J.K., and Holmes, P.J. "Probabilistic models for spike trains of single neurons". (`http://www.gatsby.ucl.ac.uk/~marius/papers/SpikTrainStats.pdf`).

Rosenblatt, F. (1958). "The Perceptron: A Probabilistic Model for Information Storage and Organization in the Brain" . *Psychological Review.* **65**(6), 386-408.

Shinomoto, S., Kim, H., Shimokawa, T., Matsuno, N., Funahashi, S., Shima, K., Fujita, I., Tamura, H., Doi, T., Kawano, K., Inaba, N., Fukushima, K., Kurkin, S., Kurata, K., Taira, M., Tsutsui, K., Komatsu, H., Ogawa, T., Koida, K., Tanji, J., and Toyama K. (2009). "Relating neuronal firing patterns to functional differentiation of cerebral cortex". *PLoS Comput. Biol.* **5**(7), e1000433.

Shinomoto, S., Shima, K., and Tanji, J. (2003). "Differences in spiking patterns among cortical neurons". *Neural Computation.* **15**, 2823-2842.

Liao, Q., and Poggio, T. (2016). "Bridging the Gaps Between Residual Learning, Recurrent Neural Networks and Visual Cortex". *arXiv* [Preprint]. arXiv:1604.03640.

Maimon, G., and Assad, J. (2009). "Beyond poisson: Increased spike-time regularity across primate parietal cortex", *Neuron.* **62**, 426-440.

McCulloch, W., and Pitts, W. (1943). "A logical calculus of the ideas immanent in nervous activity" . *The bulletin of mathematical biophysics (Kluwer Academic Publishers).* **5**(4), 115-133.

Wilson, H., and Cowan, J. (1972). "Excitatory and inhibitory interactions in localized populations of model neurons". *Biophys. J.* **12**, 1–24.

2 章

Cavallari, S., Panzeri, S., and Mazzoni, A. (2014). "Comparison of the dynamics of neural interactions between current-based and conductance-based integrate-and-fire recurrent networks". *Front. Neural Circuits.* **8**, 12.

Destexhe, A., Mainen, Z., and Sejnowski, T. (1994). "Synthesis of models for excitable membranes, synaptic transmission and neuromodulation using a common kinetic formalism". *J. Computational Neurosci.* **1**, 195-230.

Rall, W. (1967). "Distinguishing theoretical synaptic potentials computed for different some-dendritic distributions of synaptic inputs".*J. Neurophysiol.* **30**, 1138-1168.

3 章

森江隆. (2019). “ニューロモルフィックシステムと物理デバイス”. 応用物理. **88**(7), 481-485.

Benjamin, B.V., Gao, P., McQuinn, E., Choudhary, S., Chandrasekaran, A. R., Bussat, J.-M., Alvarez-Icaza, R., Arthur, J.V., Merolla, P. A., and Boahen, K. (2014). “Neurogrid: A mixed-analog-digital multichip system for large-scale neural simulations”. *Proceedings of the IEEE.* **102**(5). 699–716.

Davies, M., Srinivasa, N., Lin, T., Chinya, G., Cao, Y., Choday, S.H., Dimou, G., Joshi,P., Imam, N., Jain, S., Liao, Y., Lin, C., Lines, A., Liu, R., Mathaikutty, D., McCoy, S., Paul, A., Tse, J., Venkataramanan, G., Weng, Y., Wild, A., Yang, Y., and Wang, H. (2018). “Loihi: A neuromorphic manycore processor with on-chip learning”. *IEEE Micro.* **38**(1). 82–99.

Esser, S., Appuswamy, R., Merolla, P., Arthur, J., and Modha, D. (2015). “Backpropagation for energy-efficient neuromorphic computing”, In *Advances in Neural Information Processing Systems.*

Furber, S. B., Galluppi, F., Temple, S., and Plana, L.A. (2014).“The SpiNNaker project”. *IEEE.* **102**(5), 652–665.

Hu Y., Tang H., Wang Y., and Pan G. (2018). “Spiking deep residual network. *arXiv* [Preprint]. arXiv:1805.01352.

Hubara, I., Courbariaux, M., Soudry, D., El-Yaniv, R., and Bengio, Y. (2018). “Binarized neural networks”. In *Advances in Neural Information Processing Systems.*

Kim, M., and Smaragdis, P. (2016). “Bitwise neural networks”. *arXiv* [Preprint]. arXiv:1601.06071.

Liu, Q., Chen, Y., and Furber, S. (2017). “Noisy Softplus: an activation function that enables SNNs to be trained as ANNs”. *arXiv* [Preprint]. arXiv:1706.03609.

Merolla, P.A., Arthur, J.V., Alvarez-Icaza, R., Cassidy, A.S., Sawada, J. , Akopyan, F. , Jackson, B.L., Imam, N., Guo, C., Nakamura, Y., Brezzo, B., Vo, I., Esser, S.K., Appuswamy, R., Taba, B., Amir, A., Flickner, M.D., Risk, W.P., Manohar, R., and Modha, D.S. (2014). “A million spiking-neuron integrated circuit with a scalable communication network and interface”. *Science.* **345**(6197), 668–673.

Merolla, P.A., Arthur, J.V., Alvarez-Icaza, R., Cassidy, A.S., Sawada, J. , Akopyan, F. , Jackson, B.L., Imam, N., Guo, C., Nakamura, Y., Brezzo, B., Vo, I., Esser, S.K., Appuswamy, R., Taba, B., Amir, A., Flickner, M.D., Risk, W.P., Manohar, R., and Modha, D.S. (2014). “A million spiking-neuron integrated circuit with a scalable communication network and interface”. *Science.* **345**(6197), 668–673.

Moradi, S., Qiao, N., Stefanini, F., and Indiveri, G. (2018). “A scalable multicore architecture with heterogeneous memory structures for dynamic neuromorphic asynchronous processors (DYNAPs)”. *IEEE Transactions on Biomedical Circuits and Systems*. **12**(1). 106–122.

Rastegari, M., Ordonez, V., Redmon, J., and Farhadi A. (2016). “XNOR-Net: ImageNet classification using binary convolutional neural networks”. In *ECCV*.

Severa, W., Vineyard,C.M., Dellana, R., Verzi, S.J., and Aimone, J.B. (2019). “Training deep neural networks for binary communication with the Whetstone method”. *Nature Machine Intelligence*. **1**, 86–94.

Sengupta, A., Ye, Y., Wang, R., Liu, C., and Roy, K.. (2019). “Going Deeper in Spiking Neural Networks: VGG and Residual Architectures”. *Front. Neurosci.* **13**, 95.

4 章

Akrout, M., Wilson, C., Humphreys, P., Lillicrap, T., and Tweed, D. (2019). “Deep Learning without Weight Transport”. *arXiv* [Preprint]. arXiv:1904.05391.

Bohte, S.M., Poutré, H., and Kok, J.N. (2000). “Error-Backpropagation in Temporally Encoded Networks of Spiking Neurons”. *Neurocomputing*. **48**, 17-37.

Comsa, I.M., Krzysztof, P., Versari, L., Fischbacher, T., Gesmundo, A., and Alakuijala, J. (2019). “Temporal coding in spiking neural networks with alpha synaptic function”. *arXiv* [Preprint]. arXiv:1907.13223.

Dauwels, J., Vialatte, F., Weber, T., and Cichocki, A. (2008). “On similarity measures for spike trains”. In *Advances in Neuro-Information Processing*, 177–185. Springer.

Huh, D., and Sejnowski, T.J. (2018). “Gradient Descent for Spiking Neural Networks”. In *Advances in Neural Information Processing Systems*.

Lansdell, B.J., Prakash, P., and Kording, K.P. (2019). “Learning to solve the credit assignment problem”. *arXiv* [Preprint]. arXiv:1906.00889.

Lee, J.H., Delbruck, T., and Pfeiffer, M. (2016). “Training Deep Spiking Neural Networks Using Backpropagation”. *Front. Neurosci.* **10**, 508.

Lillicrap, T., Cownden, D., Tweed, D., and Akerman, C. (2016). “Random synaptic feedback weights support error backpropagation for deep learning”. *Nat. Commun.* **7**, 13276.

Nøkland, A. (2016). “Direct Feedback Alignment Provides Learning in Deep Neural Networks”. *arXiv* [Preprint]. arXiv:1609.01596.

Ponulak, F., and Kasiński, A. (2010). “Supervised learning in spiking neural networks

with ReSuMe: sequence learning, classification, and spike shifting". *Neural Comput.* **22**(2), 467-510.

Tavanaei, A., and Maida, A. (2019). "BP-STDP: Approximating Backpropagation using Spike Timing Dependent Plasticity". *Neurocomputing.* **330**, 39-47.

Thiele, J.C., Bichler, O., and Dupret, A. (2019). "SpikeGrad: An ANN-equivalent Computation Model for Implementing Backpropagation with Spikes". *arXiv* [Preprint]. arXiv:1906.00851.

Woźniak, S., Pantazi, A., Bohnstingl, T., and Eleftheriou, E.(2018). "Deep learning incorporating biologically-inspired neural dynamics". *arXiv* [Preprint]. arXiv:1812.07040.

van Rossum, M.C.W. (2001). "A Novel Spike Distance". *Neural Computation.* **13**(4), 751–763.

Wu, Y., Deng, L., Li, G., Zhu, J., and Shi, L. (2018). "Spatio-Temporal Backpropagation for Training High Performance Spiking Neural Networks". *Front. Neurosci.* **12**, 331.

Zenke, F., and Ganguli, S. (2018) "SuperSpike: Supervised Learning in Multilayer Spiking Neural Networks". *Neural Comput.* **30**, 1514–1541.

5 章

Bell, C., Han, V., Sugawara, Y., and Grant, K. (1997). "Synaptic plasticity in a cerebellum-like structure depends on temporal order". *Nature.* **387**, 278-281.

Bi, G.Q., and Poo, M.M. (1998). "Synaptic Modifications in Cultured Hippocampal Neurons: Dependence on Spike Timing, Synaptic Strength, and Postsynaptic Cell Type". *Journal of Neuroscience.* **18**, 10464-10472.

Diehl, P., and Cook, M. (2015). "Unsupervised learning of digit recognition using spike-timing dependent plasticity". *Front. Comput. Neurosci.* **9**, 99.

Feldman, D.E. (2012). "The Spike-Timing Dependence of Plasticity". *Neuron.* **75**(4), 556-71.

Fino, E., Glowinski, J., and Venance, L. (2005). "Bidirectional activity-dependent plasticity at corticostriatal synapses". *J. Neurosci.* **25**, 11279-11287.

Morrison, A., Diesmann, M., and Gerstner, W. (2008). "Phenomenological models of synaptic plasticity based on spike timing". *Biol. Cybern.* **98**(6), 459–478.

Song, S., Miller, K., and Abbott, L. (2000). "Competitive Hebbian learning through spike-timing-dependent synaptic plasticity". *Nat Neurosci.* **3**(9), 919-26.

6 章

Buzsáki, G. (2015). "Hippocampal sharp wave-ripple: a cognitive biomarker for episodic memory and planning". *Hippocampus*. 25. 1073–1188.

Euston, D., Tatsuno, M., and McNaughton, B. (2007). "Fast-forward playback of recent memory sequences in prefrontal cortex during sleep". *Science*. **318**, 1147–1150.

Haykin, S. (2002). "*Adaptive Filtering Theory* (5th ed.)". Pearson.

Nicola, W., and Clopath, C. (2017). "Supervised learning in spiking neural networks with FORCE training". *Nat. Commun.* **8**(1), 2208

Nicola, W., and Clopath, C. (2019). "A diversity of interneurons and Hebbian plasticity facilitate rapid compressible learning in the hippocampus". *Nat. Neurosci.* **22**, 1168–1181.

Sussillo, D., and Abbott, L. (2009). "Generating coherent patterns of activity from chaotic neural networks". *Neuron.* **63**, 544–557.

あとがき

ここまでお読みくださり, ありがとうございます. 本書の作成にはかなり時間を費やしましたが, それでも残念ながら書けなかったことが多数あります (かなり様々なことが適当になってしまいました). また, 書いている内容に数学的・生理学的ミスがある可能性は十分にあります. この本は自分の勉強のために書いたので, もしミスを見つけてくださった方はお手数ですが, 私の twitter(`@tak_yamm`) に連絡していただければ幸いです. また, twitter の方々には色々と情報提供をしてくださったことを感謝いたします. 最後に, 指導していただいている先生方に, 論文を書かずこのような本を書いてしまったことをお詫びいたします.

2019 年 9 月 22 日　山拓

時間の都合上書けなかったリスト

- コンパートメントモデル
- 神経生理学の入門 (本来つける予定でした)
- FitzHugh-Nagumo モデルなどと nullcline による分岐解析
- タイプ 1, 2 ニューロンについて
- Time-rescaling theorem
- 発火系列からの発火率の推定
- 電気シナプスのモデル
- 増強シナプスと減衰シナプス
- Dopamine-Modulated STDP (DA-STDP) と強化学習
- Triplet STDP
- Convolutional SNN
- Rate coding と Temporal coding の比較と学習法

など. これ以外にも書けていないことは大量に存在します.

ゼロから作る Spiking Neural Networks

発行日	2019 年 9 月 22 日	(初版)
	2019 年 9 月 28 日	(第 2 版)
	2019 年 10 月 21 日	(第 2.05 版)
発　行	AIMS/阪医 Python 会	
著　者	山拓	
Twitter ID	@tak_yamm	
印刷所	株式会社 日光企画	