

# 2021 年度 修士論文

深層学習を用いた ILC 崩壊点検出アルゴリズムの開発

九州大学大学院 理学府 物理学専攻  
粒子物理学分野 素粒子実験研究室

後藤 輝一

指導教員 末原 大幹, 川越 清以

2021 年 1 月 28 日



九州大学  
KYUSHU UNIVERSITY



---

## 概要

本研究では人工知能技術の一つである深層学習を用いて、国際リニアコライダー (ILC) で行う実験に使用する為の崩壊点検出アルゴリズムの開発を行なった。崩壊点検出アルゴリズムはジェット再構成の最初段であり、荷電粒子の飛跡から重いフレーバーのクォークの崩壊点を探索するアルゴリズムである。ILC のジェット再構成は崩壊点検出アルゴリズムの後、ジェットクラスタリング、フレーバータギングと続く。現在、ILC ではジェット再構成に LCFIPlus というソフトウェアが使用されている。LCFIPlus は iLCSof t というソフトウェアエコシステムの一つであり、これらのジェット再構成アルゴリズムを統合している。LCFIPlus ではフレーバータギングにのみ、機械学習技術の一つである Boosted Decision Trees (BDTs) が使用されており、その他のアルゴリズムは人の手によって定められた閾値によるデータの剪定を用いた手法が使用されている。

近年、粒子物理学の分野では物理解析やシミュレーションに深層学習を使用し、精度や性能を改善する取り組みが行われている。深層学習とは教師あり学習の一つであり、分類や回帰問題を解く機械学習技術の一つである。事象再構成についても、この深層学習を用いた性能の改善が期待されており、本研究はその様な取り組みの一つである。深層学習は取り扱うデータや問題の性質によって様々な発展的手法が提案されており、本研究では特に言語や音声といった系列データを取り扱う回帰型ニューラルネットワーク（リカレントニューラルネットワーク）と同じく系列データを取り扱う為の Attention 機構を使用した。リカレントニューラルネットワークでは、既存のネットワーク構造をそのまま使うのではなく、直近の系列情報に依存しづらい様な独自のネットワーク構造を構築した。深層学習の訓練データとして ILC の検出器コンセプトの一つである International Large Detector (ILD) の検出器フルシミュレーションデータを使用したが、本研究のアルゴリズムの基本的な発想は検出器の違いによらず使用できる。

本論文では、深層学習を用いた崩壊点検出アルゴリズムの開発とアルゴリズムの LCFIPlus への導入をまとめた。LCFIPlus 内の機能は Marlin というアプリケーションフレームワークのプロセッサーとして運用されている。ここでは本アルゴリズムを LCFIPlus フレームワーク内に実装し、既存の LCFIPlus のアルゴリズムと置換可能にした。また、本アルゴリズムについての性能は LCFIPlus のアルゴリズムと比較する事によって評価した。評価指標として飛跡段階での崩壊点の再構成効率を用いた。LCFIPlus の崩壊点検出アルゴリズムと比較して誤認識率の高い結果となつたが、一方で再構成効率の良い結果を得ることができた。



# 目次

<b>第 1 章</b>	<b>序論</b>	<b>10</b>
1.1	標準模型	10
1.2	ILC 計画	12
1.3	ILC の物理	12
1.4	ILC の検出器 -International Large Detector (ILD)-	14
1.5	ILC のソフトウェアと事象再構成	15
1.5.1	ソフトウェア	16
1.5.2	飛跡の再構成	17
1.5.3	ジェットの再構成	18
1.6	本研究の目的	20
1.7	本論文の流れ	20
<b>第 2 章</b>	<b>深層学習</b>	<b>22</b>
2.1	機械学習と深層学習	22
2.2	パーセプトロン	23
2.2.1	単純パーセプトロン	24
2.2.2	多層パーセプトロン	25
2.3	ニューラルネットワーク	26
2.3.1	ニューラルネットワークの構造	27
2.3.2	ニューラルネットワークの学習	29
2.3.3	ディープニューラルネットワーク	32
2.4	リカレントニューラルネットワーク	33
2.4.1	リカレントニューラルネットワークの構造と学習	33
2.4.2	リカレントニューラルネットワークの問題点	35
2.4.3	長・短期記憶 (Long Short-Term Memory, LSTM)	36
2.5	Attention	41
2.5.1	エンコーダー・デコーダーモデル	41
2.5.2	Attention	42

---

2.6	ハイパーパラメータ	45
<b>第3章</b>	<b>崩壊点検出の為のネットワーク</b>	<b>46</b>
3.1	事象	46
3.1.1	事象全体の性質	46
3.1.2	飛跡の情報と前処理	50
3.2	深層学習を用いた崩壊点検出の実現	52
3.3	飛跡対についてのネットワーク	53
3.3.1	ネットワークの構造	55
3.3.2	ネットワークの学習と戦略	56
3.3.3	ネットワークの評価	59
3.4	任意の数の飛跡についてのネットワーク	68
3.4.1	ネットワークの構造	68
3.4.2	ネットワークの学習と戦略	76
3.4.3	ネットワークの評価	77
<b>第4章</b>	<b>深層学習を用いた崩壊点検出</b>	<b>84</b>
4.1	崩壊点検出アルゴリズム	84
4.2	崩壊点検出の最適化と評価手法	87
4.2.1	SVのタネの選別	87
4.2.2	崩壊点の生成	89
4.3	崩壊点検出の性能	90
<b>第5章</b>	<b>現行の手法 (LCFIPlus)との比較</b>	<b>93</b>
5.1	崩壊点検出単体での比較	93
5.2	より詳細な比較	94
5.2.1	アルゴリズムの修正	95
5.2.2	本研究の崩壊点検出を用いたジェット再構成の性能	96
<b>第6章</b>	<b>まとめと今後の展望</b>	<b>99</b>
<b>付録A</b>	<b>ソースコード</b>	<b>102</b>
A.1	任意の数の飛跡についてのネットワーク	102
A.1.1	独自のリカレントニューラルネットワーク	102
A.2	崩壊点の再構成	123
<b>参考文献</b>		<b>141</b>

# 図目次

1.1	標準模型の素粒子	11
1.2	ILC の全体像	12
1.3	ILC 計画の今後	13
1.4	重心系エネルギーとヒッグス事象生成断面積の関係	14
1.5	International Large Detector (ILD)	15
1.6	primary vertex と secondary vertex の図示	18
1.7	深層学習によるジェットの再構成	21
2.1	機械学習の中の深層学習の位置付け	24
2.2	単純パーセプトロン	24
2.3	ヘヴィサイドの階段関数	25
2.4	多層パーセプトロン	26
2.5	活性化関数	27
2.6	ニューラルネットワーク	28
2.7	リカレントニューラルネットワーク	34
2.8	リカレントニューラルネットワークの重みの明示的な表現	34
2.9	リカレントニューラルネットワークの出力方法	35
2.10	LSTM の流れ	36
2.11	単体の LSTM	37
2.12	LSTM の各ゲートについての図解	39
2.13	Stacked LSTM	39
2.14	双方向 LSTM	40
2.15	LSTM によるエンコーダー・デコーダーモデル	42
2.16	Attention と LSTM によるエンコーダー・デコーダーモデル	43
2.17	Key, Value, Query の図示	43
2.18	Additive Attention と Dot-Product Attention	44
3.1	終状態 $b\bar{b}$ での崩壊点の例	48
3.2	事象に含まれる飛跡の本数と崩壊点の個数	48

3.3	LCFIPlus によって予想される崩壊点の位置の分布 . . . . .	51
3.4	LCFIPlus によって予想される崩壊点の位置と $\chi^2$ 値の相関 . . . . .	51
3.5	終状態 $b\bar{b}$ での崩壊点 . . . . .	54
3.6	飛跡対についてのネットワークの概略図 . . . . .	55
3.7	各終状態での分類クラスのデータ数の比 . . . . .	56
3.8	訓練データでの分類クラスのデータ数の比 . . . . .	57
3.9	評価のための飛跡対についてのネットワーク . . . . .	60
3.10	ネットワークのスコアとクラス分類の効率の関係 . . . . .	62
3.10	ネットワークのスコアとクラス分類の効率の関係 . . . . .	63
3.11	各モデルの ROC 曲線 . . . . .	64
3.11	各モデルの ROC 曲線 . . . . .	65
3.12	各モデルの混合行列と各モデルの相対値 . . . . .	66
3.13	t-SNE による次元削減の比較 . . . . .	68
3.14	リカレントニューラルネットワークを用いた崩壊点生成 . . . . .	69
3.15	崩壊点生成のためのリカレントニューラルネットワーク構造 . . . . .	69
3.16	系列 1 ステップについての独自リカレントニューラルネットワーク構造 . . . . .	70
3.17	独自リカレントニューラルネットワーク構造の解釈 . . . . .	71
3.18	Attention を組み込んだエンコーダー・デコーダーモデルへの拡張 . . . . .	72
3.19	独自リカレントニューラルネットワークの Attention への拡張 . . . . .	73
3.20	本研究における Additive Attention の図解 . . . . .	75
3.21	飛跡順のシャッフル . . . . .	77
3.22	標準的な LSTM と独自のネットワークの比較 . . . . .	79
3.23	各データ属性の効率とスコアの関係 . . . . .	80
3.24	各データ属性の ROC 曲線 . . . . .	81
3.25	Attention Weight . . . . .	83
4.1	崩壊点検出アルゴリズム . . . . .	85
4.2	閾値と SV のタネの効率, 純度の関係 . . . . .	88
4.3	閾値と SV のタネの効率, 純度についての PR 曲線 . . . . .	88
4.4	同一の崩壊チェインと同一の親粒子 . . . . .	90
4.5	閾値と崩壊点検出の性能の関係 . . . . .	91
5.1	(差し替え) フレーバータギングの性能に関する ROC 曲線 . . . . .	97
5.2	LCFIPlus と同等の選別を行った後の性能 . . . . .	98

# 表目次

1.1	ILD サブディテクターの詳細なパラメータ (バレル) . . . . .	15
1.2	ILD サブディテクターの詳細なパラメータ (エンドキャップ) . . . . .	16
3.1	MC シミュレーションデータの性質 . . . . .	47
3.2	データサンプルの事象数と用途 . . . . .	49
3.3	ソフトウェア・ハードウェアの環境 . . . . .	53
3.4	飛跡対についての崩壊点の種類 . . . . .	54
3.5	飛跡対についてのネットワークにおける訓練可能なパラメータ数 . . . . .	58
3.6	評価のための飛跡対についてのモデル . . . . .	59
3.7	任意の数の飛跡についてのネットワークの入力変数の大きさ . . . . .	76
3.8	任意の数の飛跡についてのネットワークにおける訓練可能なパラメータ数 . . . . .	78
4.1	データサンプル $b\bar{b} - 08$ を用いた性能 . . . . .	92
4.2	データサンプル $c\bar{c} - 08$ を用いた性能 . . . . .	92
5.1	LCFIPlus での性能値 . . . . .	94
5.2	崩壊点検出のソフトウェア動作環境 . . . . .	95
5.3	(差し替え) 再構成された崩壊点と疑似崩壊点の個数 . . . . .	96
5.4	フレーバータギングのカテゴリー . . . . .	97

# 第 1 章

## 序論

本章では、まず 1.1 節で素粒子の振る舞いを記述する理論である、標準模型 (the Standard Model, SM) について紹介する。

次に 1.2 節にて、この標準模型を超える物理 (Physics beyond the Standard Model, BSM) を探索するための国際リニアコライダー (International Linear Collider, ILC) 計画についての説明を行う。ILC が目指す物理について 1.3 節で述べ、ILC で使用する予定の検出器やソフトウェアに関する説明を 1.4 節と 1.5 節で行う。

更に本研究の目的について 1.6 節で、本論文の流れについて 1.7 節でそれぞれ述べ本論文の序論とする。

本章の作成にあたり、参考文献 [1, 2] を使用した。

### 1.1 標準模型

宇宙の誕生や、生物の発生と同様に、物質の起源は人類の根元的な問いの一つである。そのような物質を構成する最小の粒子のことを素粒子といい、その素粒子の振る舞いを記述する理論の一つが標準模型である。標準模型によると、素粒子はスピン半整数のフェルミ粒子とスピン整数のボース粒子に分類される。

標準模型ではフェルミ粒子は全てスピン  $1/2$  の粒子で構成され、陽子や中性子などのハドロンを構成するクォークと電子やニュートリノなどのレプトンに分けられる。更に、クォークは電荷が  $+2/3$  のアップクォーク系列と  $-1/3$  のダウントクォーク系列に、レプトンは電荷が  $-1$  の荷電レプトンと中性電荷の中性レプトン (ニュートリノ) に分けられる。また、それぞれ世代と呼ばれるものを構成し現在は 3 つの世代が確認されている。クォークの場合はアップクォーク  $u$ 、チャームクォーク  $c$ 、トップクォーク  $t$ 、ダウントクォーク  $d$ 、ストレンジクォーク  $s$ 、ボトムクォーク  $b$  が存在する。レプトンの場合は荷電レプトンとして、電子  $e^-$ 、ミュー粒子  $\mu^-$ 、タウ粒子  $\tau^-$ 、中性レプトンとして、電子ニュートリノ  $\nu_e$ 、ミューニュートリノ  $\nu_\mu$ 、タウニュートリノ  $\nu_\tau$  が存在している。これらの系列や世代間の違いをフレーバーと呼んでいる。

ボース粒子は、基本的な四つの相互作用である強い相互作用、弱い相互作用、電磁相互作用、重力相互作用の内、重力相互作用を除いた三つの相互作用をそれぞれ媒介するスピン 1 のゲージ粒子と、対称性を破り素粒子に質量を与えるスピン 0、中性電荷のヒッグス粒子  $H$  で構成される。電磁相互作用を媒介する粒子として、中性電荷の光子  $\gamma$ 、強い相互作用を媒介する粒子として、中性電荷のグルーオン  $g$ 、弱い相互作用を媒介する粒子として、電荷  $\pm 1$  の  $W$  ボソン ( $W^\pm$ )、中性電荷の  $Z$  ボソン ( $Z$ ) が存在する。

更に素粒子には、質量やスピンが等しく電荷の正負のみが反転した反粒子が存在する。<sup>\*1</sup> これら反粒子と通常の粒子が衝突すると対消滅が起こり、その質量が全てエネルギーへと変換される。一方、ある粒子の質量の二倍以上のエネルギーを生じさせた場合は対生成が起こり、その粒子と反粒子の対が生成されることがある。

以上の素粒子を標準模型の素粒子といい、一般に図 1.1 のように纏められている。

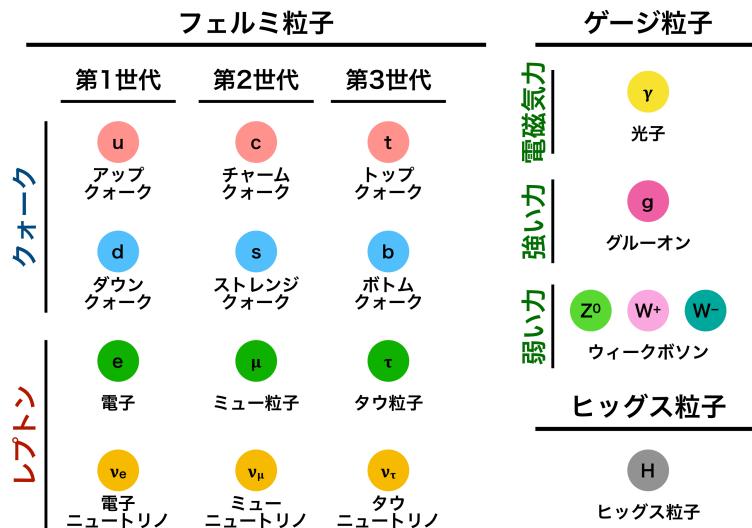


図 1.1: 標準模型の素粒子

標準模型は様々な実験で非常によく確かめられているが、ダークマターをはじめとする幾つかの物理現象を説明できていない。これらの標準模型で説明できない物理現象を Physics beyond the Standard Model (BSM) といい、現在は様々な実験施設で BSM の探索が行われている。次節の ILC 計画はそのような試みの一つである。

<sup>\*1</sup> 狹義には Dirac 方程式の負のエネルギー解の解釈である。

## 1.2 ILC 計画

ILC 計画とは、日本の東北にある北上山地に全長 20.5 km の国際リニアコライダー (ILC) を建設する計画である (図 1.2)。ILC 計画は国際共同研究であり、2013 年に出版された The Technical Design Report (TDR) [3] には 2400 人の研究者、48ヶ国、392 の大学と研究機関のグループが署名している。

また技術開発は国際将来加速器委員会 (The International Committee for Future Accelerator, ICFA) の下、国際推進チーム (International Development Team, IDT) によって実行されており、この IDT を 2022 年を目処に ILC 準備研究所 (ILC Pre-Lab) に移行する計画となっている。ILC 計画の今後の大まかな流れを図 1.3 に示す。

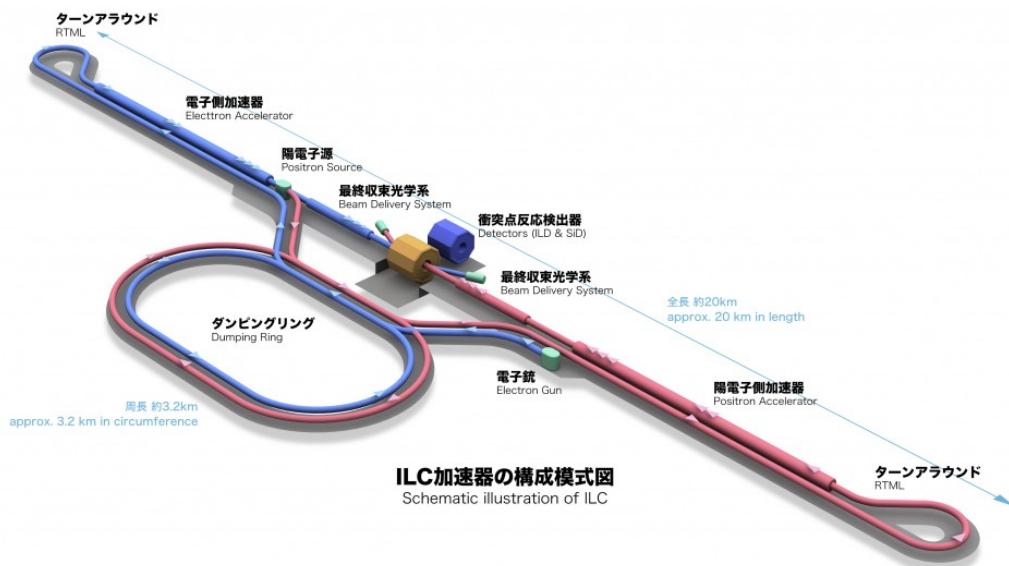


図 1.2: ILC の全体像 [4]

## 1.3 ILC の物理

ヒッグス粒子が 2012 年に欧州原子核研究機構 (CERN) の大型ハドロン衝突型加速器 (Large Hadron Collider, LHC) で発見されて以降、ヒッグス粒子の性質について、より詳細な調査が行われている。ヒッグス機構は標準模型の中で電弱相互作用の対称性を破り素粒子に質量を与える役割を担っており、またヒッグス粒子は質量に比例した結合定数を持つという特徴を持っている。このような性質からヒッグス粒子の振る舞いは標準模型によって詳細に決定され、BSM によって標準模型の素粒子の振る舞いに変化が生じた場合、ヒッグス粒子はその影響を受けると予想されている。特にヒッグス粒子の結合定数は、そのような BSM の模型やパラメータの違いによって異なった変化をするため、ヒッグス粒子の結合定数を調べる事によって BSM の探索やその模型識別が可能である。

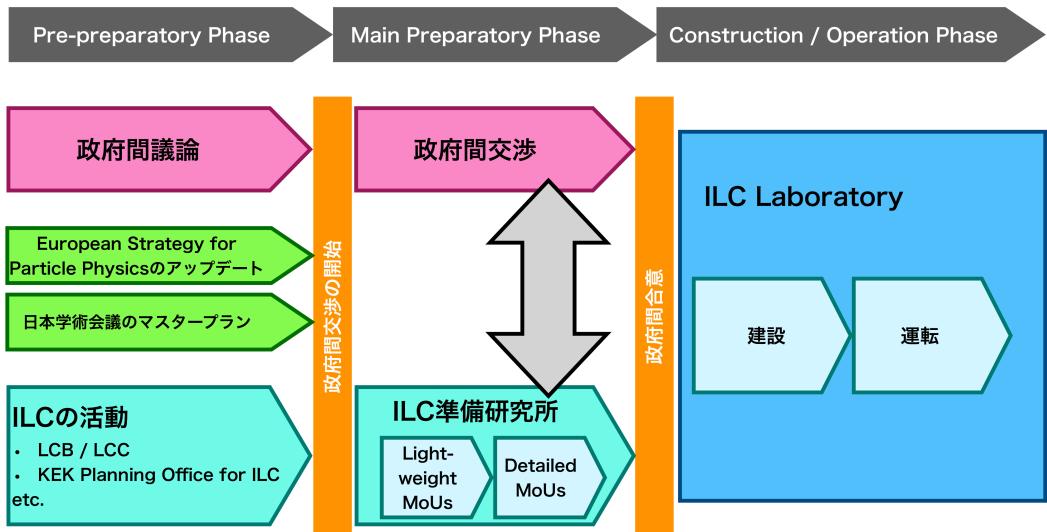


図 1.3: ILC 計画の今後 [5]

ILC は、このヒッグス粒子の性質を詳細に調べ、新物理を発見する為のヒッグスファクタリーとしての役割が期待されている。LHC が陽子-陽子を衝突させる加速器であるのに対し、ILC は電子-陽電子を衝突させる加速器である。電子と陽電子は粒子と反粒子の関係となっており、ILC は LHC と比較して目的とする事象に対しエネルギーをより効率的に使うことができる。更に電子-陽電子衝突の場合は相互作用の関係から、背景事象が少ないという特徴を持っている。

ILC は電子・陽電子衝突によって、Z 粒子とヒッグス粒子を生成する  $e^+e^- \rightarrow Zh$  事象の反応断面積が最大となる重心系エネルギー  $\sqrt{s} = 250$  GeV での運転開始 (ILC250) を予定している (図 1.4)。また、ILC には様々な物理目標を達成する為に多数のアップグレードオプションが存在し、重心系エネルギーについては主線形加速器を延長、加速勾配を上昇することで 1 TeV 以上への拡張が可能である。

$e^+e^- \rightarrow Zh$  事象はヒッグス粒子の反跳粒子である Z 粒子を再構成することによって、ヒッグス粒子の崩壊モードに寄らずヒッグス粒子を識別・測定できるという点で非常に重要である。また、背景事象である  $e^+e^- \rightarrow Z\gamma$  事象や  $e^+e^- \rightarrow ZZ$  事象に関してもよく理解されており、電弱相互作用の計算によって不定性を 0.1 % 程度に抑えることができる [1]。この反跳粒子を使用した解析では  $e^+e^- \rightarrow Zh$  事象の全断面積を得ることができ、絶対正規化されたヒッグス粒子の結合定数やヒッグス粒子の暗黒物質への崩壊についての測定が可能となる。

ILC ではヒッグス粒子の崩壊分岐比について、 $BR(h \rightarrow b\bar{b}, c\bar{c}, gg, WW^*, ZZ^*)$  の精密測定が期待されている。特に  $BR(h \rightarrow b\bar{b}, c\bar{c})$  の精密測定は、第 3, 2 世代のクォークとヒッグス粒子についての湯川結合を理解する為の手がかりとして非常に重要である。

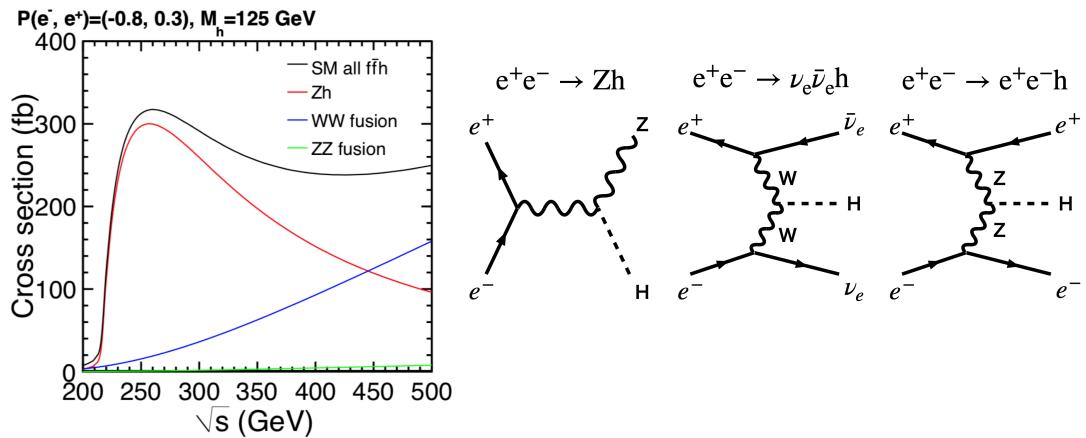


図 1.4: 重心系エネルギーとヒッグス事象生成断面積の関係 [6]。ヒッグス粒子の質量を 125 GeV とした時の  $e^+e^- \rightarrow Zh$  事象の生成断面積を赤線でプロットしている。また、WW fusion  $e^+e^- \rightarrow \nu\bar{\nu}h$  事象、ZZ fusion  $e^+e^- \rightarrow e^+e^-h$  事象をそれぞれ青線、緑線で表現している。

これらの重いクォーク対は、それぞれ真空中でクォークの粒子反粒子対を生成・結合しハドロンとなる。更に、この過程で生成されたクォークも同様にハドロンを形成するため、初めのクォーク対のそれぞれの進行方向には多数のハドロン粒子が生成されることとなる。これをジェットといい  $BR(h \rightarrow b\bar{b}, c\bar{c}, g\bar{g})$  の精密測定では、このジェットの元となったクォークのフレーバーを高精度に同定する必要がある。このジェットの再構成については、1.5.3 項にて説明する。

## 1.4 ILC の検出器 -International Large Detector (ILD)-

ILC では Silicon Detector (SiD) と International Large Detector (ILD) の二つの検出器コンセプトが検討されている。本研究は ILD の検出器シミュレーションデータを用いている為、本節では ILD について簡単な解説を行う。ただし、本研究の基本的な発想はそのような検出器の違いに寄らず使用することができる。

ILD はヒッグス粒子や電弱相互作用の物理からの要求値を満たすように設計され、また後述する particle flow algorithm (PFA) (1.5.2 項) によって最適化されている。また ILD は様々なサブディテクターによって構成され、ビームの衝突点 (図 1.5 の右下) を包む様に内側から順に、Vertex Detector (VTX), Silicon Internal Tracker (SIT), Time Projection Chamber (TPC), 電磁カロリメータ (ECAL), ハドロンカロリメータ (HCAL), Iron Yoke (Muon) が並んでいる。また、HCAL と Iron Yoke の間には超伝導ソレノイドがあり、3.5T の磁場をかけている。ILD では、VTX や SIT, TPC を用いて荷電粒子の飛跡を測定し、ECAL によって電子や光子、HCAL によってハドロン粒子のエネルギーを測定する。

衝突点の前方、後方方向には、Forward Tracking Detector (FTD), Luminosity Calorimeter (LumiCAL), LHCAL, Beam Calorimeter (BeamCAL) が並んでいる。これらのサブディテクターは輝度の測定やビーム状態の監視などを行っている。それぞれのサブディテクターの詳細については表 1.1 と表 1.2 にまとめる。

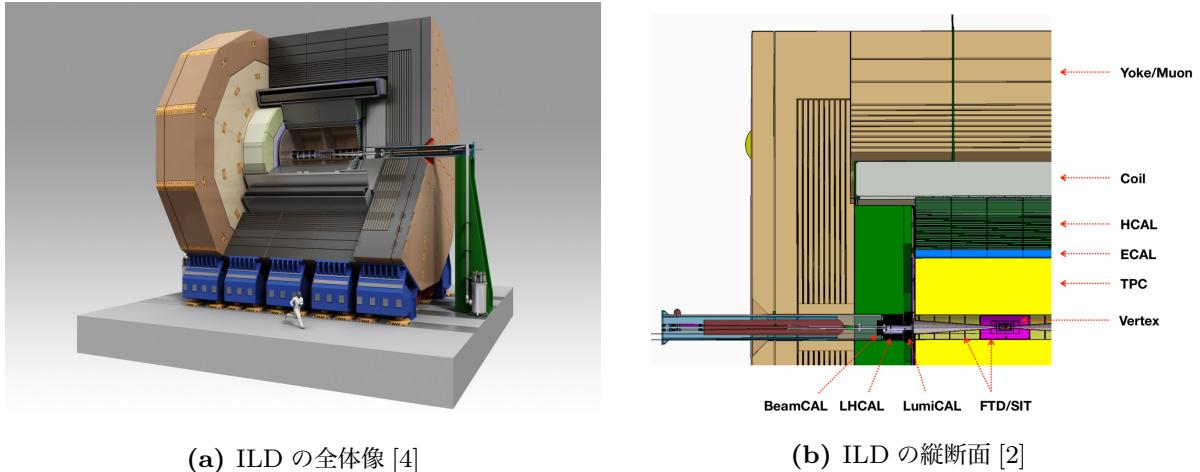


図 1.5: International Large Detector (ILD)

表 1.1: ILD サブディテクターの詳細なパラメータ (バレル) [2]

	$r_{in}$ [mm]	$r_{out}$ [mm]	$z_{max}$ [mm]	要素技術
VTX	16	60	125	シリコンピクセルセンサー
SIT	153	303	644	シリコンピクセルセンサー
TPC	329	1770	2350	マイクロパターンガス検出器
SET	1773	1776	2300	シリコンストリップセンサー
ECAL	1805	2028	2350	吸収層：タングステン センサー：シリコン/シンチレーター
HCAL	2058	3345	2350	吸収層：スチール センサー：シンチレーター/RPC ガス
Coil	3425	4175	3872	
Muon	4450	7755	4047	センサー：シンチレーター

## 1.5 ILC のソフトウェアと事象再構成

ここでは ILC で使用されるソフトウェアと事象再構成について述べる。事象再構成とは、加速器実験によって得られるデータから飛跡やジェットなどの物理情報を再構成するアルゴリズムである。そのような再構成は電子-陽電子の衝突毎に行われ、この衝突一回分の事を事象という。

ILC のソフトウェアは iLCSoft[7] と呼ばれるソフトウェアエコシステムにまとめられている。

表 1.2: ILD サブディテクターの詳細なパラメータ (エンドキャップ) [2]

	$z_{min}$ [mm]	$z_{max}$ [mm]	$r_{in}$ [mm]	$r_{out}$ [mm]	要素技術
FTD	220	371		153	シリコンピクセルセンサー
	645	2212		300	シリコンストリップセンサー
ECAL	2411	2635	250	2096	吸収層：タングステン センサー：シリコン/シンチレーター
HCAL	2650	3937	350	3226	吸収層：スチール センサー：シンチレーター/RPC ガス
Muon	4072	6712	350	7716	センサー：シンチレーター
BeamCAL	3115	3315	18	140	吸収層：タングステン GaAs 読み出し
LumiCAL	2412	2541	84	194	吸収層：タングステン センサー：シリコン
LHCAL	2680	3160	130	315	吸収層：タングステン

ILC における事象再構成は、飛跡再構成や PFA といった粒子の再構成 (1.5.2 項) と、更にそれらによって再構成された粒子を用いたジェットの再構成 (1.5.3 項) に区分できる。ジェットの再構成は崩壊点検出、ジェットクラスタリング、フレーバータギングという行程で行われ、ジェット再構成アルゴリズムとして iLCSoft 内の LCFIPlus[8, 9] というソフトウェアが使用されている。

### 1.5.1 ソフトウェア

ILC は実際の実験データをまだ取得できないため、本研究で使用するデータは全てシミュレーションデータである。シミュレーションデータは標準模型と BSM の模型を用いて、モンテカルロ (Monte Carlo, MC) 法によって生成されている。それらのシミュレーションデータは LCIO と呼ばれる階層型の Event Data Model (EDM) によって管理されている。LCIO では、MC 情報から事象の生データ、デジタル化、解析や後述する再構成に至るまでが紐づけられており、階層的に取り扱うことができる。

ILC のソフトウェアは二つの検出器コンセプト (ILD, SiD) で概ね共通しており、前述したように現在は iLCSoft というソフトウェアエコシステムによって統括されている。提供されている API の言語は C++・java・Fortran である。

それらソフトウェアモジュールは Marlin[10] という C++ アプリケーションフレームワークによって運用されており、プロセッサーと呼ばれるモジュールを作成・組み込むことにより、様々な再構成・解析アルゴリズムを簡単に別のモジュールへ置き換えることができる。また、データの入出力は LCIO, ROOT フォーマットによって行われる。

### 1.5.2 飛跡の再構成

飛跡の再構成は、シミュレーションされた検出器のデータから、粒子（飛跡）を再構成する飛跡再構成と、そのようにして得られた個々の飛跡検出器（VTX, SIT, TPC など）の粒子についての情報を繋ぎ合わせ、より高精度の粒子情報を提供する PFA という手順によって行われる。

#### 飛跡再構成

飛跡再構成では、まず荷電粒子の飛跡をパターン認識を用いて再構成し、次にそれらの飛跡について運動学的な物理量をフィッティングによって抽出している。LCIOにおいて飛跡は飛跡の曲率  $\Omega$ , インパクトパラメータ  $d_0$ ,  $z_0$ , 方向パラメータ  $\phi$ ,  $\tan \lambda$  の perigee track parameterization によって保持されている。ILDにおいて異なるサブディテクターの飛跡再構成は異なるアルゴリズムが使用されている。

#### Particle Flow Algorithm (PFA)

飛跡再構成によって運動学的な物理量を得られるのは荷電粒子のみである。中性粒子は VXD や TPC に飛跡を残さない為、カロリメータ (ECAL, HCAL) を用いた再構成が必要となる。このように粒子の性質によって、再構成を行うべき最適なサブディテクターは異なっている。ILC では PFA を用いた高い粒子識別を目指している。PFAにおいて荷電粒子は飛跡検出器によって測定され、光子や中性ハドロンはそれぞれ ECAL や HCAL によって再構成されている。PFA ではジェット中の個々の粒子を分離する事によって、ジェットエネルギー分解能の大幅な向上が期待されており、その実現には高精細なカロリメータと高精度な飛跡検出器が必要となる。

iLCSoft では PandoraPFA というアルゴリズムが使用されている。このアルゴリズムでは、まずカロリメータのヒットをクラスター化し、それらクラスターとトラッキング情報を関連づけ荷電粒子と中性粒子の分離を行っている。出力は PandoraPFO と呼ばれる再構成された粒子のオブジェクトであり、これは以降の解析や更なる再構成に使用可能である。

以上が飛跡の再構成である。飛跡の再構成では、検出器で得られた情報から粒子を再構成するまでを行なっている。実際には注目すべき物理事象は前述したジェットのような特徴的なシグネチャを残す為、次項のジェットの再構成による更なる解析が必要である。

### 1.5.3 ジェットの再構成

事象中に生じたクォークは1.3節で述べたようにジェットを形成する。ジェットには多数の粒子(飛跡)が含まれ、それら多数の粒子の親となる粒子が崩壊した点を崩壊点(Vertex)という。特に、ハドロンのような準安定な親粒子の崩壊点の事をsecondary vertexといい、電子-陽電子の衝突点をprimary vertexという(図1.6)。ジェットの再構成では、まずこの崩壊点を崩壊点検出アルゴリズムを用いて探索し、得られた崩壊点を用いてジェット中の粒子を分離するジェットクラスタリングが行われる。最後に、ジェット中の飛跡や崩壊点から親粒子のクォーク・フレーバーを識別するフレーバータギングが行われる。

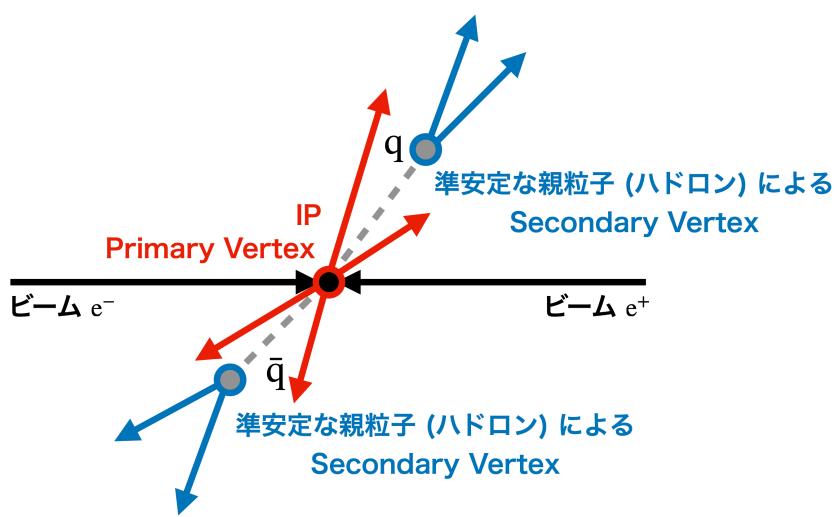


図1.6: primary vertexとsecondary vertexの図示。左右から電子・陽電子ビームが入射され、図中央で衝突したと仮定している。灰色の破線は準安定なハドロンを表現しており、赤線の飛跡と共にIP(primary vertex)から生じている。ハドロンは更に図右上と左下で崩壊し、secondary vertexを残す。青線はこのsecondary vertex由来の飛跡である。

崩壊点検出では、二本以上の飛跡について交点を求めるフィッティングを用いている。この時、フィッティングの健全性は $\chi^2$ 値によって把握している。

まず primary vertex の再構成を Tear-Down 法によって行う。具体的には予想されるビームスポットと事象中の全飛跡についてフィッティングを行い、 $\chi^2$ 値が一定以下になるまで $\chi^2$ 値への寄与の大きい飛跡を一本ずつ取り除いていく。残った飛跡を primary vertex 由来であると判定し、以降の再構成に用いる。

次に secondary vertex の再構成を Build-Up 法によって行う。ここでは、primary vertex に含まれていない飛跡について、二本の飛跡(飛跡対)の全ての組み合わせを作りフィッティングを行う。そのようにして得られた $\chi^2$ 値と運動量の方向、不変質量などをカットベースに判定し、secondary vertex の候補を選別する。更に、この飛跡対に対して飛跡を加えていくことで secondary vertex を再構成している。

崩壊点検出で得られた崩壊点を用いて、ジェットクラスタリングが行われる。ジェットクラスタリングでは崩壊点の情報を使用して中性粒子の飛跡を含めた全ての飛跡についてクラスタリングを行う。LCFIPplus ではこのジェットクラスタリングの前に、重いハドロン粒子のセミレプトニック崩壊によって生じた孤立レプトンをエネルギーやインパクトパラメータを用いて探索している。次に得られた孤立レプトンと secondary vertex の組み合わせを行う。孤立レプトンについては運動量方向を、secondary vertex については primary vertex からの角度を用いそれらの開き角が一定の閾値以下であるかによって評価している。これらの手順により再構成された孤立レプトンや崩壊点はジェットクラスタリングのコアとして使用される。ジェットクラスタリングは Durham アルゴリズム [11] を用いており、粒子の運動量方向とジェットの開き角、修正 Durham 距離に基づいてジェットの再構成を行っている。

LCFIPplus ではジェットのフレーバーをより精度よく同定するために jet vertex refiner アルゴリズムが使用されている。jet vertex refiner はシングルトラック崩壊点検出と崩壊点コンバイナの二つのステップで行われる。シングルトラック崩壊点検出では崩壊点を一つしか含まないジェットについて、飛跡を一本だけ含む疑似崩壊点を探索するアルゴリズムである。崩壊点コンバイナは二つ以上の崩壊点を含むジェットについて崩壊点の個数を最大二つに結合させるアルゴリズムである。ジェット中の崩壊点について更に一つに纏められる場合は新たな崩壊点への置き換えを行う。

クラスター化されたジェットについてフレーバー識別が行われる。フレーバータギングでは Boosted Decision Trees (BDTs) を用いて親粒子のフレーバーを識別している。BDTs は ROOT の TMVA パッケージを使用し、ボトム・フレーバーのジェット、チャーム・フレーバーのジェット、アップ、ダウン、ストレンジ・フレーバーのジェットの三つにクラスで分類している。入力変数はジェット内の崩壊点や飛跡などの構成要素によって作成され、ジェットのエネルギーに依存する様な変数については規格化を行っている。

以上がジェットの再構成である。様々な物理解析において、ジェットの個数やそのフレーバーの識別は信号事象と背景事象の弁別や物理解析などに使用されている。したがって、ジェットの再構成の性能向上はあらゆる物理解析の性能向上と直結していると言える。

## 1.6 本研究の目的

本研究の目的は、深層学習を使用して 1.5.3 項で紹介した崩壊点検出を開発・改善することである。

1.3 節でも述べた様に ILC はヒッグス粒子の精密測定を通じた新物理の発見を目的としている。ILC では、 $\text{BR}(h \rightarrow b\bar{b}, c\bar{c}, gg, WW^*, ZZ^*)$  の精密測定、特に  $\text{BR}(h \rightarrow b\bar{b}, c\bar{c})$  を通じて、第 3, 2 世代のクォークとヒッグス粒子についての湯川結合の更なる理解が期待されている。この終状態  $b\bar{b}$ ,  $c\bar{c}$  の違いの把握には、高性能なジェット解析・識別は必須である。

ILC では現在 LCFIPlus 内の崩壊点検出が使用されているが、primary vertex や secondary vertex の選別に人が定めた閾値が多く含まれており、それら閾値に基づいたデータの剪定を行っている。<sup>\*2</sup>このような人が定めた閾値は最適ではなく、何らかの情報を欠損してしまっている可能性がある為、データの持つ情報をより十全に活かし柔軟に判断できる手法の開発が必要である。この様な手法の一つとして深層学習(2章)が考えられる。

深層学習は近年非常に発展目覚しい技術領域であり、粒子物理学の分野では積極的に取り入れる事で事象再構成や物理解析の精度の向上が期待されている。一方で深層学習の技術は難解な部分も多い為、有効に活用されている例は少なく、ILC の事象再構成においてもほとんど使用されていない。

本研究では、この深層学習をジェット再構成の最初段である崩壊点検出アルゴリズムに適用する事で、これまで用いていた手法の性能改善や解析時間の短縮などを目指す。また、最終的には図 1.7 に示すように、全ての事象再構成アルゴリズムを深層学習やその他の機械学習技術に置き換えることを目標としている。

更に、本研究は深層学習のネットワークやアルゴリズムを現在使用しているソフトウェア上で動作させることに關しても意義のある研究である。その様な深層学習使用についての前例と実装を示すことで、ILC での後進の深層学習研究への礎としたい。

## 1.7 本論文の流れ

本章と 2 章は本論文の導入である。

2 章では本論文の核となる技術である深層学習について解説を行う。ここでは、本研究を理解する為に必要な技術領域や背景理論について簡潔な導入を行い、3 章以降では、この 2 章の内容を前提とした議論を行う。ただし深層学習に関する種々のテクニックについては経験則によるものが多いため、本研究で使用したものについては 2 章では説明せず、3 章で述べる。また、具体的な実装に關しても同様に 2 章では記載せず、付録 A にまとめる事とする。

3 章と 4 章、5 章は本論文の本題である。3 章では本研究で使用するデータと作成した深層学

---

<sup>\*2</sup> この様な手法をカットベースという。

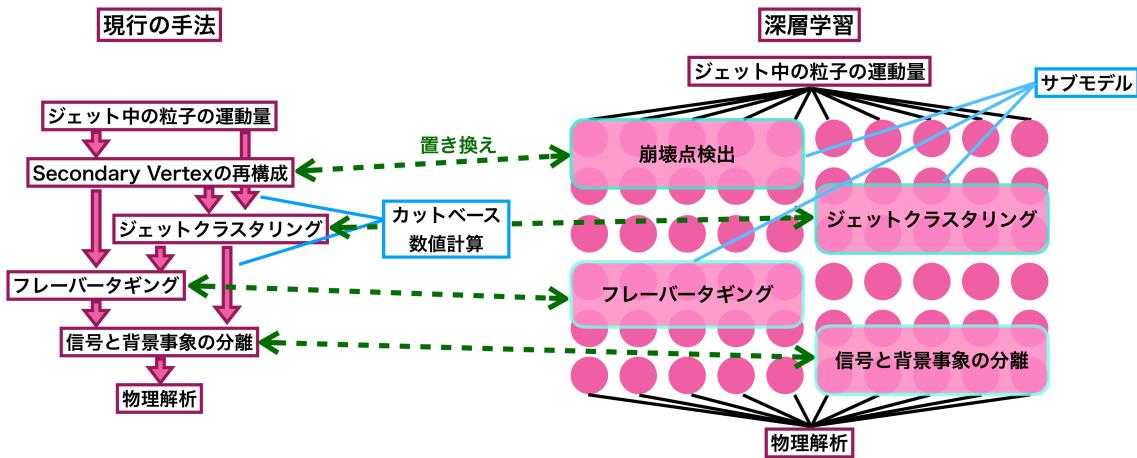


図 1.7: 深層学習によるジェットの再構成。図左は現行の手法、図右は目標としている深層学習を用いた再構成手法の概念図である。現行の手法では多くの場合、数値計算やそこから得られる変数に閾値を課してカットベースの評価が行われている。現在は深層学習への置き換えの第一段階として、それぞれの役割に特化したサブモデルと入れ替えを行い性能の改善を図っている。

習のネットワークについて、その構造の解説や評価を行う。また、深層学習を使用した崩壊点検出についての発想や計算環境に関しても 3 章で述べる。

4 章では 3 章で作成したネットワークを用いた崩壊点検出について、アルゴリズムや各種最適化、簡単な評価を行う。

5 章ではその様にして得られた崩壊点検出と、LCFIPplus で使用されている現行の崩壊点検出との比較について述べる。

6 章は本論文の結論である。ここでは、本研究をまとめると共に今後の展望について述べる。

## 第 2 章

# 深層学習

本章では、深層学習 (Deep Learning, DL) について述べる。本章の作成にあたり、参考文献 [12, 13, 14] を使用した。

まず深層学習の導入として、2.1 節で機械学習 (Machine Learning, ML) と深層学習の概要について簡単に紹介する。

次に 2.2 節では、深層学習を理解する上で前提となるパーセプトロン (Perceptron) というネットワークを解説する。パーセプトロンは、後述するニューラルネットワークの先駆けとなる技術である。

2.3 節では、深層学習の基礎技術であるニューラルネットワーク (Neural Network, NN) を導入する。主に、2.3.1 項でニューラルネットワークを構築するために必要な計算手順について、2.3.2 項でニューラルネットワークの学習に関して重要な技術要素についてそれぞれ説明する。

深層学習は、2.3 節までの基盤的な技術を使用するだけでも様々な問題に対応できるが、扱うデータや問題の性質によって、更に応用的な使い方が求められる。本研究ではそのような応用技術を幾つか用いてネットワークの作成を行なっている。その為、2.4 節や 2.5 節ではそのような深層学習の応用技術の解説を行う。

2.4 節では、系列データを取り扱うためのリカレントニューラルネットワーク (Recurrent Neural Network, RNN) を導入する。2.5 節では、近年注目されている注意機構 (Attention) と呼ばれる技術について説明する。

最後に 2.6 節にて、ニューラルネットワークが持つハイパーパラメータについてまとめる。

### 2.1 機械学習と深層学習

深層学習とは、機械学習の技術の一つである。本節ではまず、この機械学習について簡単に説明し、その後、機械学習における深層学習の位置付けを述べる。

機械学習とは、データに現れるパターンや統計情報を計算機（学習器）に「学習」させることによって、逐一プログラミングをすることなく未知の問題に対応させる為の技術である。これは人間の持つ知性を機械に実現する、人工知能（Artificial Intelligence, AI）に関する研究の一分野であると言える。このような研究は、1956年のダートマス会議 [15] から始まり、現在は第三期のAIブームと言われている。機械学習は、機械（計算機）が独自に未知の問題を解く為の技術や手法の総称であるが、問題に対するアプローチの仕方によって、教師あり学習、教師なし学習、強化学習などに分類することができる（図2.1）。

- 教師あり学習

教師あり学習とは、訓練データ（Training data）と呼ばれる正解がラベル付けされたデータを用い、学習器の出力を正解に近付けるように学習器を更新していく手法である。主にクラス分類を行う分類問題や、連続値を予測する回帰問題などの問題を解くことができる。具体的な例としてサポートベクターマシン（Support Vector Machine, SVM [16, 17]）や決定木（Decision Tree）などが挙げられる。

- 教師なし学習

教師なし学習とは、訓練データを用いず、データの持つ数学モデルや構造を抽出する技術である。主にクラスタリングや次元圧縮などに使用され、代表的な手法は、k平均法や主成分分析などである。

- 強化学習

強化学習とは、環境とのやり取りから報酬を受け取り、エージェントを構築していく手法である。学習は報酬を最大化するように進み、教師あり学習の一分野のようにみなす事も出来るが、強化学習は一連の行動に対しての報酬を考慮する点で異なる。強化学習は様々な分野で使用されているが、主に長期的な戦略が必要となるゲームなどの領域で用いられている。

深層学習は、このような機械学習の中で、基本的には、回帰問題や分類問題などを解く教師あり学習に分類される。しかし近年では、半教師あり学習やディープクラスタリング、深層強化学習といった様々な技術的応用が提案されている。次節以降では、この深層学習の基盤技術について紹介する。

## 2.2 パーセプトロン

パーセプトロンは深層学習の基礎となる技術であり、1958年、Rosenblattによって提案された [18]。ここでは、このパーセプトロンについて解説することで、次節のニューラルネットワークへの導入とする。

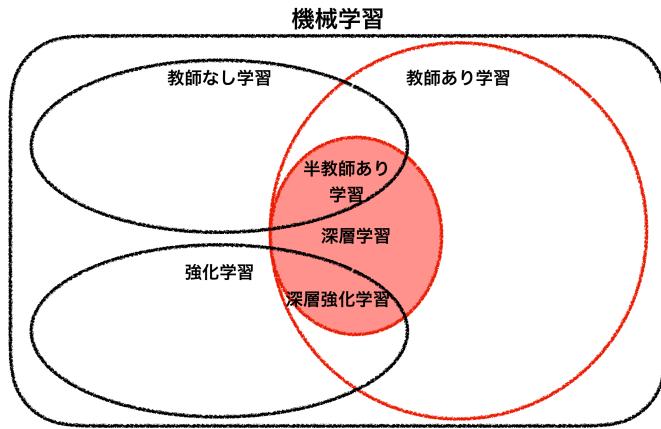


図 2.1: 機械学習の中の深層学習の位置付け。深層学習は基本的に教師あり学習であるが、半教師あり学習分類やクラスタリングなど応用的な研究が行われている。

### 2.2.1 単純パーセプトロン

パーセプトロンとは、情報を伝達するネットワークである。ここでのネットワークとは、ある情報を受け取り、それを後方へ伝達するような構造のことを言うものとする。まず、最も簡単なパーセプトロンとして、図 2.2 のような構造を考える。図 2.2 は、二つの入力  $x_1, x_2$  を受け取り、一つの出力  $y$  を行なっているネットワークである。このような入力や出力の数や入力や出力そのものの事をノードやニューロンという。また、図 2.2 のように、ただ入力と出力のみを持っているパーセプトロンを特に単純パーセプトロン (Simple Perceptron) という。

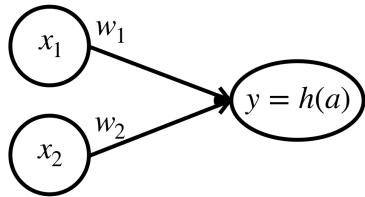


図 2.2: 単純パーセプトロン。入力を  $x_1, x_2$ 、出力を  $y$  と置いた。また、各ノードの繋がりを矢印で表現し、同時にそれぞれの線は学習可能な重みを表している。

単純パーセプトロンの情報伝達は、簡単な計算で定義される。出力  $y$  は、 $x_1, x_2$  とそれとの重み  $w_1, w_2$  を用いて、

$$\begin{aligned} y &= h(a) \\ a &= w_1x_1 + w_2x_2 \end{aligned} \tag{2.1}$$

と計算される。

ここで、出力  $y$  は関数  $h$  によって変換されている。このような関数を活性化関数 (Activation function) という。特に単純パーセプトロンでは、活性化関数  $h$  としてヘヴィサイドの階段関数 (図 2.3) を用いる。

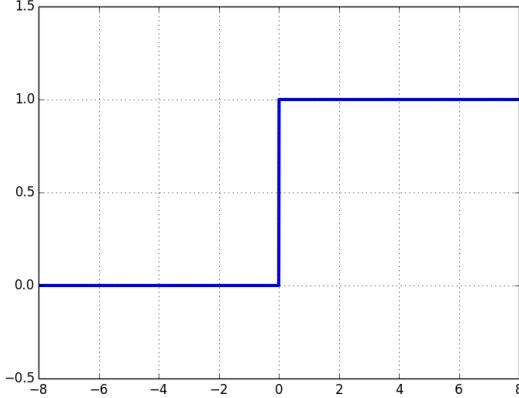


図 2.3: ヘヴィサイドの階段関数

その閾値を  $\theta$  とすると、出力は更に、

$$y = h(a) = \begin{cases} 0 & (a = w_1x_1 + w_2x_2 \leq \theta) \\ 1 & (a = w_1x_1 + w_2x_2 > \theta) \end{cases} \quad (2.2)$$

と書ける。この時、単純パーセプトロンはある一定値  $\theta$  までは “0”，それ以上であれば “1” を返す二値信号のネットワークであると考えることが出来る。パーセプトロンやニューラルネットワークにおいて、学習可能なパラメータは重み  $w_1, w_2$  であり、これら重みを更新していく操作を学習 (トレーニング, Training) という。

## 2.2.2 多層パーセプトロン

単純パーセプトロンは線形な問題を解く事しか出来なかつたが、層を重ねることで非線形に対応できるという点で非常に高い発展性を持っていた [19]。そのように単純パーセプトロンを重ねたネットワークの事を多層パーセプトロン (Multi Layer Perceptron, MLP) という。多層パーセプトロンは図 2.4 のように表現出来る。

多層パーセプトロンは単純パーセプトロンとは異なり、入力、出力以外に、中間層 (隠れ層) を持っている。ここで、中間層を  $x_1^1, x_2^1$  と置くと、単純パーセプトロンと同様に入力  $x_1^0, x_2^0$  とそれぞれの重み  $w_{11}^0, w_{12}^0, w_{21}^0, w_{22}^0$  を用いて

$$\begin{aligned} x_1^1 &= w_{11}^0 x_1^0 + w_{21}^0 x_2^0 \\ x_2^1 &= w_{12}^0 x_1^0 + w_{22}^0 x_2^0 \end{aligned} \quad (2.3)$$

と計算でき、また出力  $y$  についても、 $x_1^1, x_2^1$  とそれぞれの重み  $w_1^1, w_2^1$  を用いて、

$$\begin{aligned} y &= h(a) \\ a &= w_1^1 x_1^1 + w_2^1 x_2^1 \end{aligned} \quad (2.4)$$

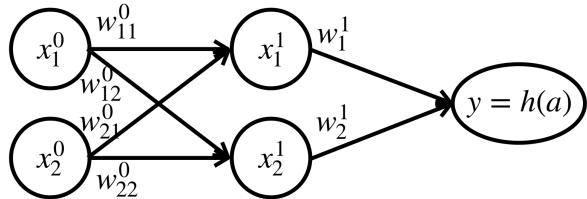


図 2.4: 多層パーセプトロン。図 2.2 と同様にノードを繋ぐそれぞれの線は学習可能な重みを表現している。前方のノードは後方のノードの全てと接続されている点にも注意が必要である。

となる。

以後、特に断らない場合はあるノード  $x$ , 重み  $w$  について、層の深さ、前後のノードを次のように表現する。

$$\begin{aligned} x^{(\text{層の深さ})}_{(\text{ノード番号})} \\ w^{(\text{層の深さ})}_{(\text{前のノード番号}) (\text{後ろのノード番号})} \end{aligned} \quad (2.5)$$

多層パーセプトロンは、学習の手法や層を重ねるに連れて重みが更新出来なくなる勾配消失問題など様々な課題を抱えていた。次節ではこれらの問題を後述する誤差逆伝播法 (Backpropagation) や活性化関数によって解決したニューラルネットワークについて解説する。

## 2.3 ニューラルネットワーク

本節ではニューラルネットワークについて解説を行う。ただし、ここでは計算手法や言葉の定義についてのみ述べる。実装方法については現在様々なフレームワーク [20, 21, 22, 23] があり、それぞれで実装の仕方が異なっている。本研究では tensorflow-keras を用いた。<sup>\*1</sup>

ニューラルネットワークに関する技術は「ニューラルネットワークの構造」についてと「ニューラルネットワークの学習」についてに大きく分けられると考えている。前者は主に入力から出力までのネットワークの構築を、後者は構築されたネットワークの重み更新についての技術である。

---

<sup>\*1</sup> 具体的なコードに関しては付録 A を参照。

### 2.3.1 ニューラルネットワークの構造

ニューラルネットワークは様々な技術によって支えられているが、その基本構造は前節の多層パーセプトロンと全く同じである。ニューラルネットワークと多層パーセプトロンとの大きな構造の違いは活性化関数である。ニューラルネットワークでは様々な活性化関数が提案されており<sup>\*2</sup>、これが勾配消失問題を解消する鍵となっている。活性化関数は重み更新のために微分可能な関数である必要があるが、どのような関数を選ぶかはユーザーに委ねられている。勾配消失や重み更新についての詳しい解説は 2.3.2 節で行う。以下に活性化関数の例を示す(図 2.5)。

- 階段関数

$$h(a) = \begin{cases} 0 & (a \leq \theta) \\ 1 & (a > \theta) \end{cases} \quad (2.6)$$

- シグモイド関数

$$h(a) = \frac{1}{1 + \exp(-a)} \quad (2.7)$$

- tanh 関数

$$h(a) = \tanh(a) \quad (2.8)$$

- ReLU (Rectified Linear Unit, ランプ) 関数 [24]

$$h(a) = \begin{cases} 0 & (a \leq \theta) \\ a & (a > \theta) \end{cases} \quad (2.9)$$

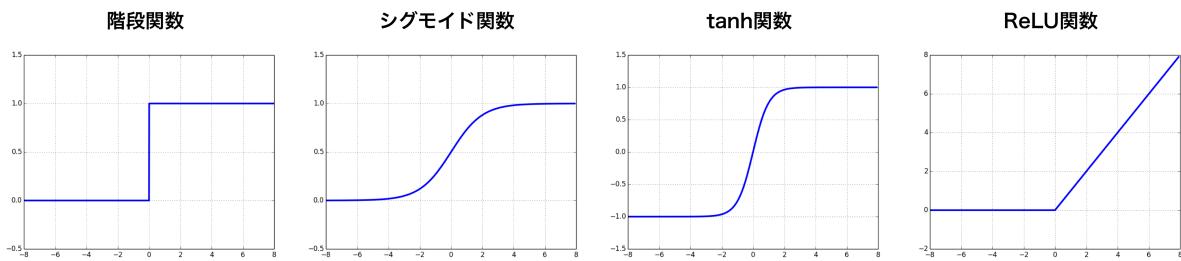


図 2.5: 活性化関数

一般に、ニューラルネットワークは多層パーセプトロンと同様の構造であるので、図 2.6 のように表現出来る。

---

<sup>\*2</sup> 多層パーセプトロンはニューラルネットワークの内、活性化関数に階段関数を使った特別なネットワークであると再定義できる。

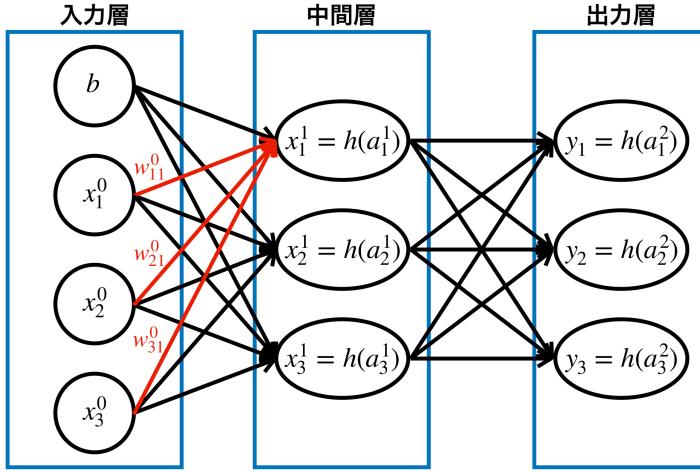


図 2.6: ニューラルネットワーク。基本構造は多層パーセプトロンと同様であるとわかる。赤線は中間層  $x_1^1$  へ入力される重みである。

ここで、中間層  $x_1^1$  は入力  $x_1^0, x_2^0, x_3^0$  とそれぞれの重み  $w_{11}^0, w_{21}^0, w_{31}^0$  を用いて、

$$\begin{aligned} x_1^1 &= h(a_1^1) \\ a_1^1 &= w_{11}^0 x_1^0 + w_{21}^0 x_2^0 + w_{31}^0 x_3^0 + b_1^0 \end{aligned} \quad (2.10)$$

と計算できる。また、バイアスとして  $b$  を導入している。これはパーセプトロンの閾値  $\theta$  に対応している。<sup>\*3</sup> 中間層  $x_2^1, x_3^1$  についても同様に、

$$\begin{aligned} x_2^1 &= h(a_2^1) \\ a_2^1 &= w_{12}^0 x_1^0 + w_{22}^0 x_2^0 + w_{32}^0 x_3^0 + b_2^0 \\ x_3^1 &= h(a_3^1) \\ a_3^1 &= w_{13}^0 x_1^0 + w_{23}^0 x_2^0 + w_{33}^0 x_3^0 + b_3^0 \end{aligned} \quad (2.11)$$

と書ける。

また、これら  $x_1^1, x_2^1, x_3^1$  の計算は行列とベクトルを用いて、より簡潔に表現できる。

$$\begin{aligned} \mathbf{x}^1 &= \begin{pmatrix} x_1^1 \\ x_2^1 \\ x_3^1 \end{pmatrix} = h(\mathbf{a}^1) \\ \mathbf{a}^1 &= \begin{pmatrix} a_1^1 \\ a_2^1 \\ a_3^1 \end{pmatrix} = W^0 \mathbf{x}^0 + \mathbf{b}^0 = \begin{pmatrix} w_{11}^0 & w_{21}^0 & w_{31}^0 \\ w_{12}^0 & w_{22}^0 & w_{32}^0 \\ w_{13}^0 & w_{23}^0 & w_{33}^0 \end{pmatrix} \begin{pmatrix} x_1^0 \\ x_2^0 \\ x_3^0 \end{pmatrix} + \begin{pmatrix} b_1^0 \\ b_2^0 \\ b_3^0 \end{pmatrix} \end{aligned} \quad (2.12)$$

行列  $W^0$  とベクトル  $\mathbf{b}^0$  は学習可能な重みであり、得られた新たな状態ベクトル  $\mathbf{a}^1$  は微分可能な任意の活性化関数  $h$  によって、中間層  $\mathbf{x}^1$  へと変換される。以下、これを繰り返すことによって、ネットワークは構築されている。

<sup>\*3</sup>  $b$  を左辺に移行する。

出力層における活性化関数は一般に回帰問題では恒等関数を、分類問題ではソフトマックス関数と呼ばれる関数を使用する。回帰問題において、最終的な出力は数値（連続値）であるため、恒等関数によって、変換を行わずそのまま出力することが一般である。

$$y_k = h(a_k^2) = a_k^2 \quad (2.13)$$

一方で、分類問題では、最終的な出力は分類されたクラスとなるため、以下のようなソフトマックス関数を使用する。

$$y_k = h(a_k^2) = \frac{\exp(a_k^2)}{\sum_{i=1}^N \exp(a_i^2)} \quad (2.14)$$

このソフトマックス関数は、分母が総和、分子がその一要素の形をしており、 $y_k$  を  $k$  について足し合わせると 1 になることがわかる。このことから、出力  $y_k$  は  $k$  番目のクラスについての確率として解釈でき、分類問題において、どのクラスにどの程度該当するかを表現することに相当している。

前述したように、これは最も基本的なニューラルネットワークであり、このような全結合（Fully connected, Dense）な層を重ねたネットワークを順伝播型（フィードフォワード）ニューラルネットワーク（Feedforward Neural Network）という。

### 2.3.2 ニューラルネットワークの学習

教師あり学習であるニューラルネットワークにおける学習は、損失関数（コスト関数、Loss function）を最小化するように、重みを更新していく事で行われる。損失関数とは、訓練データの正解ラベルとネットワークの出力がどの程度離れているかを計算するための関数である。この損失関数は取り組む問題や訓練データの性質によって適切に選択する必要がある。ここでは、よく使用される損失関数として以下の二つを挙げる。

- 交差エントロピー誤差 (Categorical Cross Entropy)

$$L = - \sum_k^N t_k \log y_k \quad (2.15)$$

- 平均二乗誤差 (Mean Squared Error)

$$L = \frac{1}{N} \sum_k^N (t_k - y_k)^2 \quad (2.16)$$

$t_k, y_k$  はそれぞれ  $k$  番目の正解ラベルとクラスの出力（確率や値）を示している。分類問題については交差エントロピー誤差が、回帰問題については平均二乗誤差が主に使用される。

分類問題において、正解ラベル  $t$  は、あるクラスに関して 0 か 1 かのベクトル (one-hot vector) で表現されることが一般的である。例えば、赤、青、緑について分類を行う場合 (3 クラス分類という)、赤を  $(1, 0, 0)$ 、青を  $(0, 1, 0)$ 、緑を  $(0, 0, 1)$  と定義する。また、ネットワークの出力  $y$  はどのクラスに属するかの確率となっている。例えば、赤、青、緑がそれぞれ 80 %, 10 %, 10 % の場合は出力  $y$  は  $(0.8, 0.1, 0.1)$  と書ける。したがって、正解ラベルを赤とするとき損失関数  $L$  は

$$\begin{aligned} L &= - \sum_k^3 t_k \log y_k \\ &= -t_1 \log y_1 - t_2 \log y_2 - t_3 \log y_3 \\ &= -1 \cdot \log 0.8 = 0.22314\cdots \end{aligned} \tag{2.17}$$

と計算される。

回帰問題において、出力  $y$ 、正解ラベル  $t$  は共に連続値であるため、平均二乗誤差のような二つの差を用いる損失関数が一般的である。

前述したように、ネットワークの学習はこの損失関数を最小化するように進む。今、損失関数は変数  $y_k$  の関数で表現出来ており、このような関数の最小値を求めるためには、単に変数  $y_k$  を用いて偏微分を行い勾配を求めれば良い。計算機において、このような勾配を求め、徐々に関数を最小化していく手法を勾配降下法 (Gradient Descent Method) という。勾配降下法において、次のステップの変数  $y'_k$  は次のように計算される。

$$y'_k = y_k - \eta \frac{\partial L}{\partial y_k} \tag{2.18}$$

ここで、ステップ幅を決定する定数  $\eta$  をニューラルネットワークにおいて学習率 (learning rate) という。学習率は 0.001 などの定数を問題やネットワークによって適切に選ぶ必要がある。このようなネットワークについて更新されない初期設定のパラメータをハイパーパラメータという。ハイパーパラメータについては後の 2.6 節で述べる。

ニューラルネットワークにおいて、各重みを更新するための勾配は連鎖律 (chain rule) を用いて計算される。これは更新する重みと最小化される損失関数の間に output 層と活性化関数が存在しているためである。<sup>\*4</sup> 具体的には、ある重み行列  $W$  に対して、勾配降下法、連鎖律を考慮すると、次のステップの重み行列  $W'$  は

$$\begin{aligned} W' &= W - \eta \frac{\partial L}{\partial W} \\ &= \left( \begin{array}{ccc} w_{11} & w_{21} & w_{31} \\ w_{12} & w_{22} & w_{32} \\ w_{13} & w_{23} & w_{33} \end{array} \right) - \eta \left( \begin{array}{ccc} \frac{\partial L}{\partial w_{11}} & \frac{\partial L}{\partial w_{21}} & \frac{\partial L}{\partial w_{31}} \\ \frac{\partial L}{\partial w_{12}} & \frac{\partial L}{\partial w_{22}} & \frac{\partial L}{\partial w_{32}} \\ \frac{\partial L}{\partial w_{13}} & \frac{\partial L}{\partial w_{23}} & \frac{\partial L}{\partial w_{33}} \end{array} \right) \\ &= W - \eta \sum_k \frac{\partial L(y_k)}{\partial y_k} \frac{\partial y_k}{\partial a_k} \frac{\partial a_k}{\partial W} \end{aligned} \tag{2.19}$$

<sup>\*4</sup> 損失関数はクラスの出力の関数であり、クラスの出力は活性化関数によって計算され、活性化関数は出力層の関数である。

と計算される。

このような最適化問題に関して、いくつかのアルゴリズムが提案されている。現在は確率的勾配降下法 (Stochastic Gradient Descent, SGD [25]) やそれを基礎とした RMSProp [26], Adam [27]などの手法がよく使用されている。

学習方法における、ニューラルネットワークと多層パーセプトロンの大きな違いは、重みの更新を出力層から逆伝播させる誤差逆伝播法 (Backpropagation [28]) という手法の有無である。再度、図 2.6 を考える。全ての出力、中間層を行列計算を用いて記述すると、

$$\begin{aligned} \mathbf{a}^1 &= \begin{pmatrix} a_1^1 \\ a_2^1 \\ a_3^1 \end{pmatrix} = W^0 \mathbf{x}^0 + \mathbf{b}^0 = \begin{pmatrix} w_{11}^0 & w_{21}^0 & w_{31}^0 \\ w_{12}^0 & w_{22}^0 & w_{32}^0 \\ w_{13}^0 & w_{23}^0 & w_{33}^0 \end{pmatrix} \begin{pmatrix} x_1^0 \\ x_2^0 \\ x_3^0 \end{pmatrix} + \begin{pmatrix} b_1^0 \\ b_2^0 \\ b_3^0 \end{pmatrix} \\ \mathbf{x}^1 &= h(\mathbf{a}^1) \\ \mathbf{a}^2 &= \begin{pmatrix} a_1^2 \\ a_2^2 \\ a_3^2 \end{pmatrix} = W^1 \mathbf{x}^1 = \begin{pmatrix} w_{11}^1 & w_{21}^1 & w_{31}^1 \\ w_{12}^1 & w_{22}^1 & w_{32}^1 \\ w_{13}^1 & w_{23}^1 & w_{33}^1 \end{pmatrix} \begin{pmatrix} x_1^1 \\ x_2^1 \\ x_3^1 \end{pmatrix} \\ y_k &= \sigma(a_k^2) = \frac{\exp(a_k^2)}{\sum_{i=1}^n \exp(a_i^2)} \end{aligned} \tag{2.20}$$

と書ける。ここで、出力部分のソフトマックス関数を  $\sigma$  と書いた。

ある重み  $w_{11}^1$  について考える。損失関数  $L$  の重み  $w_{11}^1$  による偏微分は、連鎖律を考慮して、

$$\begin{aligned} y_1 &= \sigma(a_1^2) \\ a_1^2 &= w_{11}^1 x_1^1 + w_{21}^1 x_2^1 + w_{31}^1 x_3^1 = \sum_{i=1}^N w_{i1}^1 x_i^1 \end{aligned} \tag{2.21}$$

より、

$$\begin{aligned} \frac{\partial L}{\partial w_{11}^1} &= \frac{\partial L}{\partial y_1} \frac{\partial y_1}{\partial a_1^2} \frac{\partial a_1^2}{\partial w_{11}^1} \\ &= \frac{\partial L}{\partial y_1} \frac{\partial \sigma(a_1^2)}{\partial a_1^2} x_{11}^1 \end{aligned} \tag{2.22}$$

と計算できる。

ここで、勾配の計算に活性化関数の偏微分が常に積の形で含まれていることがわかる。この活性化関数の偏微分が 0 になり、そこから抜け出せなくなると、その勾配は常に 0 になり消失してしまう、これが勾配消失である。勾配消失に陥った場合は、重みが適切に更新されず、学習が不十分になってしまふ。このような問題は活性化関数を変更することによって改善され、現在は ReLU 関数がよく用いられている。

また、更に浅い層の重み  $w_{11}^0$  について考えると、

$$\begin{aligned} x_1^2 &= h(a_1^1) \\ a_1^1 &= w_{11}^0 x_1^0 + w_{21}^0 x_2^0 + w_{31}^0 x_3^0 = \sum_{i=1}^N w_{i1}^0 x_i^0 \end{aligned} \tag{2.23}$$

より、

$$\begin{aligned}
 \frac{\partial L}{\partial w_{11}^0} &= \sum_k^N \frac{\partial L}{\partial y_k} \frac{\partial y_k}{\partial a_k^2} \frac{\partial a_k^2}{\partial x_1^2} \frac{\partial x_1^2}{\partial a_1^1} \frac{\partial a_1^1}{\partial w_{11}^0} \\
 &= \sum_k^N \frac{\partial L}{\partial y_k} \frac{\partial y_k}{\partial a_k^2} \frac{\partial a_k^2}{\partial x_1^2} \frac{\partial h(a_1^1)}{\partial a_1^1} x_{11}^0 \\
 &= \sum_k^N \left( \frac{\partial L}{\partial y_k} \frac{\partial y_k}{\partial a_k^2} w_{1k}^1 \right) \frac{\partial h(a_1^1)}{\partial a_1^1} x_{11}^0
 \end{aligned} \tag{2.24}$$

と計算できる。この計算は更に層を重ねた場合でも同様の手順で行うことができる。

重みの更新は基本的に全ての訓練データを使用するのではなく、訓練データをいくつかの塊に分け、その塊について損失関数を計算することで行われる。このような手法はミニバッチ学習と呼ばれる。(訓練データ全てを用いたものをバッチ学習という。) ミニバッチ学習に使用されるデータの数をミニバッチサイズ(あるいは単にバッチサイズ)という。これも後述するハイパーパラメータの一つである。ミニバッチ学習はバッチ学習と比較して二つの利点が存在する。

一つは膨大なデータを直接処理しなくても良いという点である。一般に深層学習で使用されるデータは非常に膨大であり、GPUなどのメモリに乗らない場合があるが、ミニバッチ学習ではこれを回避することができる。

もう一つは学習が停滞しづらいという点である。訓練データと比較してサイズの小さいミニバッチは上記の勾配が0になりやすく、局所的な最小点での学習の停滞を回避することができる。

ただしミニバッチ学習のバッチサイズが小さくなつた場合には、損失関数が平均化されず学習が不安定になる(収束しなくなる)という問題が生じる場合がある。<sup>\*5</sup>

### 2.3.3 ディープニューラルネットワーク

ディープニューラルネットワーク(Deep Neural Network, DNN)という言葉の定義は非常に曖昧である。<sup>\*6</sup> 2.1節で述べたように、現在は第三期AIブームであると言われている。これは上述してきた技術的成熟に加え、計算機の性能が向上したことにより、より層を重ねた(深い)ニューラルネットワークの学習が可能になった結果であると言える。2006年のHintonらによるauto-encoder[29]や2014年にIanによって提案された敵対的生成ネットワーク(Generative Adversarial Network, GAN[30])など、様々な発展的な応用がなされ、現在においても毎年のように新しいネットワークが提案されている。本節で述べたニューラルネットワークはその基盤技術の一部である。次節以降では、系列データを扱うためのニューラルネットワークの応用について紹介する。

---

<sup>\*5</sup> バッチサイズが1(1サンプルのみ)のミニバッチ学習をオンライン学習という。

<sup>\*6</sup> 少なくとも私ははつきりとした定義を存じない。

## 2.4 リカレントニューラルネットワーク

前節で紹介したようなフィードフォワードニューラルネットワークは系列データを扱う際、重み行列が固定的な大きさでしか保持出来ないという点<sup>\*7</sup>と直前の系列に依存した学習が出来ないという点に関して課題を抱えている。これらの課題を解決するために提案されたのが、リカレントニューラルネットワークというネットワーク構造である。本節では、このリカレントニューラルネットワークについて解説を行う。リカレントニューラルネットワークは主に系列データ、特に時系列データを取り扱うためのネットワークである。このような時系列に関するニューラルネットワークは自然言語処理などの分野で発展し、音声認識や機械翻訳といった技術に使用されている。リカレントニューラルネットワークの構造は、フィードフォワードニューラルネットワークと比較すると複雑であるが、要素計算は全結合であり、基本的にはその組み合わせで理解できる。2.4.1 項では、そのようなリカレントニューラルネットワークの構造と学習について述べる。その後、リカレントニューラルネットワークの抱える問題と、ゲート (Gate) と記憶セル (Cell) 呼ばれる技術によってその問題を克服した長・短期記憶 (Long Short-Term Memory, LSTM [31]) というネットワークを 2.4.2 項と 2.4.3 項でそれぞれ紹介する。

### 2.4.1 リカレントニューラルネットワークの構造と学習

リカレントニューラルネットワークの構造は、これまでのフィードフォワードニューラルネットワークとは大きく異なる。そのネットワーク構造はいくつかの表現方法が存在しているが、本論文では時間について展開した図で書くこととする。図 2.7 の左側が時間について展開していない図、右側が時間について展開した図である。左側では、時間についての構造をループで表現し、任意の時間  $t$  についての入力  $x_t$  と出力  $h_t$  を持つネットワークとして表現している。右側では、時間についての構造を展開し、出力や入力を系列情報とともに表現している。これまでのフィードフォワードネットワークは情報の伝達を左右に描いていたが、このリカレントニューラルネットワークは上下に描き、系列の流れを左右で表現されることが多い。図 2.7 の右側では、出力  $h$  が二つ存在し、上に進むものが出力に、右に進むものが次の系列への入力になっていることがわかる。このように、現在の出力を次の系列の入力として使うことで、直前の系列情報への依存性を導入している。

これまでと同様に明示的にネットワークの重みを描画すると、図 2.8 のようになる。図 2.7 の RNN に当たる部分が展開され、重みを線で表現した図になっている。図より、一つ前に系列の出力  $h_{t-1}$  と現在の系列の入力  $x_t$  を用いて、現在の系列の出力  $h_t$  が生成されていることがわかる。 $h$  についての赤い線に関する重み行列を  $W_h$ 、 $x$  についての黒い線に関する重み行列を  $W_x$  と置くと、出力  $h_t$  は次のように計算できる。

---

<sup>\*7</sup> 通常、深層学習の重み行列  $W$  は入力 × 出力の大きさであり入力の数を変更できない。

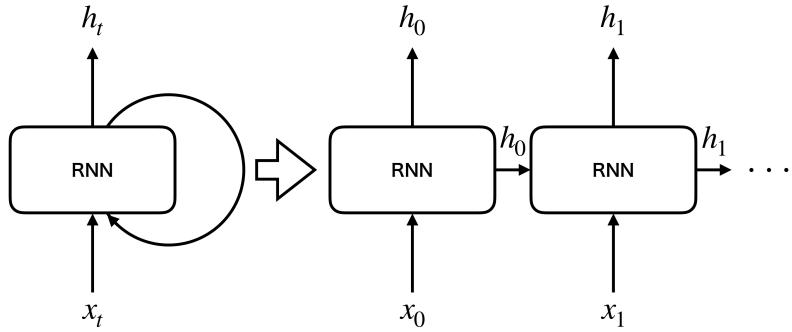


図 2.7: リカレントニューラルネットワーク。図左は時間について展開していない描画方法、図右は時間について展開した描画方法である。下方から入力され、上方に出力している。横方向は系列を表現しており、入力  $x_t$  は全ての系列で同じ形状でなければならない。

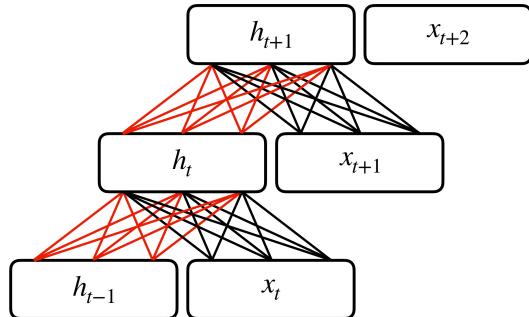


図 2.8: リカレントニューラルネットワークの重みの明示的な表現。赤線で示した重みは隠れ状態  $h$  についての重み  $W_h$ 、黒線で示した重みは隠れ層  $x$  についての重み  $W_x$  である。図 2.7 で示した“RNN”部はここでは線に相当している。ネットワークは上に積み重なっているように表現しているが、実際にはこれらの重み行列は全ての系列で同一のものである。

$$\mathbf{h}_t = \tanh(W_h \mathbf{h}_{t-1} + W_x \mathbf{x}_t)$$

$$\mathbf{a}_t = \begin{pmatrix} w_{h,11} & w_{h,21} & w_{h,31} \\ w_{h,12} & w_{h,22} & w_{h,32} \\ w_{h,13} & w_{h,23} & w_{h,33} \end{pmatrix} \begin{pmatrix} h_{t-1,1} \\ h_{t-1,2} \\ h_{t-1,3} \end{pmatrix} + \begin{pmatrix} w_{x,11} & w_{x,21} & w_{x,31} \\ w_{x,12} & w_{x,22} & w_{x,32} \\ w_{x,13} & w_{x,23} & w_{x,33} \end{pmatrix} \begin{pmatrix} x_{t,1} \\ x_{t,2} \\ x_{t,3} \end{pmatrix} \quad (2.25)$$

リカレントニューラルネットワークでは活性化関数として  $\tanh$  関数を使用している。前述したように、個々の要素計算はフィードフォワードニューラルネットワークの様に全結合で構成されていることがわかる。ここで、非常に重要な性質として、学習可能な重み行列  $W_h, W_x$  は全ての系列  $t$  について同一のものであり、大きさが不变であることに注意する。

このように再帰的に重み行列を使用することで、行列の大きさが可変でないという性質を回避し、系列情報を取り入れることに成功している。また、このような入力の系列長についての柔軟性は、長さが不定であるリアルタイムな時系列データを扱えるという点で重要である。

リカレントニューラルネットワークの出力方法は、問題によっていくつかのパターンが存在する。(図 2.9) 例えば、語句の分類の様な問題の場合は、一つの入力に対して、一つの出力を得る Many to Many という出力の作り方を行う。また、機械翻訳のデコーダーなど、一つの入力に対して、複数の出力を得たい場合は One to Many を用いる。最後に、感情分析の様に複数の入力に対して、一つの出力を得たい場合は Many to One を用いる。

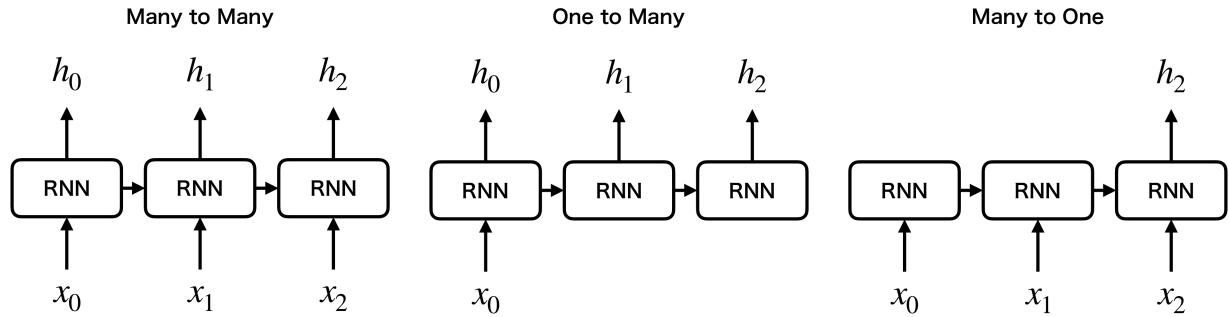


図 2.9: リカレントニューラルネットワークの出力方法

リカレントニューラルネットワークの学習は基本的にフィードフォワードニューラルネットワークと同様であるが、図 2.8 に見る様にネットワークは系列にしたがって深くなっている。この為、重み更新はこの系列を遡って行う必要がある。このような誤差逆伝播法の事を Backpropagation Through Time (BPTT) という。実際には計算リソースの削減のため、Truncated BPTT という、時系列方向に適当な長さで切り取り計算を行う手法が使用される。

## 2.4.2 リカレントニューラルネットワークの問題点

リカレントニューラルネットワークは時間方向に展開し、それを遡ることによって学習を行なっているため、系列の長さに依存して非常に深いネットワークが構築される。したがって、リカレントニューラルネットワークは真に深いネットワークであると言えるが、深い層からの勾配は非常に消失あるいは爆発しやすく、容易に勾配消失・爆発を招いてしまうという問題が生じている。勾配消失はリカレントニューラルネットワークの活性化関数である  $\tanh$  関数に起因している。2.3.2 項で解説したように、誤差逆伝播法は連鎖律によって計算され、その計算には活性化関数の微分が含まれている。ここで  $\tanh$  関数の微分は、

$$\begin{aligned} \frac{\partial y(x)}{\partial x} &= \frac{\partial}{\partial x} \tanh(x) = \frac{1}{\cosh^2(x)} \\ &= 1 - \frac{\cosh^2(x) - 1}{\cosh^2(x)} = 1 - \frac{\sinh^2(x)}{\cosh^2(x)} = 1 - \tanh^2(x) = 1 - y^2 \end{aligned} \tag{2.26}$$

と計算される。

$1 - y^2$  は、 $y = 0$  以外において常に 1 より小さい値を取ってしまう。リカレントニューラルネットワークでは系列長に応じて、この 1 より小さい値 ( $1 - y^2$ ) が複数回掛けられてしまうため、勾配消失が生じやすくなっている。同時にリカレントニューラルネットワークでは、連鎖律によって系列長に応じて同じ重み行列  $W_h$  を複数回掛けしており、この重み行列の値に応じて、勾配が発散あるいは消失してしまうことが考えられる。

また、リカレントニューラルネットワークは時系列上の複数の入出力から、矛盾した重み更新を受け取ってしまう入力重み衝突、出力重み衝突という問題も抱えている。

更に、リカレントニューラルネットワークはその構造上、長期的な系列情報を保持できないという課題も存在している。

以上のような問題を解決するため、ゲートとセルと呼ばれる技術を導入したものを、ゲート付きリカレントユニット (Gated Recurrent Unit, GRU [32]) という。次項では、このゲートを用いたリカレントニューラルネットワークの一つである LSTM について紹介する。

### 2.4.3 長・短期記憶 (Long Short-Term Memory, LSTM)

LSTM のネットワーク構造全体を図 2.10 に示す。リカレントニューラルネットワークとの最も大きな違いは、LSTM は隠れ状態 (出力) を二つ ( $h_t, c_t$ ) 持っているという点である。 $c_t$  は長期的な記憶セルを示しており、図 2.10 では上部の赤い線で表現されている。一方、 $h_t$  は、リカレントニューラルネットワークと同様に短期的な系列情報の伝達と出力に使用されている。LSTM は三つの入力  $h_{t-1}, c_{t-1}, x_t$  を受け取り、二つの出力  $h_t, c_t$  を提供するネットワークであるとみなすことができる。

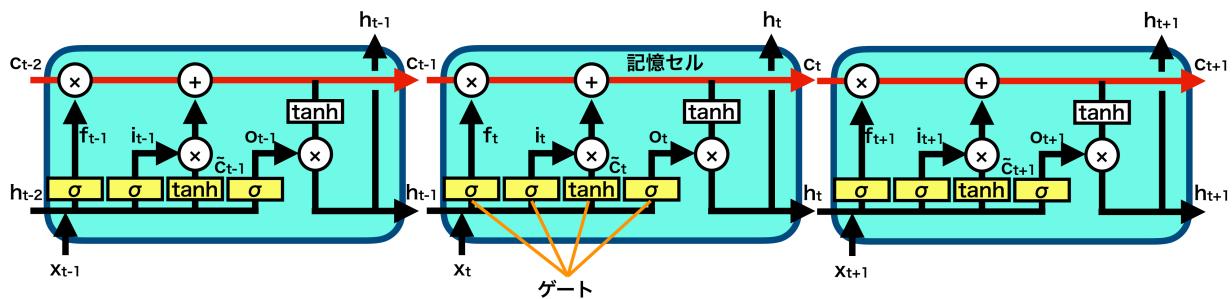


図 2.10: LSTM の流れ。リカレントニューラルネットワークの様に時間について展開した図である。赤線が記憶セル、黄色の各部がそれぞれゲートである。出力と隠れ状態である  $h, c$  は直前の系列から受け取り、入力  $x$  は下方から導入されている。

また、LSTM は内部に前項で述べた勾配消失や勾配爆発、入力重み衝突、出力重み衝突といった様々な問題を解決するためのゲートと呼ばれる構造を四つ持っている (図 2.10)。

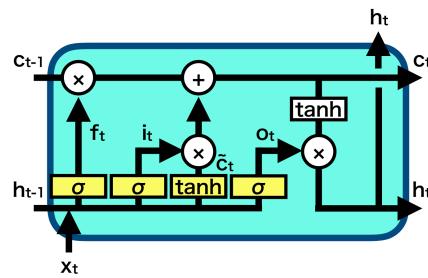


図 2.11: 単体の LSTM。系列 1 ステップ分の LSTM を切り出したものである。リカレントニューラルネットワークと同様に重み行列は各ステップに渡って同一のものが使用されている。

四つのゲートはそれぞれ忘却ゲート, 入力ゲート, セルの更新, 出力ゲートと呼ばれている。入力ゲート, 出力ゲートはそれぞれ重み衝突を回避するために導入されており, 忘却ゲート, セルの更新は長期記憶の適切な長期記憶セルの更新を行う為のアイデアである。それぞれの役割と演算を以下にまとめると。

- 忘却ゲート

図 2.12a の赤い線で表現している領域では, 入力  $\mathbf{h}_{t-1}, \mathbf{x}_t$  を用いて, どの程度, 直前の長期記憶セル  $c_{t-1}$  を忘れるかの度合いである  $f_t$  を計算している。 $f_t$  は, 次のようにかける。

$$f_t = \sigma(W_f \mathbf{x}_t + R_f \mathbf{h}_{t-1}) \quad (2.27)$$

したがって最終的には,

$$c_{t-1} f_t = c_{t-1} \sigma(W_f \mathbf{x}_t + R_f \mathbf{h}_{t-1}) \quad (2.28)$$

として, 次のゲート以降で長期記憶を更新するための容量を確保していると解釈できる。

- 入力ゲート

入力ゲートは忘却ゲートと全く同じ構造 (図 2.12b) をしており, 次のように計算できる。

$$i_t = \sigma(W_i \mathbf{x}_t + R_i \mathbf{h}_{t-1}) \quad (2.29)$$

忘却ゲートではどの程度, 長期記憶セルを忘れるかを計算していたように, 入力ゲートでの  $i_t$  は, 次のセルの更新時に新しい長期記憶セルをどの程度重視するかを表現していると解釈できる。

- セルの更新

図 2.12c では, まず更新された長期記憶セル  $\tilde{c}_t$  を計算している。

$$\tilde{c}_t = \tanh(W_c \mathbf{x}_t + R_c \mathbf{h}_{t-1}) \quad (2.30)$$

次に、これまでの三つのゲートの結果をまとめることで、新しい隠れ状態である長期記憶セル  $\mathbf{c}_t$  を計算できる。

$$\begin{aligned}\mathbf{c}_t &= \mathbf{c}_{t-1} \mathbf{f}_t + \tilde{\mathbf{c}}_t \mathbf{i}_t \\ &= \mathbf{c}_{t-1} \sigma(W_f \mathbf{x}_t + R_f \mathbf{h}_{t-1}) + \tanh(W_c \mathbf{x}_t + R_c \mathbf{h}_{t-1}) \sigma(W_i \mathbf{x}_t + R_i \mathbf{h}_{t-1})\end{aligned}\quad (2.31)$$

第一項では、直前の長期記憶セル  $\mathbf{c}_{t-1}$  をどの程度忘れるかを  $\mathbf{f}_t$  によって制御し、第二項では、新しく計算された長期記憶セル  $\tilde{\mathbf{c}}_t$  をどの程度重視するかを  $\mathbf{i}_t$  によって制御している。

- 出力ゲート

出力ゲートでは、ここまで計算された  $\mathbf{c}_t$  と入力  $\mathbf{x}_t, \mathbf{h}_{t-1}$  を用いて、最終的な出力となる  $\mathbf{h}_t$  を計算している。ただし、出力ゲート  $\mathbf{o}_t$  自体は入力ゲートや忘却ゲートと全く同じ形(図 2.12d)をしている。具体的な計算は次のように書ける。

$$\begin{aligned}\mathbf{o}_t &= \sigma(W_o \mathbf{x}_t + R_o \mathbf{h}_{t-1}) \\ \mathbf{h}_t &= \tanh(\mathbf{c}_t) \mathbf{o}_t \\ &= \tanh(\mathbf{c}_t) \sigma(W_o \mathbf{x}_t + R_o \mathbf{h}_{t-1}) \\ &= \tanh(\mathbf{c}_{t-1} \sigma(W_f \mathbf{x}_t + R_f \mathbf{h}_{t-1}) + \tanh(W_c \mathbf{x}_t + R_c \mathbf{h}_{t-1}) \sigma(W_i \mathbf{x}_t + R_i \mathbf{h}_{t-1})) \\ &\quad \sigma(W_o \mathbf{x}_t + R_o \mathbf{h}_{t-1})\end{aligned}\quad (2.32)$$

以上のように、LSTM の内部構造は煩雑であるが、その構成要素は全てリカレントニューラルネットワークと同様に全結合で計算できる。以下に最終的な出力の計算についてまとめる。

$$\begin{aligned}\mathbf{c}_t &= \mathbf{c}_{t-1} \sigma(W_f \mathbf{x}_t + R_f \mathbf{h}_{t-1}) + \tanh(W_c \mathbf{x}_t + R_c \mathbf{h}_{t-1}) \sigma(W_i \mathbf{x}_t + R_i \mathbf{h}_{t-1}) \\ \mathbf{h}_t &= \tanh(\mathbf{c}_{t-1} \sigma(W_f \mathbf{x}_t + R_f \mathbf{h}_{t-1}) + \tanh(W_c \mathbf{x}_t + R_c \mathbf{h}_{t-1}) \sigma(W_i \mathbf{x}_t + R_i \mathbf{h}_{t-1})) \\ &\quad \sigma(W_o \mathbf{x}_t + R_o \mathbf{h}_{t-1})\end{aligned}\quad (2.33)$$

ここで、隠れ状態(出力)  $\mathbf{h}_t, \mathbf{c}_t$  の計算は全て入力  $\mathbf{h}_{t-1}, \mathbf{c}_{t-1}, \mathbf{x}_t$  によって計算できている。また、学習可能な重み行列は  $W_f, W_i, W_c, W_o, R_f, R_i, R_c, R_o$  の八つである。一般に、適宜バイアス  $b$  が加えられる。

リカレントニューラルネットワークや LSTM は更に重ねられる(Stacked)という非常に強力な性質を持っている。そのようなネットワークを図 2.13 に示す。図からわかるように、一段目の LSTM の出力が二段目の LSTM の入力になっている。一方で、セルは両者で共有されず、独立した状態を保持している。それら以外の基本的な構造は一段であった時の LSTM と変わっていない。このように、LSTM は系列としての深さだけでなく、フィードフォワードニューラルネットワークと同様に重ねることによる深さの確保が可能である。勿論この重ねる操作は二段以上への拡張が可能であり、その場合は二段目の出力を三段目の入力に使うことによって実現できる。どの程度重ねるかはハイパーパラメータである。

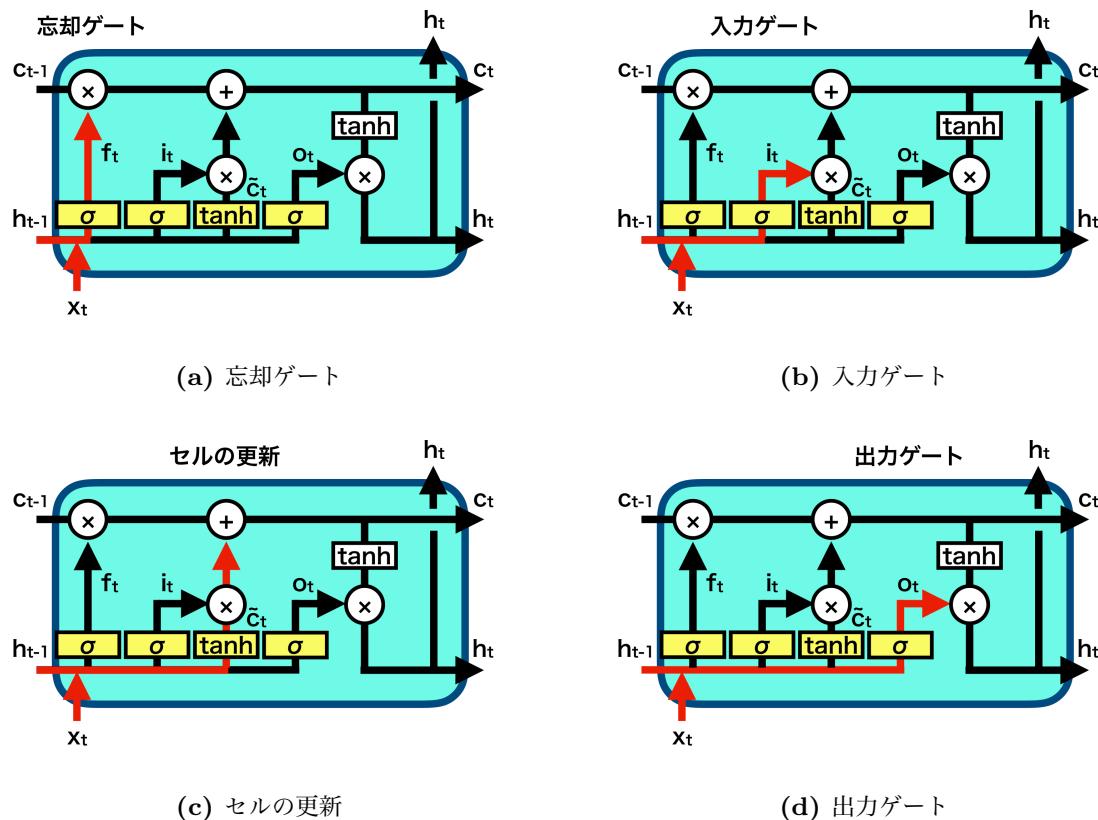


図 2.12: LSTM の各ゲートについての図解。それぞれ赤線部に沿って情報が伝達される。

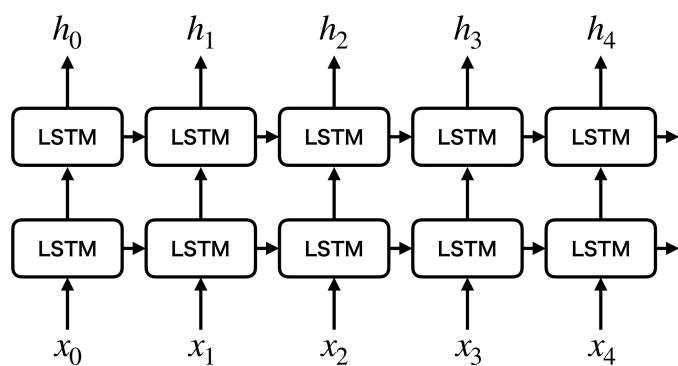


図 2.13: Stacked LSTM。ここでは LSTM の二つの隠れ状態を一つの線で表現している。下方を一段目、上方を二段目と呼ぶと、一段目の入力  $x$  と比べ、一段目の出力・二段目の入力はより抽象度の高い情報となっている。

更に、双方向 (Bidirectional) LSTM と呼ばれるネットワークについても解説する。これは、図 2.14 のように、片側を順方向に、もう一方を逆方向に系列を読み込むことにより、自然言語処理などに見られる、将来的な系列情報への依存を導入することができる。ここで、前述の LSTM を重ねる手法と異なり、それぞれの LSTM の入力はそれぞれ独立しており、前段の出力を使用していないことに注意が必要である。また、この双方向 LSTM の構造上の性質として、系列データを全て持つておき必要があるという点にも留意しなければならない。したがって、リアルタイムな問題については、未来の情報を得ることができない為、この双方向 LSTM を用いることはできない。また、この双方向 LSTM を重ねることも可能である。

リカレントニューラルネットワークはこのように次々と拡張され、より複雑で難解な系列情報の処理について、高い性能を発揮している。

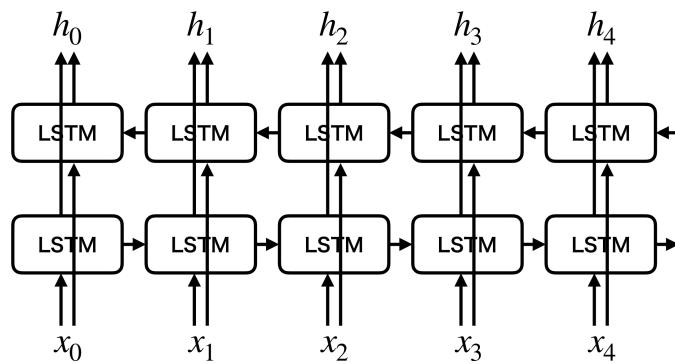


図 2.14: 双方向 LSTM。Stacked LSTM とは異なり、一段目の LSTM の出力は二段目の入力に使用されておらず、計算はそれぞれ並列的に行われる。よって出力は LSTM 二つ分の大きさになっている。

リカレントニューラルネットワークはその構造が再帰的であるという点（並列化が困難である）から、学習が遅く、重いという課題を抱えている。また、後述するエンコーダー・デコーダーモデルにおいては、データの系列の長さに応じた情報を確保できないという欠点が存在している。<sup>\*8</sup>次節では、このような問題を解決するための注意機構 (Attention) という技術を解説する。

<sup>\*8</sup> 例えば、機械翻訳を用いて 100 単語分の英文を日本語に翻訳する場合と 10 単語分の英文を翻訳する場合において、100 単語分の英文が持つ情報の方が 10 単語分の英文と比べて多いことは明らかであるが、リカレントニューラルネットワークはこれらの情報の多さの違いに対応できず、常に同じ量の情報から日本語を生成してしまう。

## 2.5 Attention

Attention [33, 34] とはエンコーダー・デコーダーモデルに使用され、デコーダー部のある系列がエンコーダー部のどこに注意するかを計算する機構である。主に機械翻訳や意味理解などに使用されており、様々な応用が議論されているが、ここでは基本的な Attention の理論と考え方についてのみ解説する。

2.5.1 項では、まず Attention を理解する上で必要不可欠なエンコーダー・デコーダーモデルについて述べ、その中で前節のリカレントニューラルネットワークが抱える問題について紹介する。次に、2.5.2 項で Attention の理論や計算について述べる。

### 2.5.1 エンコーダー・デコーダーモデル

LSTM を用いたエンコーダー・デコーダーモデルの大まかな構成を図 2.15 に示す。LSTM ではエンコーダー部とデコーダー部を繋ぐ情報はエンコーダーの最後の層の出力を使用することが多い。この場合、エンコーダー部は図 2.9 の Many to One、デコーダー部は One to Many を使用しており、エンコーダーによって抽出された情報はこの One の部分に集められることになる。この出力は Many to One の内、Many であるデータの系列の長さ（系列長）に依存せず、常に同じ大きさとなる。これは長い系列長であっても、短い系列長であっても同じ量の情報を使用しているということを意味している。したがって、より長い系列長や短い系列長を扱う場合は、そのデータが持つ情報を完全に保持、表現することは難しく、情報の欠損により性能が下がってしまうという問題があった。例えば、機械翻訳において英文の和訳を考えると、エンコーダー部は英文を受け取り、デコーダー部は日本語の文章を出力する。この時、長い英文の翻訳にはより長い日本語の文章が必要な事は明らかであるが、LSTM のみを用いたエンコーダー・デコーダーモデルではこの文の長さの違いに対応する事ができない。

この問題を解決するための技術が Attention である。Attention を組み込んだ LSTM のエンコーダー・デコーダーモデルを図 2.16 に示す。Attention を用いないエンコーダー・デコーダーモデルとの大きな違いは、エンコーダー部が Many to One から Many to Many の構造に変更されている点である。このようにエンコーダー部の構造を変える事によって、エンコーダー部は系列長（Many）に依存した情報量を確保することができる。

Attention を用いたエンコーダー・デコーダーモデルではデコーダー部の構造にも変更が加えられている。具体的には、デコーダー部はデコーダー部の“ある”系列がエンコーダー部の“どの”系列に注目するかを計算する事によって、エンコーダー部から必要な情報を抽出している。どの系列に注意するかについての度合いを Attention Weight といい、Attention Weight はエンコーダー部の全ての系列についての重みである為、実際にはエンコーダー部の系列長だけ次元を持ったベクトルである。<sup>\*9</sup>

---

<sup>\*9</sup> Attention Weight はデコーダーの個々の系列について個別に計算される。

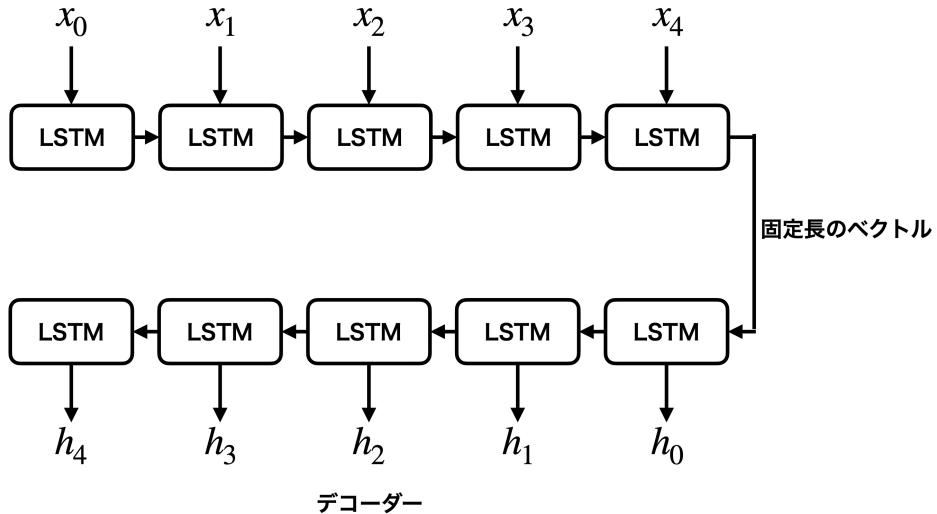


図 2.15: LSTM によるエンコーダー・デコーダーモデル。図上部がエンコーダー部、図下部がデコーダー部である。エンコーダー部の入力  $x$  から抽出された情報は図右の固定長のベクトルへと集約される。デコーダー部ではこの固定長のベクトルを初期状態にして出力  $h$  の生成を行う。

現在, Attention Weight を計算する方法が幾つか存在し, 代表的な計算方法について次項で述べる。

## 2.5.2 Attention

Attention では一般にエンコーダー部の出力を Key, Value という。また, デコーダー部のある系列  $t$  について Attention Weight を計算する際, 直前の系列  $t - 1$  のデコーダーの隠れ状態 ( $c_{De, t-1}$ ) を Query という(図 2.17)。ここで Attention Weight とはデコーダー部の個々の系列について個別に計算される量である為, Query についても同様にデコーダー部の系列の数だけ存在する事に注意する。一方 Key や Value はデコーダー部の全ての系列について共通である。本項では, デコーダー部のある系列  $t$  についての Attention に関する量を「 $t$  番目の」と呼ぶこととすると, デコーダーの隠れ状態 ( $c_{De, t-1}$ ) は  $t$  番目の Query となり, この  $t$  番目の Query によって計算される Attention Weight は  $t$  番目の Attention Weight となる。

一般に, Key, Value について  $t$  番目の Query を用いて,  $t$  番目の Attention Weight を計算する方法として, Additive Attention [33], Dot-Product Attention [34] のどちらかの手法を用いることが多い。<sup>\*10</sup> それぞれの計算を図 2.18 に示す。どちらの手法においても  $t$  番目の Attention Weight は  $t$  番目のエネルギーをソフトマックス関数によって規格化したものとなっている。<sup>\*11</sup>  $t$  番目のエネルギーを計算する手法はそれぞれ異なり, Additive Attention では学習可能な重みを用いた全結合層によって値を抽出し, Dot-Product Attention では単に Key

<sup>\*10</sup> どちらの手法も本質的な意味合いは同じであり, 後述する処理の違いなどの差しかない。

<sup>\*11</sup> このエネルギーとは物理学のエネルギーとは全く異なるものであるが, ここでは原著論文に従ってエネルギーと呼ぶこととする。

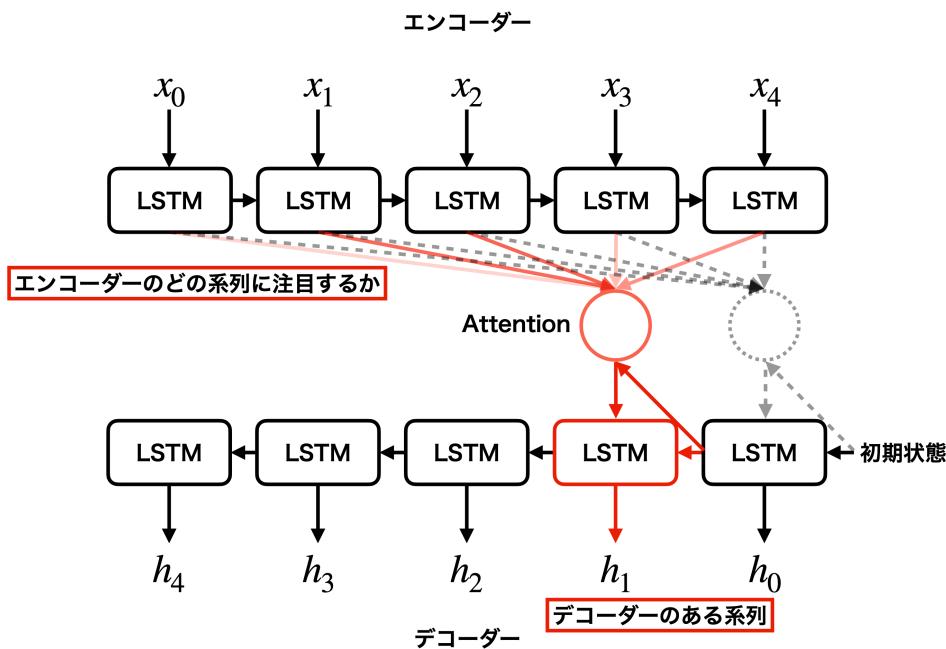


図 2.16: Attention と LSTM によるエンコーダー・デコーダーモデル。図上部がエンコーダー部、図下部がデコーダー部である。エンコーダー部は Many to Many の出力になっており、デコーダー部はデコーダー部の“ある”系列がエンコーダー部の“どの”系列に注目するかを計算する事によって、エンコーダー部から必要な情報を抽出している。

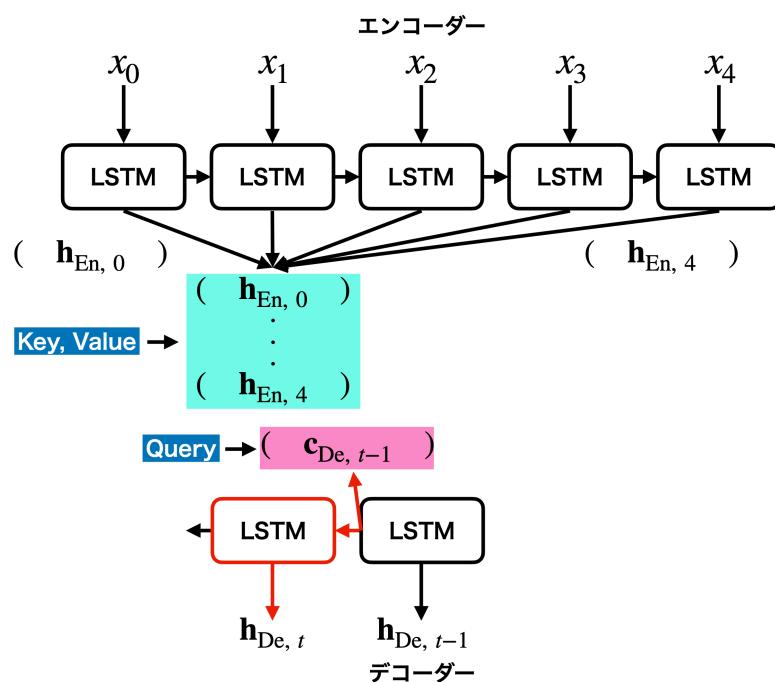
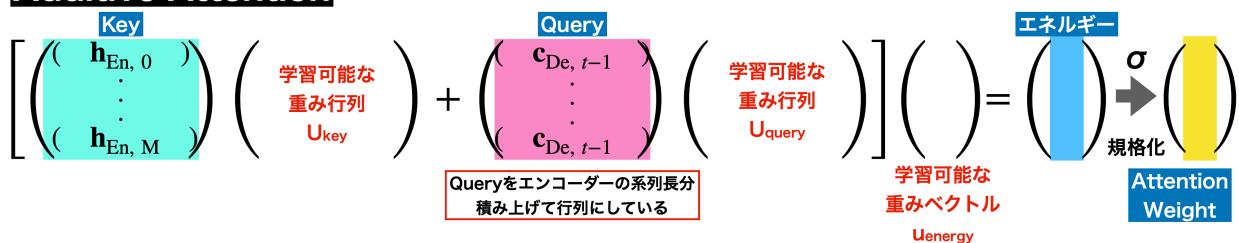


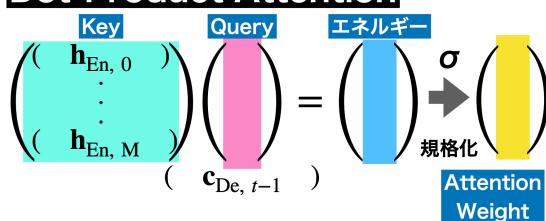
図 2.17: Key, Value, Query の図示

と Query を掛け合わせることで計算している。この為, Additive Attention では Attention の内部で重みの更新・学習が発生し, Dot-Product Attention と比較して動作が遅くなるという欠点が存在する。一方, Dot-Product Attention は Key と Query を直接掛け合わせる為, 行列演算の制約から Key と Query のそれぞれの大きさについて制限がある。ここで, エンコーダー部の系列長を M とすると t 番目の Attention Weight の大きさが M 次元のベクトルとなっていることがわかる。Attention では得られた t 番目の Attention Weight と Value を用いてデコーダー部の為の情報 (t 番目のコンテキスト) が計算される。t 番目のコンテキストをどの様にデコーダー部に使用するかに関しては 3.4.1 項で述べる。

### Additive Attention



### Dot-Product Attention



### コンテキストの計算

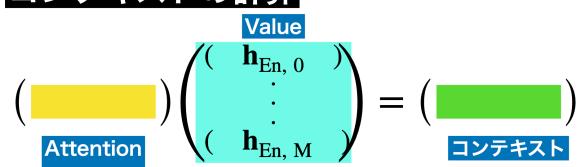


図 2.18: Additive Attention と Dot-Product Attention

Attention は Attention Weight を確認する事によってネットワーク内部の理解につながるという点でも非常に強力である。本研究における Attention Weight に関する具体的な図は図 3.25 に示している。Attention は “Attention is all you need [35]” と言われるほど, 近年非常に注目されている技術である。ここでは, Attention を LSTM の補助として使用している例を挙げたが, Attention は様々な技術と組み合わせられる。またそれだけでなく, Attention のみを用いて構成された, Transformer と呼ばれるモデルは 2021 年現在において, 自然言語処理の標準的なネットワークであると言われるほどの性能と, LSTM では実現できなかった速さを両立している。

## 2.6 ハイパーパラメータ

ここまで述べたように深層学習は教師あり学習であるため、訓練データを用いて重みの更新を行い、ネットワークの重みを最適化していく。しかし、ネットワークはネットワーク自身を構築するための、学習によって更新されないパラメータをいくつか持っている。このようなパラメータをハイパーパラメータという。ハイパーパラメータは学習前に設定しておく必要があり、ネットワークの性能を引き出すためには適切な最適化（ハイパーパラメータ・チューニング）が必要がある。ハイパーパラメータの種類や数はネットワークの構造によって大きく異なるが、一般にチューニングが必要とされるハイパーパラメータについて以下に示す。

- 最適化手法 (Optimizer) : RMSProp や Adam といった重み更新の為の最適化手法
- 学習率 (Learning rate) : 重み更新のステップ幅
- エポック数 (Epochs) : 学習回数・訓練データを一周学習することを 1 エポックという
- バッチサイズ (Batch size) : ミニバッチ学習における訓練データサンプルの大きさ
- ノード数 (Node) : 重み行列の大きさ
- 層の数 (Layer) : フィードフォワードネットワークにおける全結合層の数

これらはハイパーパラメータの一例であり、それぞれについて適切に選ぶ必要がある。ハイパーパラメータの探索手法も幾つか提案されており、ランダムサーチやグリッドサーチ、ベイズ最適化などが用いられている。

以上が本論文のための深層学習の導入である。この 2 章を前提として、3 章での本研究で使用したネットワークの構造を解説していく。以降の章では、ここで挙げた深層学習の用語を説明せずに使用するが、その場合は適宜、本章を参照していただきたい。また、本章の初めや 2.3 でも述べたように、深層学習の実装に関しては様々なフレームワークがあるため、ここでの記載は省かせていただく、本研究の実装に関しては付録 A や私の GitHub [36] にまとめている。

## 第 3 章

# 崩壊点検出の為のネットワーク

本章では、深層学習を用いた崩壊点検出のために構築したネットワークの詳細について述べる。

まず 3.1 節では、本研究で取り扱う事象の特性について述べる。前述したように本研究で使用する事象は全て MC シミュレーションデータである。3.1 節ではそのようなシミュレーションデータに関してソフトウェアの性質や物理的な性質について解説する。ただし、ネットワークの学習に使用した訓練データについては個々のネットワークの解説で適宜述べる。

次に、3.2 節では、深層学習を使用して、どのように崩壊点検出を実現するかについて、発想と構築したネットワークの役割について概説する。また、使用したハードウェアやソフトウェア・フレームワークの環境などについてもここで述べる。

3.3 節と 3.4 節では、本研究のために構築した個々のネットワークについて、構造・学習・性能と評価について議論する。

### 3.1 事象

本研究で使用したシミュレーションデータについて、3.1.1 項では事象全体についてのソフトウェアや物理的性質について述べる。3.1.2 項では本研究で使用する飛跡についての情報と深層学習に使用するための前処理について紹介する。

#### 3.1.1 事象全体の性質

本研究ではシミュレーションフレームワークとして、WHIZARD[37] による事象生成および、ILD 検出器シミュレーションを使用した。WHIZARD とはツリーレベルの行列要素を用いて、多粒子の散乱弾面積を計算する MC イベントジェネレータである (1.5.1 項)。これらにより生成されたデータは後述する LCFIPlus での性能評価 [9] で使用されたものと同一である。MC シミュレーションデータの性質を表 3.1 に示す。

表 3.1: MC シミュレーションデータの性質

イベントジェネレーター	WHIZARD
検出器	ILD 検出器シミュレーション
重心系エネルギー	Z 粒子の質量 (91.2 GeV)
終状態	$e^+e^- \rightarrow b\bar{b}, c\bar{c}, q\bar{q}$
Beamstrahlung/ISR	なし
ビーム偏極	なし

ここで、終状態の  $b$  はボトム・フレーバー,  $c$  はチャーム・フレーバー,  $q$  はアップ, ダウン, ストレング・フレーバーのクォークをそれぞれ表している。これらのクォーク対は自然界で直ちにハドロンを形成し、特にボトム, チャーム・フレーバーの場合は再構成可能な secondary vertex を残すジェットとなる。

データについて、終状態  $b\bar{b}$  のデータは 15 個のサンプルに、終状態  $c\bar{c}$  のデータは 13 個のサンプルに分け使用した（表 3.2）。終状態  $q\bar{q}$  のデータは 12 個のサンプルに分離したが、深層学習ネットワークの学習には使用していない。深層学習を含めた教師あり学習では、健全性のため学習に使用する訓練データと学習時の性能観測に使用する検証データ、最終的な評価に使用するテストデータは適切に分離しなければならない。この為、表 3.2 では個々のサンプルについて、それぞれの用途を明らかにした。

後述する LCFIPlus との比較において LCFIPlus のフレーバータギング (BDTs) の学習が必要である。その為、BDTs の訓練データ作成に  $c\bar{c} - 01, 02, 09, 10, 11, b\bar{b} - 01, 02, 03, 10, 11, 12$  を用いた。また、 $c\bar{c} - 01, b\bar{b} - 01$  は本節のデータ特性の調査に使用し、 $c\bar{c} - 02, b\bar{b} - 02, 03$  はネットワークの動作確認に使用した。ネットワークの訓練データの作成は  $c\bar{c} - 03, 04, 05, b\bar{b} - 04, 05, 06$  を用いて行なった。また、訓練データの正解ラベルは MC 情報や LCFIPlus のフィッターからの出力情報を使用した。

崩壊点検出を行うに当たって、終状態による崩壊点の性質の違いに注意しなければならない。終状態  $c\bar{c}$  の場合はチャーム・フレーバーのハドロンによる secondary vertex のみが生じるが、一方で終状態  $b\bar{b}$  の場合は  $b \rightarrow c$  という崩壊過程を辿り、ボトム・フレーバーのハドロンによる secondary vertex と更にそこから派生したチャーム・フレーバーのハドロンによる tertiary vertex が生じる。それぞれのハドロン粒子の典型的な飛程は寿命  $\tau$  と光速  $c$  を用いて、ボトム・フレーバーの場合は  $c\tau = 400 - 500 \mu\text{m}$ 、チャーム・フレーバーの場合は  $c\tau = 20 - 300 \mu\text{m}$  となる（図 3.1）。

また、これら以外の崩壊点として  $V^0$  粒子の崩壊 ( $K_S^0 \rightarrow \pi^+\pi^-$ ,  $\Lambda^0 \rightarrow p\pi^-$ ) や光子変換 ( $\gamma_{conv}X \rightarrow e^+e^-X$ ) を考えることができる。これらの崩壊点は secondary vertex や tertiary vertex と比較して、衝突点から遠い位置で生じる。そのような崩壊点とその崩壊点由来の飛跡を以後 others と呼ぶ。

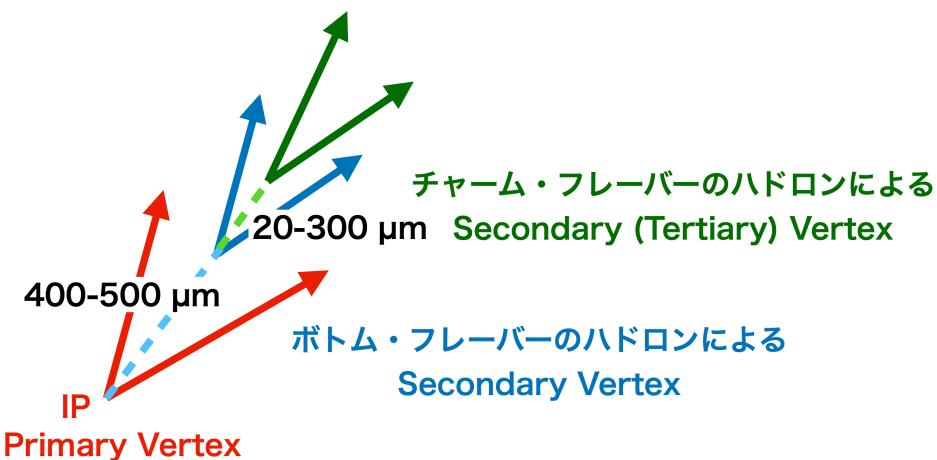


図 3.1: 終状態  $b\bar{b}$  での崩壊点の例。赤線: primary vertex 由来の飛跡。青線:secondary vertex 由来の飛跡。緑線:tertiary vertex 由来の飛跡。

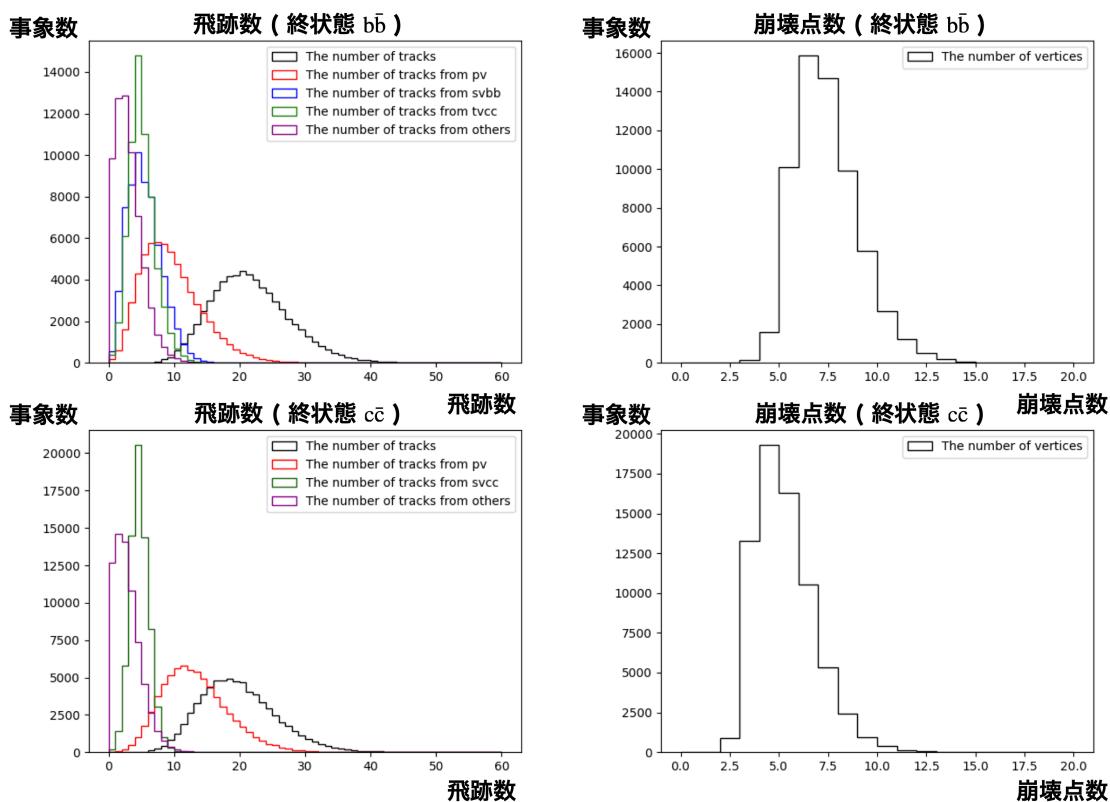


図 3.2: 事象に含まれる飛跡の本数と崩壊点の個数。図左は飛跡の本数、図右は崩壊点の個数である。飛跡の本数において、黒線は事象に含まれる全飛跡数、赤線は primary vertex 由来の飛跡数、青線はボトム・フレーバーの secondary vertex 由来の飛跡数、緑線はチャーム・フレーバーの secondary vertex 由来の飛跡数、紫線は others の飛跡数を表している。

表 3.2: データサンプルの事象数と用途

データ名	事象数	飛跡数	用途
c̄c - 01	69581	1344465	データの調査/フレーバータギングの訓練データの作成
c̄c - 02	42204	814074	動作テスト/フレーバータギングの訓練データの作成
c̄c - 03	38662	748027	飛跡対についてのネットワークの訓練データの作成
c̄c - 04	38712	747625	飛跡対についてのネットワークの訓練データの作成
c̄c - 05	38655	748089	任意の数についてのネットワークの訓練データの作成
c̄c - 06	38645	747548	ネットワーク/崩壊点検出の評価
c̄c - 07	38643	747312	ネットワーク/崩壊点検出の評価
c̄c - 08	38715	748801	ネットワーク/崩壊点検出の評価
c̄c - 09	38705	747725	フレーバータギングの訓練データの作成
c̄c - 10	38721	748025	フレーバータギングの訓練データの作成
c̄c - 11	38587	747819	フレーバータギングの訓練データの作成
c̄c - 12	38723	748904	LCFIPlusとの比較
c̄c - 13	35848	693780	LCFIPlusとの比較
bb̄ - 01	62795	1326168	データの調査/フレーバータギングの訓練データの作成
bb̄ - 02	42950	909082	動作テスト/フレーバータギングの訓練データの作成
bb̄ - 03	34985	738105	動作テスト/フレーバータギングの訓練データの作成
bb̄ - 04	34952	739130	飛跡対についてのネットワークの訓練データの作成
bb̄ - 05	35047	741568	飛跡対についてのネットワークの訓練データの作成
bb̄ - 06	35008	740662	任意の数についてのネットワークの訓練データの作成
bb̄ - 07	34000	718057	ネットワーク/崩壊点検出の評価
bb̄ - 08	33978	717972	ネットワーク/崩壊点検出の評価
bb̄ - 09	35008	740268	ネットワーク/崩壊点検出の評価
bb̄ - 10	34954	739320	フレーバータギングの訓練データの作成
bb̄ - 11	35012	740797	フレーバータギングの訓練データの作成
bb̄ - 12	34972	739953	フレーバータギングの訓練データの作成
bb̄ - 13	34986	739402	LCFIPlusとの比較
bb̄ - 14	34910	740933	LCFIPlusとの比較
bb̄ - 15	10243	216499	LCFIPlusとの比較

図 3.2 は一つの事象に含まれる飛跡の本数と崩壊点の個数である。これらの粒子識別は MC 情報を用いた。ここでは飛跡の本数は親粒子のフレーバーのみ考慮し、同フレーバーの親粒子(クォーク・反クォーク)の違いは区別していない。したがって secondary vertex については、一つの崩壊点ではなく  $b$  と  $\bar{b}$  などの複数の崩壊点の飛跡が含まれている。 $b$  と  $\bar{b}$  による崩壊点を合わせた場合の secondary vertex の飛跡の数は 5 本程度である。したがって個々の  $b$  と  $\bar{b}$  によって生じた secondary vertex に含まれる飛跡の数は 2 – 3 本程度と考えられる。崩壊点検出ではこの個々の secondary vertex を見分ける必要がある。崩壊点の個数では親粒子を全て区別している。典型的な崩壊点の個数は終状態が  $c\bar{c}$  の場合は primary vertex, チャーム・フレーバーの secondary vertex が 2 つ, others の 3 – 5 個である。終状態が  $b\bar{b}$  の場合

は primary vertex, ポトム・フレーバーの secondary vertex が 2 つ, チャーム・フレーバーの tertiary vertex が 2 つ, others の 5 – 7 個である。

### 3.1.2 飛跡の情報と前処理

飛跡一本分の情報として, 位置や運動量の情報を含んだ飛跡パラメータ 5 個 ( $d_0, z_0, \phi, \Omega, \tan \lambda$ )[38] とその共分散行列 15 個, 電荷, エネルギーの計 22 個の変数を使用した。加速器の座標系としてビーム衝突点を原点とし,  $z$  軸をビーム方向にとった座標系を使用する。また, 深層学習の学習に使用する変数は, 一般に  $[-1, 1]$  の範囲に整形した方が良いと言われている為, 変数をそれぞれ以下のような  $\tanh$  関数や線形関数などを用いて整形した。

- 飛跡パラメータ

$$d_0 = \tanh(d_0), z_0 = \tanh(z_0), \phi = \phi/\pi, \Omega = \tanh(200\ \Omega), \tan \lambda = \tanh(0.3 \tan \lambda)$$

- 飛跡パラメータの共分散行列

$$\tanh(8000(x - 0.0005))$$

- エネルギー

$$\tanh(0.5(x - 5.0))$$

更に LCFIPlus のフィッティングで得られる崩壊点の  $\chi^2$  や崩壊点の位置についてもデータを用意した。ここでは, 事象中の二本の飛跡 (飛跡対) の全ての組み合わせについて計算を行った。ただし, このような高次の情報を含んだ変数は基本的にはネットワークの学習に使用せず, 正解ラベルの一つとして使用した。LCFIPlus によって予想される崩壊点の位置について, ビーム衝突点からの距離を図 3.3 に示す。図 3.3 のグラフは両対数グラフとして表現している為, 衝突点からの距離  $10^{-2}$  mm から  $10^3$  mm のプロットとなっており衝突点は含んでいない。primary vertex と secondary vertex の衝突点からの距離は大きく異なっているが, ポトム・フレーバーとチャーム・フレーバーの secondary vertex の衝突点からの距離の分布に大きな違いはなく, ピークの位置の差は 1 mm 程度である。また, 終状態  $c\bar{c}$  と終状態  $b\bar{b}$  では  $b \rightarrow c$  の崩壊過程を辿るか否かの違いがある為, チャーム・フレーバーの崩壊点の位置が少し異なっている。others は衝突点付近では殆ど起こらず, 他の崩壊点と比較して遠い位置で崩壊している。このように崩壊点検出において位置の再構成は非常に重要な情報を含んでおり, 逆説的に位置の再構成が出来なければ崩壊点に分離は困難であると言える。

1 mm 付近のピークは LCFIPlus のフィッティングが失敗している飛跡対である。フィッティング健全性を表す  $\chi^2$  との相関を見ると, 図 3.4 のように, 1 mm 付近のデータは大きな  $\chi^2$  値を持っていることが分かる。しかし, 本研究ではフィッティングに失敗した飛跡対が何故 1 mm 付近にピークを持つのかの理解には至らなかった。

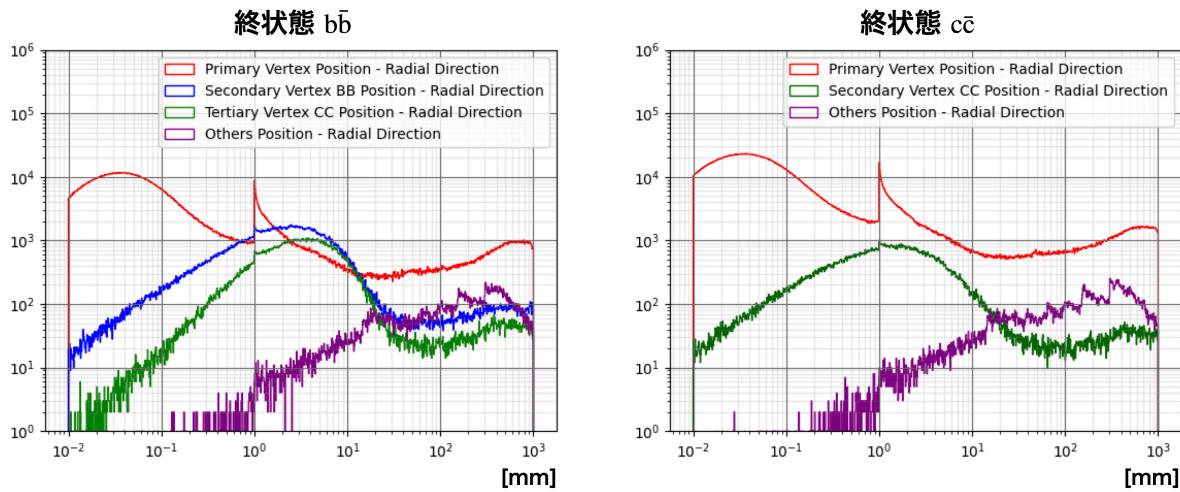


図 3.3: LCFIPlus によって予想される崩壊点の位置の分布。横軸は対数スケールの衝突点からの距離である。赤線は primary vertex の位置, 青線はボトム・フレーバーの secondary vertex の位置, 緑線はチャーム・フレーバーの secondary vertex の位置, 紫線は others の位置を表している。これらは飛跡対についての LCFIPlus の計算値である。

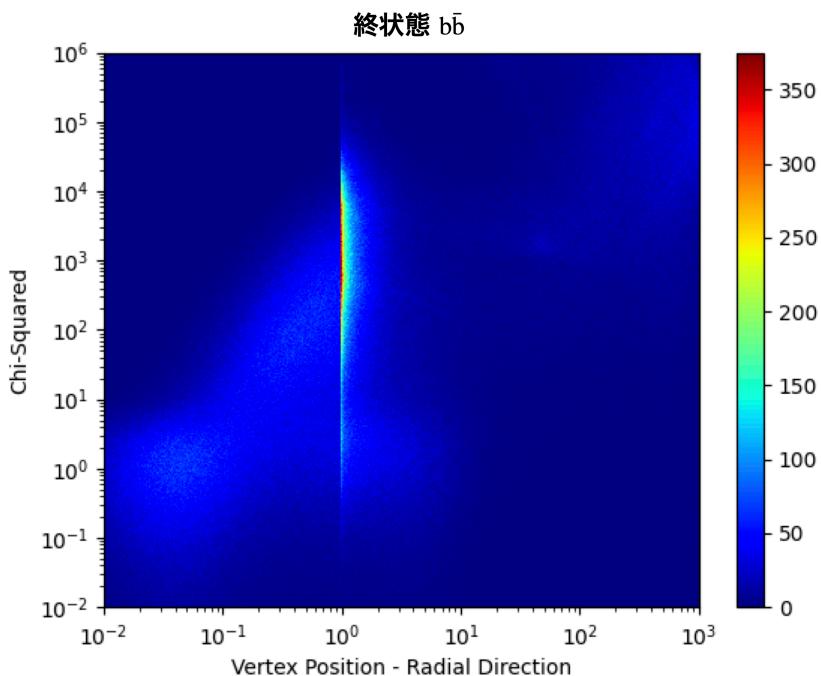


図 3.4: LCFIPlus によって予想される崩壊点の位置と  $\chi^2$  値の相関。縦軸、横軸はそれぞれログスケールの  $\chi^2$  値、ログスケールの衝突点からの距離である。カラースケールはカウントを示しており、1 mm 付近で非常に大きな値になっていることがわかる。

## 3.2 深層学習を用いた崩壊点検出の実現

深層学習は分類問題や回帰問題において強力なツールであるが、一方で基本的には教師あり学習である為、クラスタリングなどに対しては向きである。教師あり学習では訓練データから「パターン」を学び、データの持つ特徴量の空間内である種の境界を引く必要がある。この「パターン」はあらゆるデータ内で分類可能な決まった性質を持っていなければならない。例えば3.1.1項では、同一フレーバーの secondary vertex が主に二つあることを解説した。この二つの secondary vertex は事象内では位置の違いによって区別できるが、あらゆる事象間で不变的に一番や二番といったラベル付けできる性質を持っていない。<sup>\*1</sup>

崩壊点検出アルゴリズムの目的は複数の崩壊点を探索する事である。このような問題は一般にクラスタリングを用いて解くことが多い。しかし本研究で使用するデータは図3.2で示したように事象内に含まれる飛跡の本数や崩壊点の個数も異なっているという性質を持っている。これはクラスター数やクラスターに含まれる要素数が常に変わってしまうということを意味しており、崩壊点検出をクラスタリングで行うことは不適であると判断した。

以上を踏まえた上で私は次の二つのネットワークを用いた崩壊点検出を提案する。

### 1. 飛跡対についてのネットワーク

- 用途：崩壊点のタネの探索
- 入力：事象内のあらゆる飛跡対（二本の飛跡の全ての組み合わせ）
- 出力：飛跡対の属する崩壊点の種類・崩壊点の位置（衝突点からの距離）

### 2. 任意の数の飛跡についてのネットワーク

- 用途：崩壊点の生成
- 入力：崩壊点のタネ・事象内の全ての飛跡
- 出力：事象内のそれぞれの飛跡が崩壊点のタネに結合しているか否か

飛跡対についてのネットワークは崩壊点のタネとなる飛跡対を探索するネットワークである。したがって、このネットワーク単体では崩壊点を形成することはできない。そこで、私は崩壊点の生成を行う、もう一つのネットワークを構築した。崩壊点の生成を行うネットワークは二本以上の飛跡を取り扱う必要があるが、三本、四本、五本の飛跡を取り扱えるネットワークをそれぞれ構築することはネットワークや飛跡の組み合わせの数を考える上で適切ではない。よって不定の数の飛跡を再帰的に処理するネットワーク構造として、リカレントニューラルネットワークを採用した。任意の数の飛跡についてのネットワークは崩壊点のタネをリカレントニューラルネットワークの初期状態として、そこに飛跡を一本ずつ加え崩壊点の生成を行うネットワークである。

---

<sup>\*1</sup> 実際には損失関数を最小にするような順序を与えることで分離することは可能であるが本研究においては様々な secondary vertex が存在する為不適である。

本研究は以上の二つのネットワークを用いることで崩壊点検出を実現した。構造や学習についての、より詳細な個々のネットワークの解説は、後の 3.3 節や 3.4 節で述べる。

本研究のネットワークは Tensorflow/Keras フレームワークを用いて構築・学習を行なった。学習に際しては計算機として、研究室が保有する GPU, “NVIDIA TITAN RTX”や九州大学情報基盤研究開発センター研究用計算機システムを使用した。詳細なソフトウェア・ハードウェアの環境を表 3.3 に示す。

表 3.3: ソフトウェア・ハードウェアの環境

ソフトウェア	
Python	3.6.8
Tensorflow	2.1.0
Keras	2.3.1
ハードウェア (研究室内のシステム)	
CPU	AMD EPYC 7402P 24-Core Processor
メモリ	263.7 GB
GPU	NVIDIA Corporation TU102 [TITAN RTX] 2 個

### 3.3 飛跡対についてのネットワーク

ここでは 3.2 節で紹介した二つのネットワークの内、飛跡対についてのネットワークに関して述べる。主にネットワークの構造に関しては 3.3.1 項で、学習に関しては 3.3.2 項で解説する。また、そのようにして構築、訓練されたネットワーク単体についての性能と評価に関しては、3.3.3 項で述べることとする。

飛跡対についてのネットワークは、崩壊点のタネを探索するためのネットワークであり、入力は二本の飛跡についての情報、出力は飛跡対についての崩壊点の種類や位置である。この崩壊点の種類を考える上で 3.1.1 項で述べた、終状態によって生じる崩壊点の種類の違いを考慮しなければならない。例えば、終状態  $b\bar{b}$  の場合は  $b \rightarrow c$  という崩壊過程を辿り、ボトム・フレーバーの secondary vertex とチャーム・フレーバーの tertiary vertex が生じ、終状態  $c\bar{c}$  の場合はチャーム・フレーバーの secondary vertex のみが生じる。また両方の終状態について、これら以外の崩壊点である others<sup>\*2</sup>を考える必要がある。更に終状態  $b\bar{b}$  の場合について、ボトム・フレーバーの secondary vertex からの飛跡とそこから生じたチャーム・フレーバーの tertiary vertex からの飛跡を一本ずつ含んだ飛跡対を準崩壊点として定義する。(図 3.5)

\*2 ストレンジ ( $s$ ) ・フレーバーのハドロンの崩壊、光子変換

以上より、飛跡対についての崩壊点の種類は“非結合な飛跡対 (not connected, NC)”, “primary vertex (PV)”, “チャーム・フレーバーの secondary vertex (SVCC)”, “ボトム・フレーバーの secondary vertex (SVBB)”, “チャーム・フレーバーの tertiary vertex (TVCC)”, “終状態  $b\bar{b}$  での準崩壊点 (SVBC)”, “これら以外の崩壊点 (Others)”の計 7つとなる(表 3.4)。

表 3.4: 飛跡対についての崩壊点の種類

クラス名	詳細
NC	非結合な飛跡対 (not connected)
PV	primary vertex
SVCC	チャーム・フレーバーの secondary vertex
SVBB	ボトム・フレーバーの secondary vertex
TVCC	チャーム・フレーバーの tertiary vertex (TVCC)
SVBC	終状態 $b\bar{b}$ での準崩壊点
Others	上記以外の崩壊点

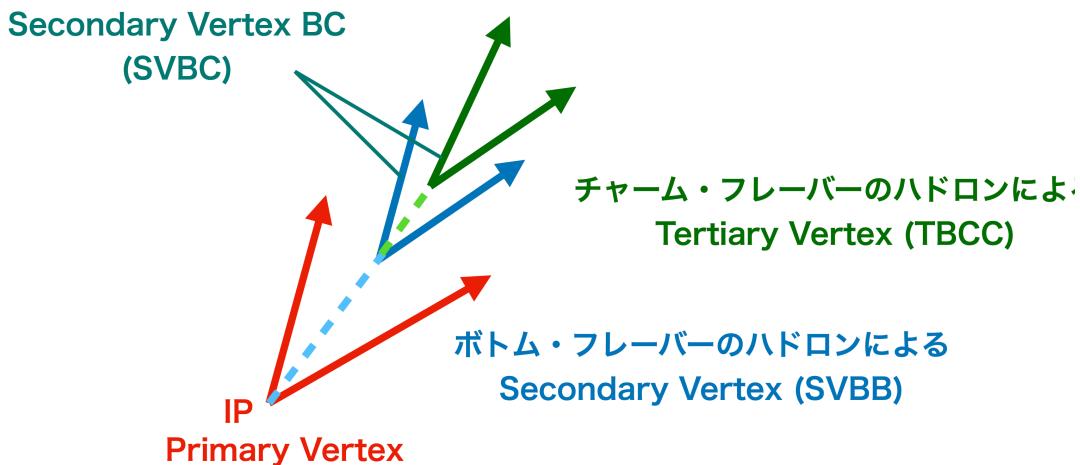


図 3.5: 終状態  $b\bar{b}$  での崩壊点

崩壊点の位置についての訓練データを作成するに当たって、正解ラベルとして図 3.3 の LCFIPlus のフィッティングで得られる計算値を用いた。<sup>\*3</sup> こちらは回帰によって衝突点からの距離を再現する。

<sup>\*3</sup> この様な計算値は本質的に正しいとは限らないが、本研究では補助的な変数として用いる事で、現行の手法を内包したネットワークの構築を目指す。

### 3.3.1 ネットワークの構造

飛跡対についてのネットワークとして非常にシンプルなフィードフォーワード構造を使用した。ネットワークの概略図を図 3.6 に示す。

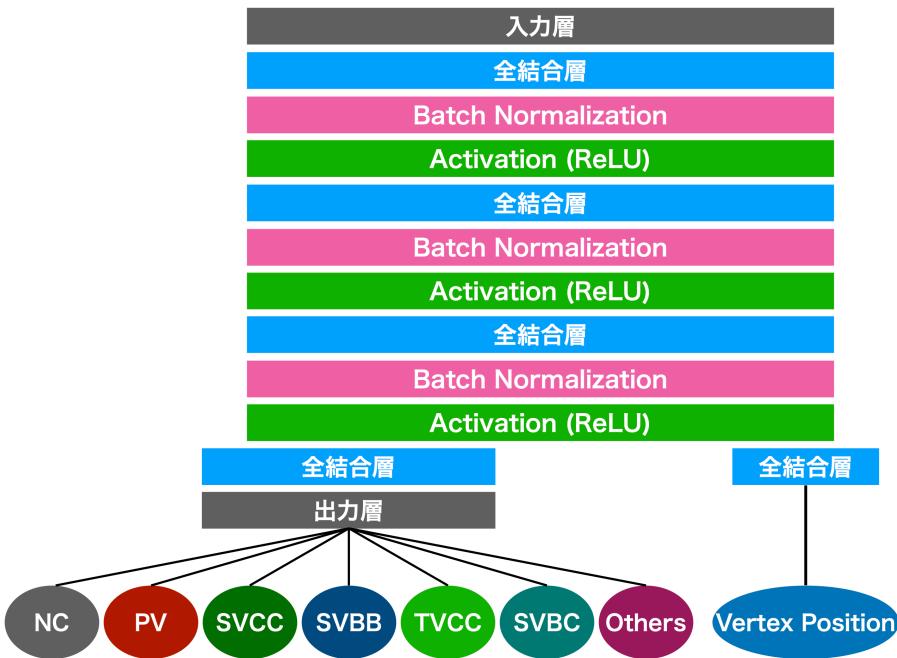


図 3.6: 飛跡対についてのネットワークの概略図。全結合層・Batch Normalization・活性化関数 (ReLU) を三回重ねている。その後、分類問題の為の全結合層と回帰問題の為の全結合層の二つに分離させ、それぞれで出力を行う。

前述したように出力は、7 クラス分類と回帰 1 つである。出力の直前の全結合層で分類問題と回帰問題に分離させた。また、過学習 (Over fitting) を避ける為、Batch Normalization[39] を全結合層の後に配置した。過学習とは、ネットワークが過度に訓練データに適合してしまい、検証データやテストデータへの汎化性能が悪化してしまう、教師あり学習の問題の一つである。また勾配消失への対策として、活性化関数は全て ReLU 関数を使用した。

### 3.3.2 ネットワークの学習と戦略

訓練データは事象中の全ての飛跡対の組み合わせを考える。よって入力変数は飛跡二本分であるので合計 44 個である。ここで次の二つの事柄に注意しなくてはならない。

- 分類クラスは二つの終状態  $b\bar{b}$  と  $c\bar{c}$  からのカテゴリー<sup>\*4</sup>を両方使用している
- 分類クラスのデータ数の比が NC や PV が支配的な不均衡データ (Imbalanced Data) となる

各終状態での分類クラスのデータ数の比を図 3.7 に示す。

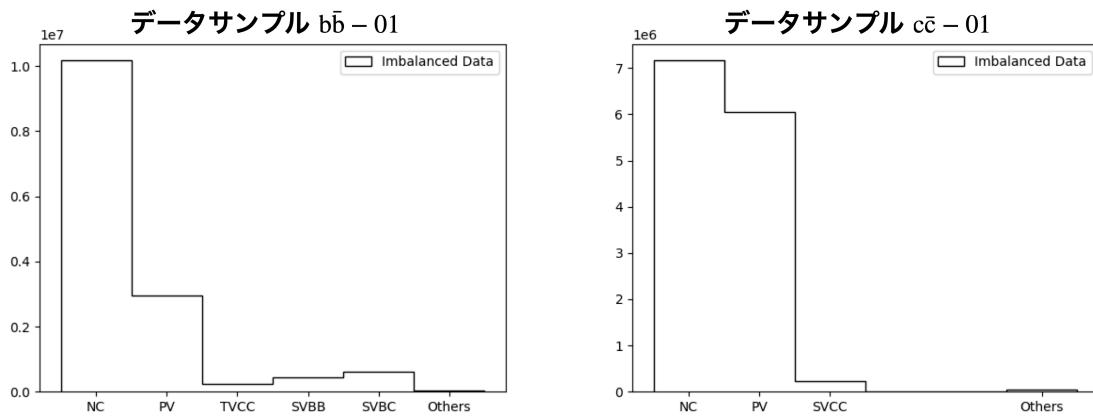


図 3.7: 各終状態での分類クラスのデータ数の比。図左が終状態  $b\bar{b}$ , 図右が終状態  $c\bar{c}$  である。終状態  $c\bar{c}$  では SVCC 以外の SV が基本的に存在しない為、終状態  $b\bar{b}$  と比較して PV の比率が多くなっている。

このような不均衡データについては、少数クラスのデータをかさ増しするオーバーサンプリング、多数クラスのデータを間引くアンダーサンプリング、損失関数のコストに重みをつけるコスト考慮型学習の主に三つの対応策が存在する。オーバーサンプリングやアンダーサンプリングは過学習や情報の欠損などの問題を抱えているため、本研究では基本的にコスト考慮型学習を用いた。ただし、二つの終状態のデータを単純に足し合わせた場合、共通する分類クラスである NC や PV がより顕著になり、他クラスの学習が不十分になると考えられる為、NC や PV に関しては各終状態毎に半分にランダムサンプリングした後、終状態  $b\bar{b}$  と  $c\bar{c}$  を足し合わせた。最終的な訓練データでの分類クラスのデータ数の比を図 3.8 に示す。

損失関数  $L$  は分類問題である崩壊点の種類に関する損失関数  $L_{CE}$  と回帰問題である崩壊点の位置に関する損失関数  $L_{LMSE}$  の二つの合計となる。

<sup>\*4</sup> 例えば NC・PV・Others は両方の終状態に共通のクラスであるが、SVBB・TVCC・SVBC は終状態  $b\bar{b}$ , SVCC は終状態  $c\bar{c}$  にしか存在しないクラスである。

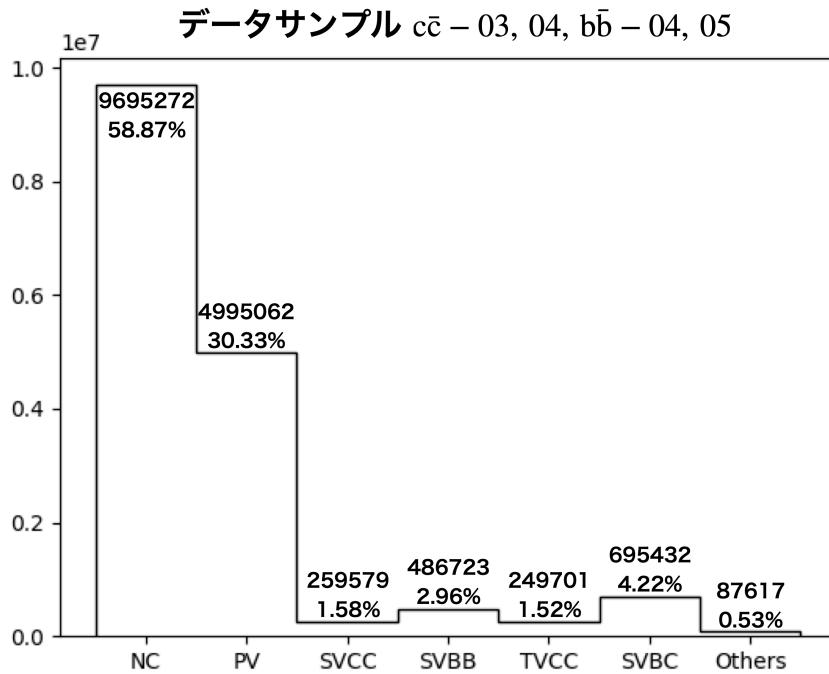


図 3.8: 訓練データでの分類クラスのデータ数の比。実際に使用する訓練データでは NC と PV のカテゴリで全体の 9 割程度となっており、非常に不均衡なデータとなっている。

本研究では、コスト考慮型学習として損失関数  $L_{CE}$  の各クラスへの重みは、分類クラスのデータ数の比の逆数を使用し不均衡データへ対策を行なった。ここでは最も数の少ない Others の重みを 1 としている。

$$\begin{aligned}
 L_{CE} = & -0.0090 t_{NC} \log(y_{NC}) - 0.0175 t_{PV} \log(y_{PV}) \\
 & - 0.3375 t_{SVCC} \log(y_{SVCC}) - 0.1800 t_{SVBB} \log(y_{SVBB}) \\
 & - 0.3509 t_{TVCC} \log(y_{TVCC}) - 0.1260 t_{SVBC} \log(y_{SVBC}) \\
 & - 1.0 t_{Others} \log(y_{Others})
 \end{aligned} \tag{3.1}$$

ここで、 $t_{NC}$ ,  $t_{PV}$ ,  $t_{SVCC}$ ,  $t_{SVBB}$ ,  $t_{TVCC}$ ,  $t_{SVBC}$ ,  $t_{Others}$  はそれぞれの分類クラスについての正解ラベルである。また、 $y_{NC}$ ,  $y_{PV}$ ,  $y_{SVCC}$ ,  $y_{SVBB}$ ,  $y_{TVCC}$ ,  $y_{SVBC}$ ,  $y_{Others}$  はそれぞれの分類クラスについてネットワークで予想されたスコアである。

崩壊点の位置についての損失関数  $L_{LMSE}$  として平均二乗誤差を使用した。ただし、図 3.3 からも分かるように、崩壊点の位置は非常に広い範囲に分布しているため、出力や正解ラベルの自然対数を使用し

$$L_{LMSE} = (\ln(t_{Position} + 1) - \ln(y_{Position} + 1))^2 \tag{3.2}$$

とした。ここで、 $t_{Position}$ ,  $y_{Position}$  はそれぞれ LCFIPlus のフィッターで予想された崩壊点の位置 (衝突点からの距離), ネットワークで予想された崩壊点の位置 (衝突点からの距離) である。

以上より、飛跡対についてのネットワークの損失関数  $L$  は

$$L = w_{\text{vertex}} L_{\text{CE}} + w_{\text{position}} L_{\text{LMSE}} \quad (3.3)$$

となる。ここで、 $w_{\text{vertex}}$ 、 $w_{\text{position}}$  は各損失関数  $L_{\text{CE}}$ 、 $L_{\text{LMSE}}$  への重みである。

本ネットワークでは崩壊点の位置を学習した後に崩壊点の種類の分類を行なった。これは崩壊点の位置、あるいはそのような情報を抽出できるネットワークのパラメータを利用して、崩壊点の種類の分類を行なって欲しいという狙いからである。これは転移学習 (Transfer Learning, TL) やファインチューニングという手法<sup>\*5</sup>に似た発想であるが、今回はこれら二つの手法とは異なり、ネットワークの重みは全て再学習に使用している。学習は以下の手順で行う。

1.  $(w_{\text{vertex}}, w_{\text{position}}) = (0.1, 0.9)$ , 1000 epoch, 学習率 LR = 0.001
2.  $(w_{\text{vertex}}, w_{\text{position}}) = (0.9, 0.1)$ , 1500 epoch, 学習率 LR = 0.001
3.  $(w_{\text{vertex}}, w_{\text{position}}) = (0.95, 0.05)$ , 500 epoch, 学習率 LR = 0.0001

また、重み更新の最適化手法として SGD を用いた。これは、Adam などでは収束が早すぎ、過学習になる恐れがあったためである。学習には 13175508 サンプル、検証には 3293878 サンプルのデータを使用した。ネットワークの学習可能な重みのパラメータ数を表 3.5 に示す。

表 3.5: 飛跡対についてのネットワークにおける訓練可能なパラメータ数

層の名称	出力の形状	パラメータ数	接続先
Pair Input	(None, 44)	0	
Dense 1	(None, 256)	11520	Pair Input
Batch Normalization 1	(None, 256)	1024	Dense 1
Activation ReLU 1	(None, 256)	0	Batch Normalization 1
Dense 2	(None, 256)	65792	Activation ReLU 1
Batch Normalization 2	(None, 256)	1024	Dense 2
Activation ReLU 2	(None, 256)	0	Batch Normalization 2
Dense 3	(None, 256)	65792	Activation ReLU 2
Batch Normalization 3	(None, 256)	1024	Dense 3
Activation ReLU 3	(None, 256)	0	Batch Normalization 3
Vertex Dense	(None, 7)	1799	Activation ReLU 3
Vertex Output	(None, 7)	0	Vertex Dense
Position Output	(None, 1)	257	Activation ReLU 3

<sup>\*5</sup> これらの手法ではあらかじめ学習済みのネットワークを別の問題解決に活用するテクニックである

### 3.3.3 ネットワークの評価

ネットワークの評価は 1. ネットワーク間の比較, 2. ネットワーク内の理解の二つの観点によって行う。本ネットワークでは, 1. ネットワーク間の比較については入力変数やネットワーク構造の異なる幾つかのネットワークのモデルについて比較を行う。2. ネットワーク内の理解については t 分布型確率的近傍埋め込み法 (t-Distributed Stochastic Neighbor Embedding, t-SNE[40]) を用いた次元削減によって, ネットワークによる情報の抽出や各クラスの分離について議論する。

#### 1. ネットワーク間の比較

深層学習を取り扱う上で気を付けなければならない問題として, ネットワークの怠け (Lazy) が存在する。ネットワークの怠けとは, ある入力変数について, 特定の要素にのみ注目する, あるいは特定の要素を無視してしまうという問題である。飛跡対についてのネットワークでは, 深層学習が取り扱いづらい飛跡パラメータの共分散 (3.1.2 項) を入力している。ここではネットワークが共分散をどの程度取り扱っているかを評価するため, 共分散を含んだ入力変数を用いたモデル A と含んでいないモデル B を構築し比較を行なった。ここでネットワークの構造として, 図 3.6 を使用し, そのようなネットワークの構造をネットワーク 1 と呼ぶことにする。

また, 3.1.2 節でも述べたように本研究における崩壊点の種類の分類において, 崩壊点の位置の再構成は必須である。したがって, そのような崩壊点の位置についての情報を再構成し, 適切に処理出来ているかを把握するため, 回帰問題の正解ラベルとして用いている LCFIPlus によって予想された崩壊点の位置についての情報を出力層の直前で入力するネットワーク 2 を構築した。更に, ネットワーク自体が予想した崩壊点の位置を明示的に入力したネットワーク 3 の構築を行った。これらネットワーク 2, ネットワーク 3 を使用したモデルをそれぞれモデル C, モデル D とし前述のモデル A, モデル B との比較を行なった。それぞれのモデルの詳細を表 3.6 にまとめる。

表 3.6: 評価のための飛跡対についてのモデル

モデル	入力変数	ネットワーク
モデル A	飛跡パラメータ, 共分散行列, 電荷, エネルギー	ネットワーク 1
モデル B	飛跡パラメータ, 電荷, エネルギー	ネットワーク 1
モデル C	飛跡パラメータ, 共分散行列, 電荷, エネルギー, 崩壊点の位置	ネットワーク 2
モデル D	飛跡パラメータ, 共分散行列, 電荷, エネルギー	ネットワーク 3

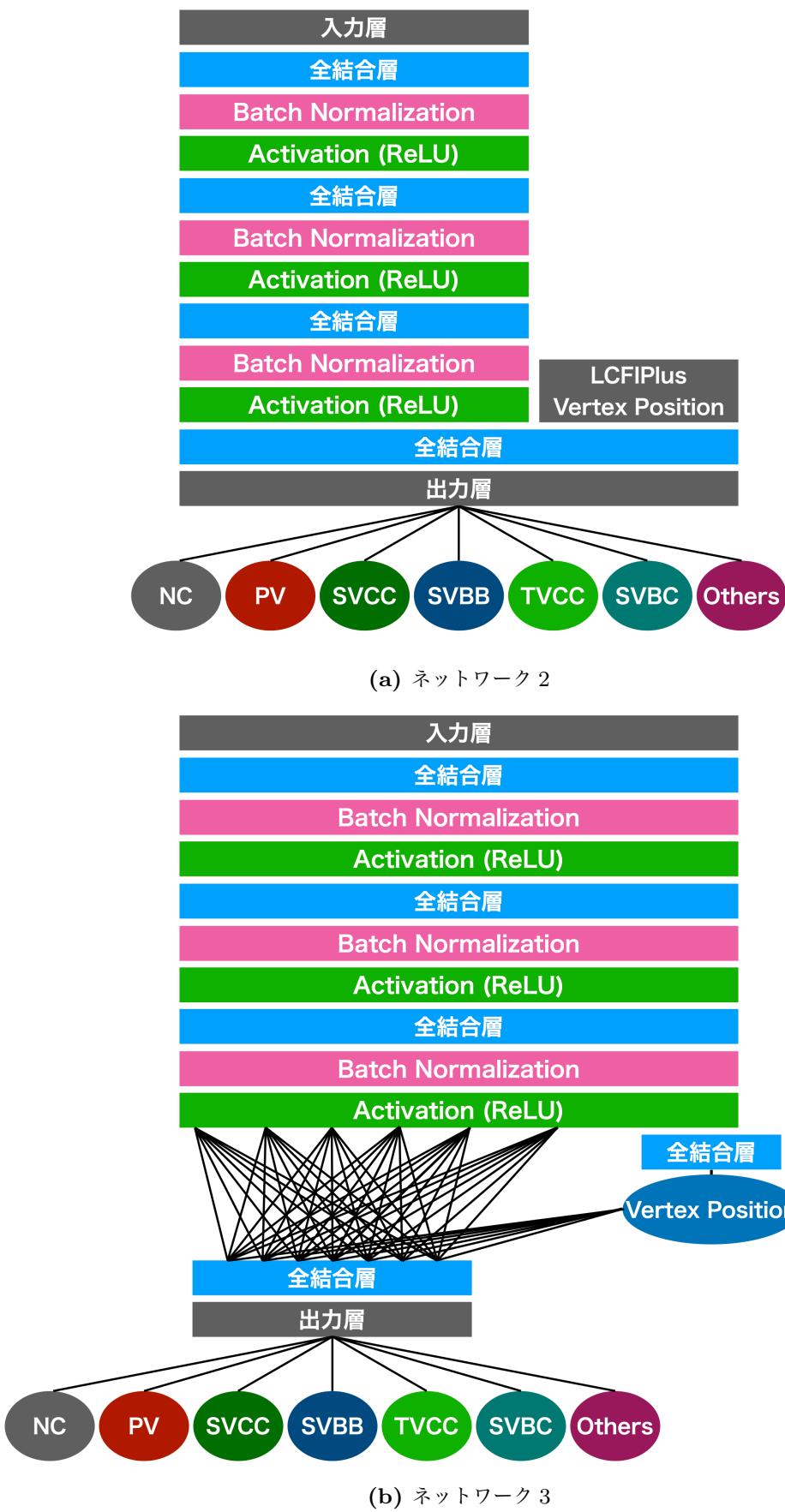


図 3.9: 評価のための飛跡対についてのネットワーク

比較は各クラス分類についての「ネットワークのスコアとその効率の変化」と「混合行列」の二つの指標を用いて行う。また、本ネットワークは7クラス分類を行っているため、ある特定のクラスについてのスコアに閾値を設け、評価することは数学的に厳密に正しくない。しかし、ここでは直感的な理解を優先し評価手法の一つとして採用している。

まず、ネットワークのスコアとクラス分類の効率の関係を図3.10に示す。図の横軸はネットワークによって得られる各クラスのスコアに設けた閾値である。また、縦軸は効率を示しており、実線で信号効率を、破線でバックグラウンド効率を表している。ここでは、信号はそれぞれの特定のクラスとし、バックグラウンドはその特定のクラス以外の全てのクラスと定義している。スコアに対して高い閾値を設けると信号・バックグラウンド共に効率は悪化し、一方で低い閾値であれば、全ての要素を信号であると判断してしまうため、両者の効率は最大となる。したがって、高い信号効率と低いバックグラウンド効率を実現できていれば、良い分類器であると判断でき、図3.10ではNC・PV・Othersについては良く分類できているが、各SVについては殆ど見分けられないことが分かる。

これら信号効率を縦軸にバックグラウンド効率を横軸に描画したものが図3.11である。このような図を受信者操作特性(Receiver Operating Characteristic, ROC)曲線といい、ROC曲線はその曲線で囲んだ面積が大きければ性能が良いと判断できる。したがって、より左側に張り出している分類器が最も良い性能となる。ここでは、横軸を対数で表現しており、よりバックグラウンド効率の低い領域において、それぞれのモデルの比較を行っている。図3.11ではどのクラスについてのモデル間の大きさ差異は見られない。ただし、NCについてのみモデルBの性能が悪いという結果を得られた。

ネットワークがどの程度の効率や純度でクラスを分類出来ているかを更に把握するため混合行列を用いる(図3.12)。混合行列は横軸をネットワークによって予想されたクラス、縦軸を正解ラベルでのクラスとしてデータを行列化したものである。したがって、対角成分が正答、それ以外は誤答である。ここでは、効率について規格化したものと純度について規格化したものの二つで評価を行う。また、比較のためモデルB,C,Dの結果についてはモデルAとの相対値で表している。

図上部は効率について規格化したものである。そのため正解ラベル方向(行方向)の和が1になっている。混合行列では、NP・PV・SVCC・Othersが比較的高い効率で分離できていることが分かる。一方、各SVについては殆ど分離できておらず、個々のフレーバーや準崩壊点などは区別できていない。ただし、SVとそれ以外との分離は実現できており、SVであることは識別できているが、それがSVのどの崩壊点の種類であるかという点については識別が不十分となっている。

図下部は純度について規格化したものである。そのため予想されたクラス方向(列方向)の和が1になっている。純度については不均衡データとしての特性が強調され、図3.8のようなNCやPVが支配的なクラス配分となっていることから、各SVに対してのNCからの汚染が顕著である。ただし、これは後述する4.2節での崩壊点のタネの選別によってある程度の軽減が可能である。

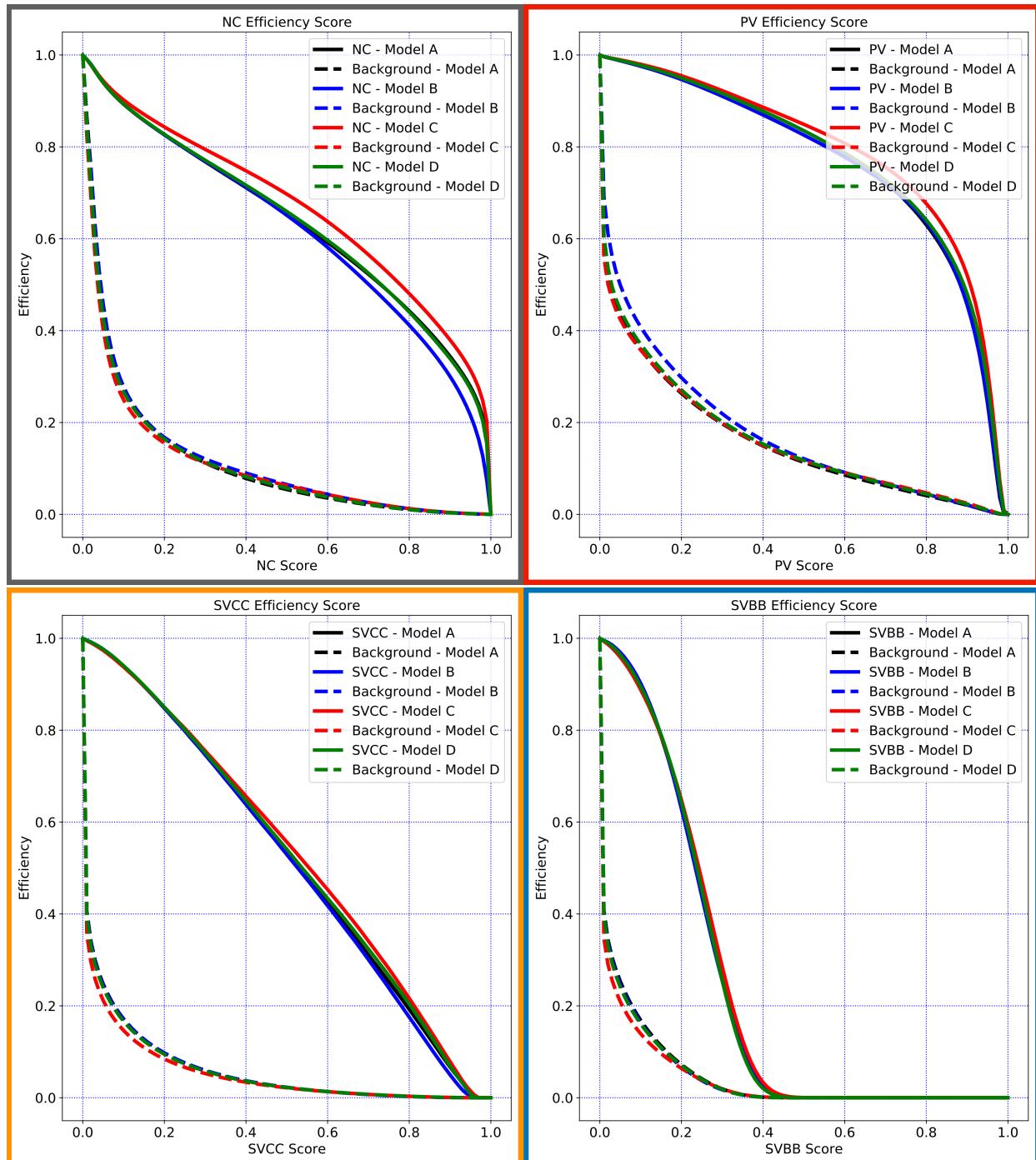


図 3.10: ネットワークのスコアとクラス分類の効率の関係。縦軸は効率、横軸はネットワークのスコアについての閾値である。実線は信号効率、破線はバックグラウンド効率である。黒線はモデル A, 青線はモデル B, 赤線はモデル C, 緑線はモデル D をそれぞれ表している。

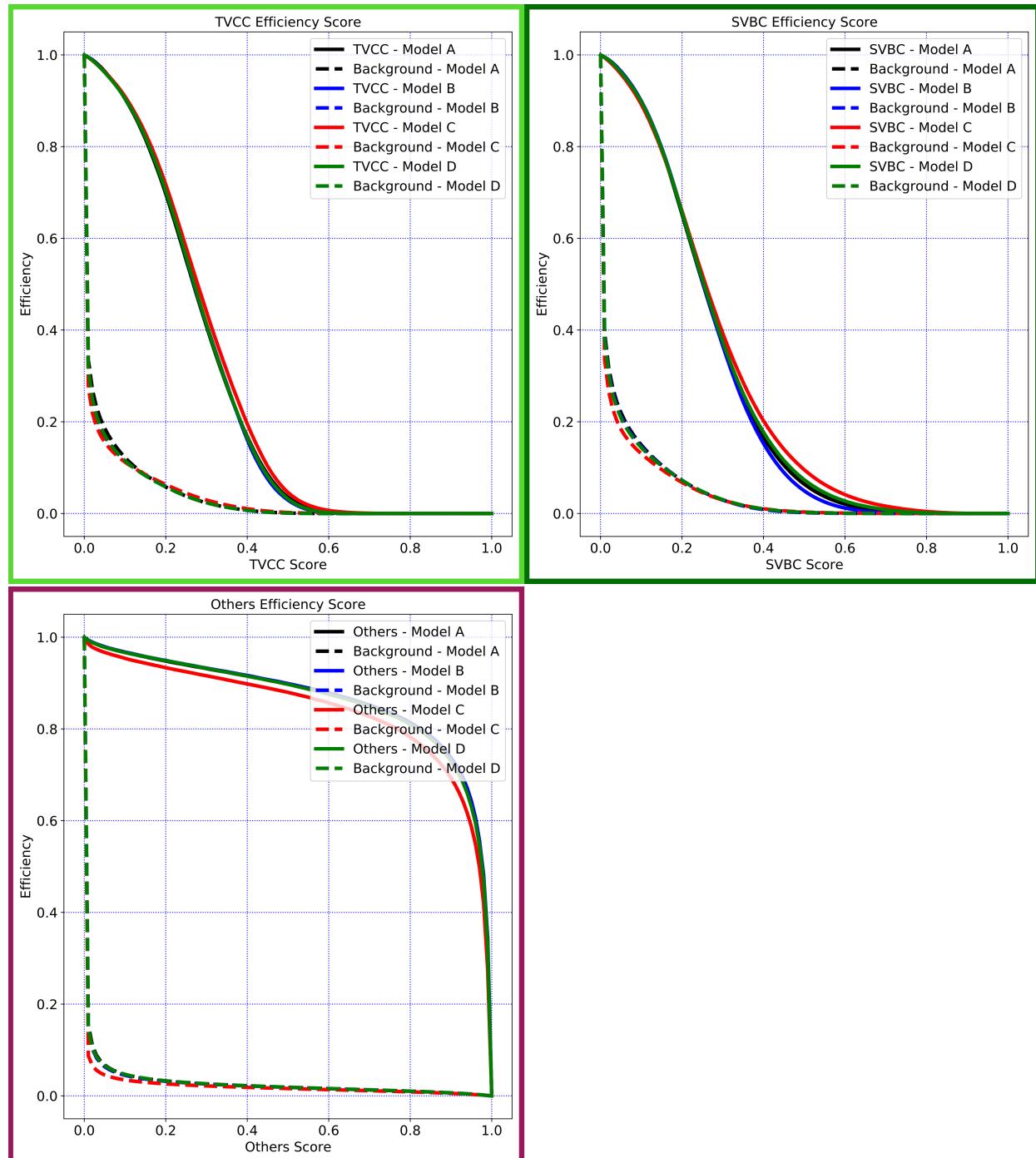


図 3.10: ネットワークのスコアとクラス分類の効率の関係。縦軸は効率、横軸はネットワークのスコアについての閾値である。実線は信号効率、破線はバックグラウンド効率である。黒線はモデル A、青線はモデル B、赤線はモデル C、緑線はモデル D をそれぞれ表している。

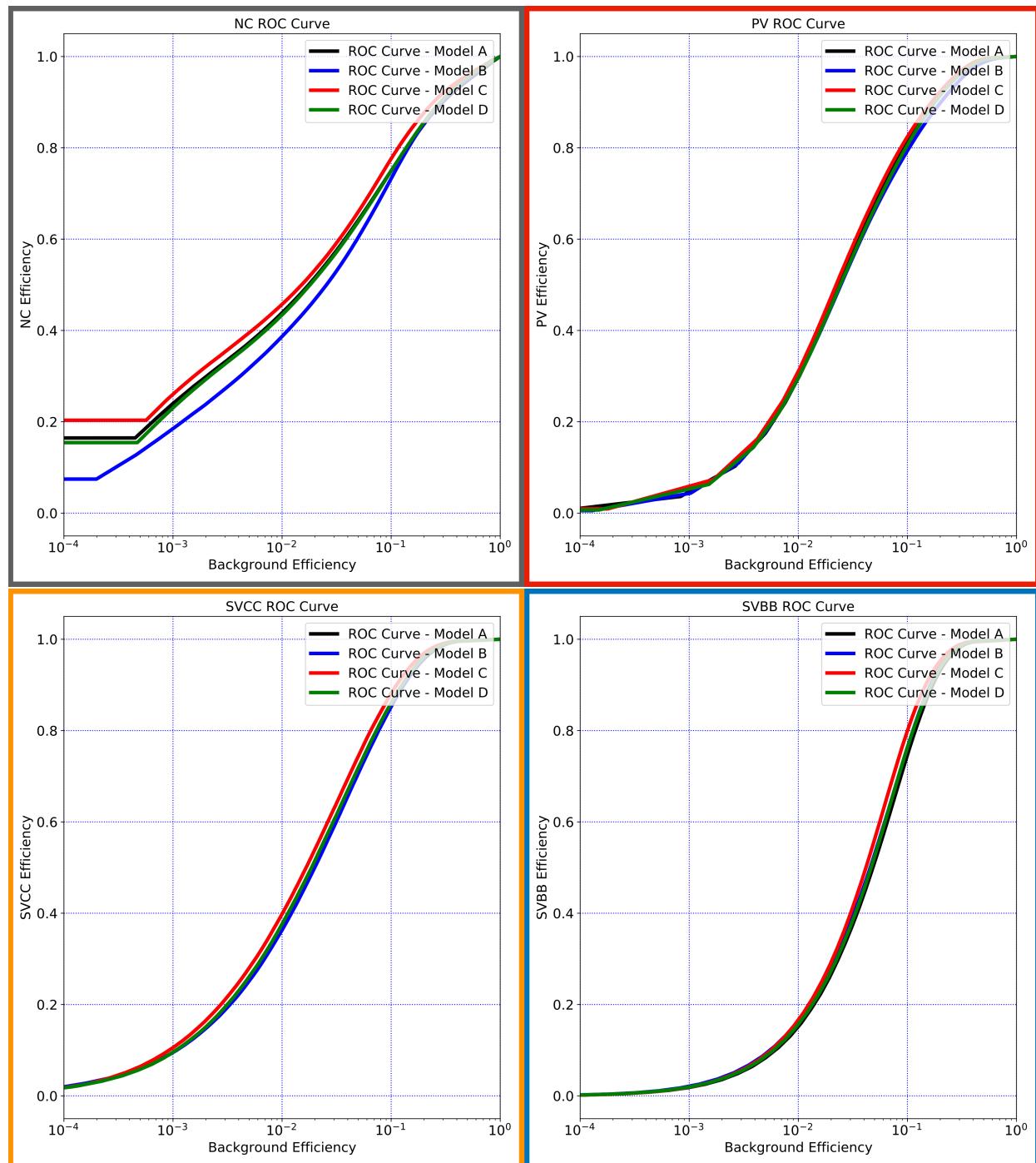


図 3.11: 各モデルの ROC 曲線。縦軸は信号効率、横軸はバックグラウンド効率である。黒線はモデル A, 青線はモデル B, 赤線はモデル C, 緑線はモデル D をそれぞれ表している。

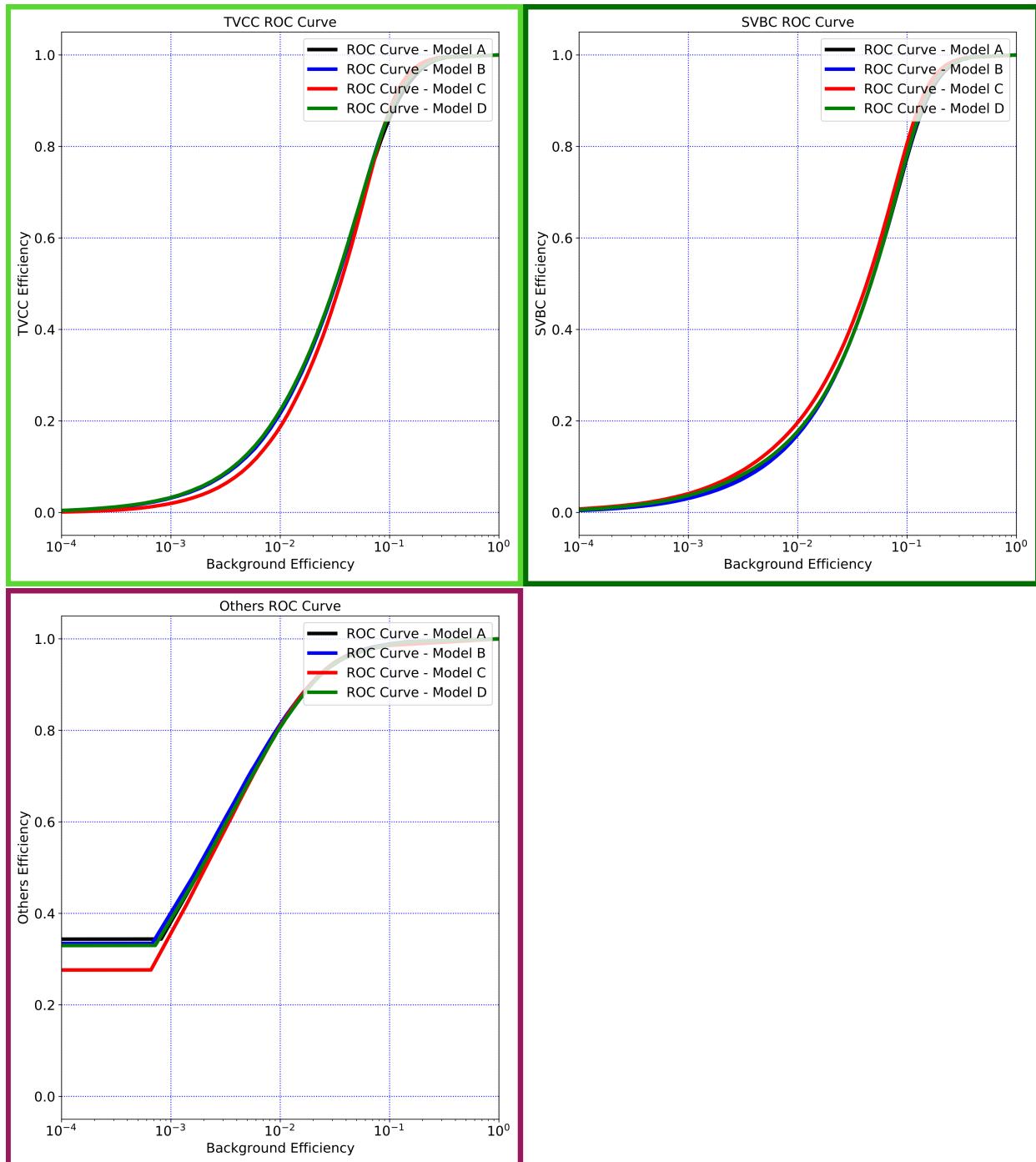


図 3.11: 各モデルの ROC 曲線。縦軸は信号効率、横軸はバックグラウンド効率である。黒線はモデル A、青線はモデル B、赤線はモデル C、緑線はモデル D をそれぞれ表している。

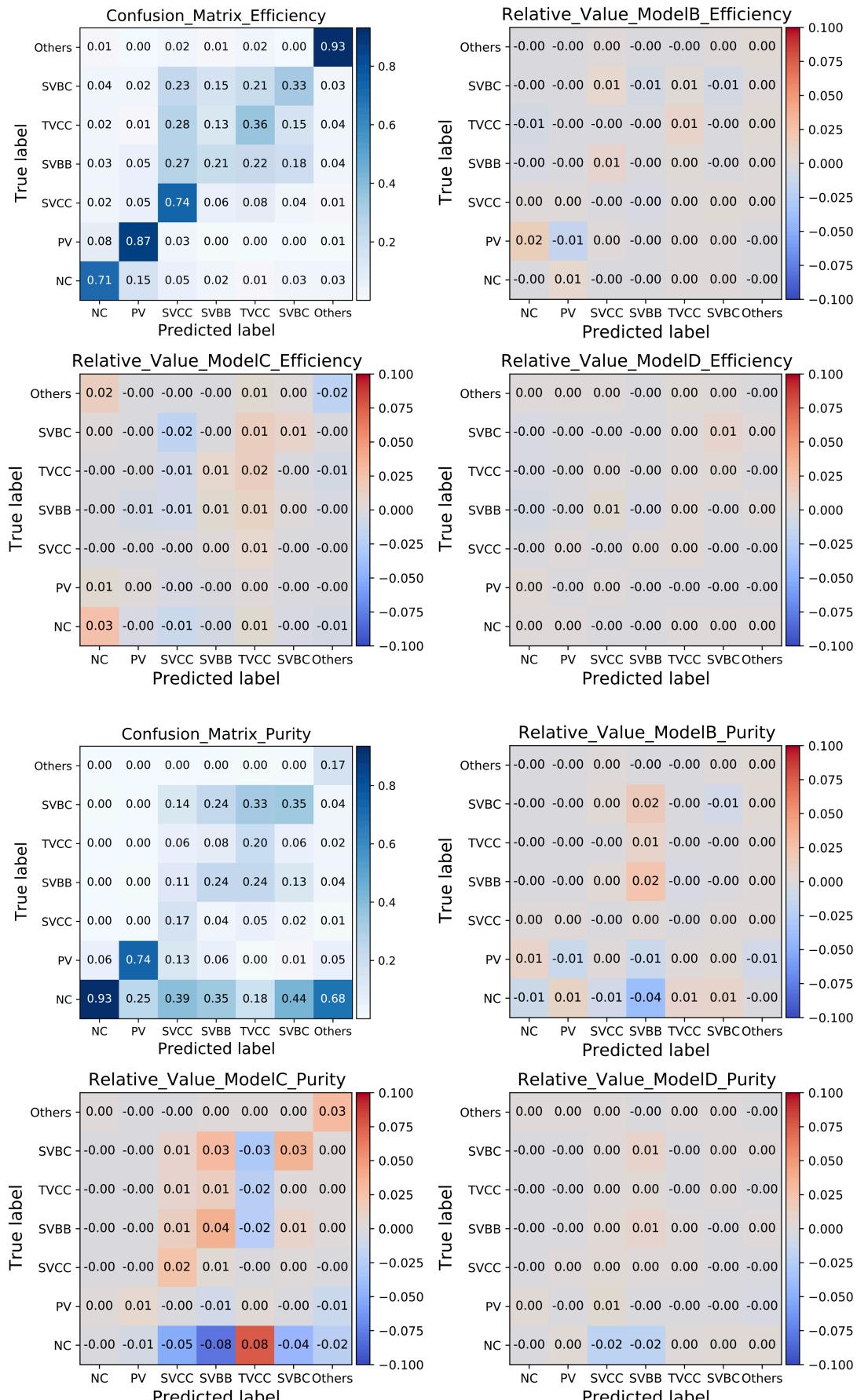


図 3.12: 各モデルの混合行列と各モデルの相対値。図上部は効率について規格化、図下部は純度について規格化した行列である。それぞれ左上がモデル A、右上がモデル B、左下がモデル C、右下がモデル D の結果である。モデル B・C・D に関してはモデル A との相対値で表現している。

相対値では対角成分が正であれば性能が高い, それ以外の成分が正であれば性能が低いと判断できる。モデル A・B・D についてはあまり変化はなく, 殆どの差異は 0.2% 程度に収まっている。モデル C についても同様に大きな違いは確認できないが, 効率について規格化した混合行列内の SVBB・TVCC・SVBC の分類性能が向上しており, かつ純度について規格化した混合行列内の NC からの SVCC・SVBB・SVBC への流入がある程度減少している。ここで, TVCC への流入については大幅に増大しているが, これは TVCC への他の SV からの流入が少なくなったことによって相対的に値が上昇している為であると考えられる。

以上のことから, ネットワークは NC・PV・Others や単に SV としては大まかにクラス分類ができていると分かる。一方, SV 内の各崩壊点の種類への分離は困難でありモデル C で多少の改善が見られることから, 詳細な位置の再構成には至っておらず, SV 内の分離に関しては更なる情報の入力, もしくは抜本的なネットワーク構造の改良が必要であると考えられる。

## 2. t-SNE によるネットワークの理解

深層学習のネットワーク内部を理解することは非常に難しい課題の一つである。ネットワークが何故, どのようにクラス分類を決定したかを把握する事は容易ではないが, どの程度各クラスを分離して判断できているかは次元削減によってある程度理解が可能である。ここでは, t-SNE という手法を用いて次元削減を行った。

t-SNE とは, 高次元の情報についての距離関係を維持しつつ, 低次元へマッピングするアルゴリズムである。データの局所的な構造や非線形の次元削減が可能であり, ここではネットワークがどの程度各クラスを分離できているかを二次元の画像として把握することができる。

そのような結果を図 3.13 に示す。この図の縦軸や横軸は圧縮後の次元をそれぞれ取っている為, 本質的に意味のない量である。したがって, 単に分布の違いのみによってその差を判断する。図左は入力変数を圧縮したもの, 図右は出力層の直前を圧縮したものである。各分類先のクラス (表 3.4) を色の違いで表現している。図左では各クラスの分布が重なっており, 図右では色毎にある程度分離し, 塊ができている事がわかる。これは深層学習を通す前の入力変数では殆ど分離できていなかった各クラスが, 出力層の直前ではある程度分離できている事を示しており, 深層学習のネットワークがその様な特徴量を上手く抽出できていることを表現している。一方, 右図においても以前, 各 secondary vertex (橙, 青, 黄緑, 緑) は非常に近い位置にマップされており, 分布が重なってしまっている。この様な場合, 各クラスの境界を適切に引く事ができない為, 分類が困難である考えられる。更に, 分布は線対称な形をしており, これは二つの飛跡についての情報を十分に混合できていないということを意味している。

以上のことから飛跡対についてのネットワークは未だ改善の余地を十分に残していると言える。

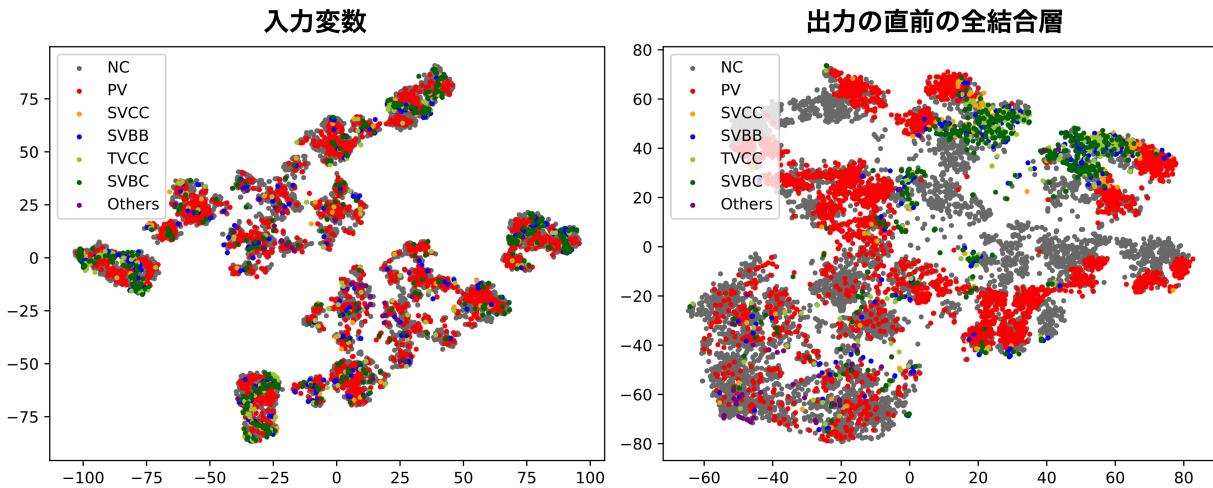


図 3.13: t-SNE による次元削減の比較。左図は入力変数 (44 変数) を二次元に削減したもの、右図は分類問題への分離の直前の全結合層 (256 変数) を二次元に削減したものである。

## 3.4 任意の数の飛跡についてのネットワーク

ここでは 3.2 節で紹介した二つのネットワークの内、任意の数の飛跡についてのネットワークに関して述べる。前節と同様にネットワークの構造・学習・評価について 3.4.1 項・3.4.2 項・3.4.3 項でそれぞれ解説する。

本節で紹介する任意の数の飛跡についてのネットワークは既にある構造ではなく、本研究で独自に開発したネットワーク構造を使用している。ここでは、それら独自構造についての解説も行う。

任意の数の飛跡についてのネットワークは、崩壊点を生成するためのネットワークである。ここでは、リカレントニューラルネットワークの初期状態として飛跡対 (崩壊点のタネ) を入力し、系列データとして事象中の全ての飛跡を入力する。また出力の作り方は Many to Many とし、事象中のそれぞれの飛跡が初期状態の崩壊点のタネに対して結合しているか否かを評価するネットワークを構築する (図 3.14)。

### 3.4.1 ネットワークの構造

まず、任意の数の飛跡についてのネットワークとして図 3.15 のようなネットワークを考える。

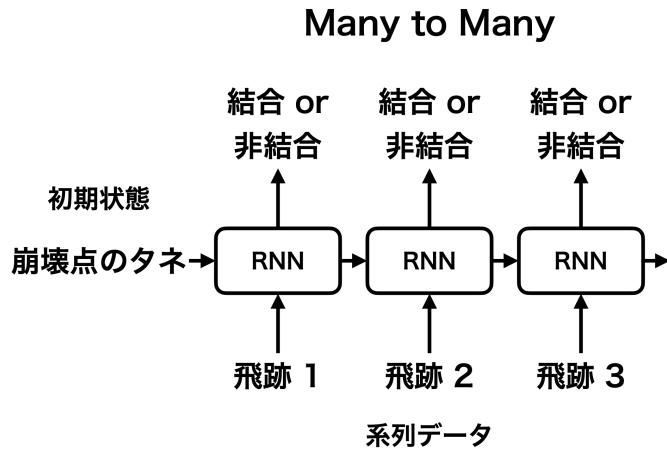


図 3.14: リカレントニューラルネットワークを用いた崩壊点生成。初期状態として崩壊点のタネを入力し、系列データとして事象中の飛跡を入力している。事象中の飛跡はそれ各自別で評価され、出力はそれらの飛跡が初期状態の崩壊点のタネと結合しているか、非結合であるかである。

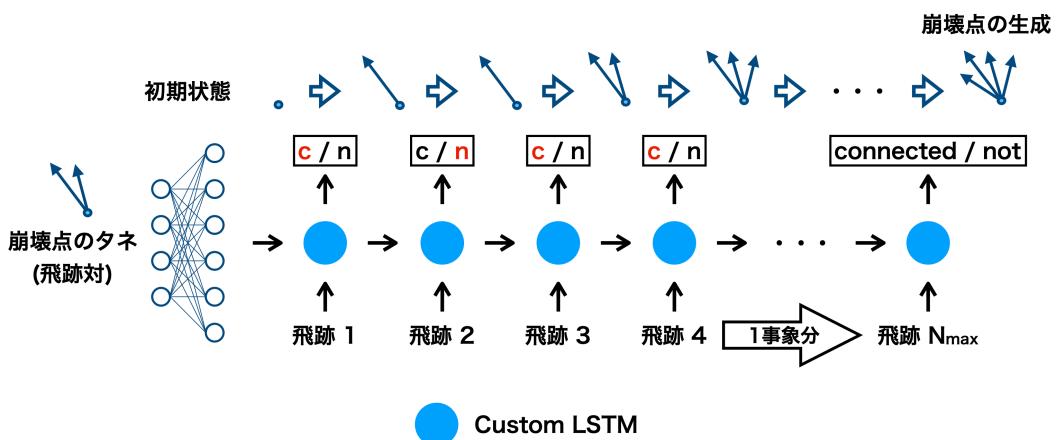


図 3.15: 崩壊点生成のためのリカレントニューラルネットワーク構造。図上部で崩壊点の生成の様子を図示した。飛跡がタネに結合していれば隠れ状態は逐次的に更新されていき、事象中の全ての飛跡を評価した段階で崩壊点が生成される。実際には系列データとして入力される飛跡に関しても、全結合層を用いて Embedding を行い抽象化している。

前述したように崩壊点のタネを初期状態として使用している。実際には崩壊点のタネは飛跡二本分の情報 (44 個の変数) であり、更に全結合層を介してより抽象的な崩壊点の情報を初期状態として入力できるようにしている。また系列データとして事象中の全ての飛跡を用いている。出力はこの飛跡がそれぞれ崩壊点のタネと結合しているか否かである。

2.4.3 項で解説したようにリカレントニューラルネットワークは系列情報を保持する為、直前の系列に依存するように設計されている。しかし飛跡は本質的に順序を持っていないことに注意する必要がある。図 3.15 では便宜的に飛跡について番号を振り表現しているが、これは人が

決めた順序であり、本来、事象中の飛跡は系列データではない。したがって、リカレントニューラルネットワークをそのまま用いることはデータの性質に合わない。この為私は、リカレントニューラルネットワークの一つである LSTM を拡張し、新しい独自のリカレントニューラルネットワークの構造をデザインした。

図 3.16 はそのような独自のリカレントニューラルネットワークについての 1 系列分のステップの詳細である。

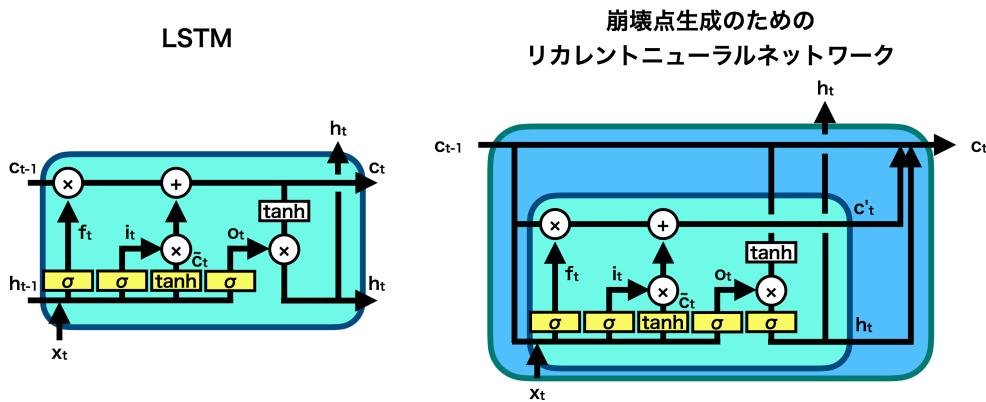


図 3.16: 系列 1 ステップについての独自リカレントニューラルネットワーク構造。図 3.15 の青丸の一つに相当している。

LSTM との大きな構造の違いは、短期記憶である  $h_t$  が隠れ状態として入出力されていない点である。入力は隠れ状態の記憶セル  $c_{t-1}$  (崩壊点の情報) と系列データ  $x_t$  (飛跡の情報) の二つである。また出力は結合・非結合を判定する  $h_t$  と隠れ状態の記憶セル  $c_t$  の二つである。隠れ状態として  $h_t$  が使用されていないため、内部の構造も通常の LSTM とは少し異なり、出力  $h_t$  と記憶セル  $c_t$  はそれぞれ

$$\begin{aligned} c_t &= (1 - h_t)c_{t-1} + h_t c'_t \\ c'_t &= c_{t-1} \sigma(W_f x_t + R_f c_{t-1}) + \tanh(W_c x_t + R_c c_{t-1}) \sigma(W_i x_t + R_i c_{t-1}) \\ h_t &= \sigma(d_h [\tanh(c_{t-1}) \sigma(W_o x_t + R_o c_{t-1})]) \end{aligned} \quad (3.4)$$

となる。ここで、学習可能な重み行列は  $W_f, W_i, W_c, W_o, R_f, R_i, R_c, R_o, d_h$  の九つである。第二式の  $c'_t$  は更新された記憶セルを示している。第三式は出力ゲートからのベクトル  $[\tanh(c_{t-1}) \sigma(W_o x_t + R_o c_{t-1})]$  に更に重み行列 (ベクトル  $d_h$ ) を掛けた形となっており、二値分類の為の一次元出力を作成している。第一式では、更新された記憶セル  $c'_t$  と直前の系列での記憶セル  $c_{t-1}$ 、出力  $h_t$  を用いて現在の記憶セル  $c_t$  を計算している。二値分類である為、 $h_t$  は 0 から 1 の値を持つはずである。したがって、第一式は結合 ( $h_t \sim 1$ ) していれば更新された記憶セル  $c'_t$  が、非結合 ( $h_t \sim 0$ ) であれば、直前の系列での記憶セル  $c_{t-1}$  が現在の記憶セル  $c_t$  となることを示している。

初期状態は崩壊点のタネであるので、以上の演算は次のように（図 3.17）解釈できる。

1.  $t - 1$  番目の崩壊点  $c_{t-1}$  と  $t$  番目の飛跡  $x_t$  が結合しているか否かの評価  $h_t$  を行う
2.  $t - 1$  番目の崩壊点  $c_{t-1}$  と  $t$  番目の飛跡  $x_t$  を用いて更新された崩壊点  $c'_t$  を計算する
3.  $t - 1$  番目の崩壊点  $c_{t-1}$  と  $t$  番目の飛跡  $x_t$  が結合しているならば更新された崩壊点  $c'_t$  を、結合していないならば  $t - 1$  番目の崩壊点  $c_{t-1}$  を  $t$  番目の崩壊点  $c_t$  として選択する

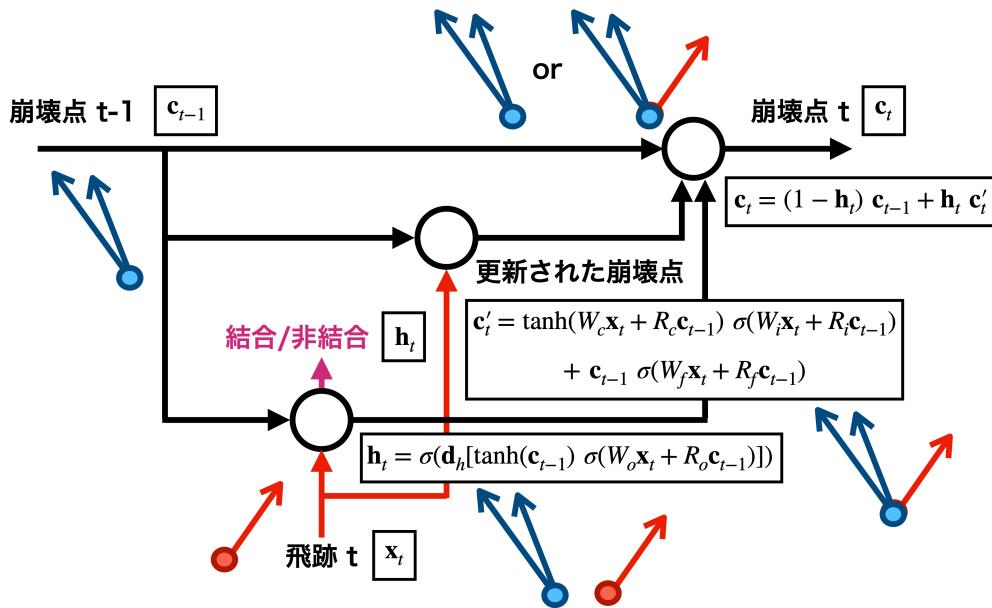


図 3.17: 独自リカレントニューラルネットワーク構造の解釈。それぞれの丸が演算に対応している。左下が演算 1, 中央が演算 2, 右上が演算 3 である。

短期記憶を破棄する事によって、独自リカレントニューラルネットワークでは、事象中の飛跡の順序（短期記憶）にできる限り依存せず、更に崩壊点のタネに対して飛跡を足していくことによって更新される崩壊点の情報（記憶セル）を表現している。

私たちはシミュレーションデータとして既に一つの事象について全ての飛跡の情報を持っている。したがって、作成したリカレントニューラルネットワークをエンコーダー・デコーダーモデルに組み込むことで、事象についての情報（コンテキスト）を活用することができると言えられる。また、エンコーダー・デコーダーモデルの間に Attention を組み込むことも同様に自然な発想である。その様なネットワークを図 3.18 に示す。

図の上部はエンコーダー部である。エンコーダー部では事象中の飛跡から事象全体の情報（コンテキスト）を抽出している。ここでは先ほどの図 3.15 で紹介したネットワークを双方向リカレントニューラルネットワークとして使用している。双方向リカレントニューラルネットワークでは個々のステップの出力は順方向と逆方向の二つ存在する為、本ネットワークでは単

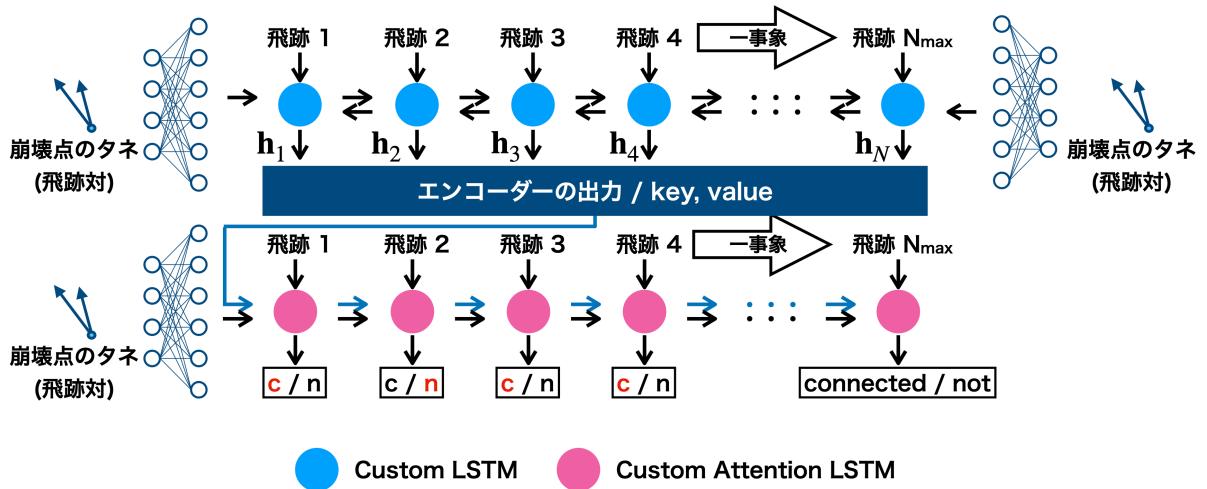


図 3.18: Attention を組み込んだエンコーダー・デコーダーモデルへの拡張。図上部がエンコーダー部、図下部がデコーダー部をそれぞれ示している。初期状態や系列データはエンコーダー部とデコーダー部のそれぞれに入力されている。エンコーダー部では双方向 RNN を用いている。ただし、個々のステップは独自構造のネットワークになっている。エンコーダー部の出力は図中央の Key・Value に集められ、デコーダー部に入力される。デコーダー部での出力はそれぞれの飛跡がタネと結合しているか、非結合であるかの二値分類である。

に出力ベクトルを結合している。<sup>\*6</sup>ここで、式 (3.4) では出力  $h_t$  は二値分類の為、一次元となっていた。しかしエンコーダー部は、デコーダー部に対してより多くの情報を残す必要がある為、式 (3.4) で示した様な一次元の情報では不足であると考えられる。このことからエンコーダー部の出力次元数を減らさないようにするために重み行列  $d_h$  を取り除き、

$$\begin{aligned} h_t &= \sigma(d_h[\tanh(c_{t-1}) \ \sigma(W_o x_t + R_o c_{t-1})]) \\ &\rightarrow h_t = \tanh(c_{t-1}) \ \sigma(W_o x_t + R_o c_{t-1}) \end{aligned} \quad (3.5)$$

として、結合/非結合のスカラー量の代わりに  $d_h$  を通す前の出力ベクトルをエンコーダーの出力としている。また図中で表現しているように、初期状態として入力されている崩壊点のタネはそれぞれ別の全結合層によって情報を抽象化されている。

図の下部はデコーダー部である。デコーダー部ではエンコーダー部で抽出された情報と崩壊点のタネ、事象中の飛跡を使用して崩壊点のタネにそれぞれの飛跡が結合しているか否かを判別している。エンコーダー部で抽出された情報は Attention によって適切に評価され、デコーダー部の“ある”飛跡がエンコーダー部の任意の飛跡に対して注意を払って、事象中の情報を取得できるようになっている。またエンコーダー・デコーダーモデルへの拡張後も、このネットワークの基本構造がリカレントニューラルネットワークであることに変わりはない為、入力す

<sup>\*6</sup> 例えば、256 次元のベクトルがそれぞれ順方向と逆方向から出力された場合は双方向による最終的な出力は 512 次元のベクトルとする

る飛跡の本数を任意に変えることが可能である。

図 3.16 で示したネットワーク構造は Attention には対応していないため、新たなネットワークの構築が必要である。そのようなネットワークを図 3.19 に示す。

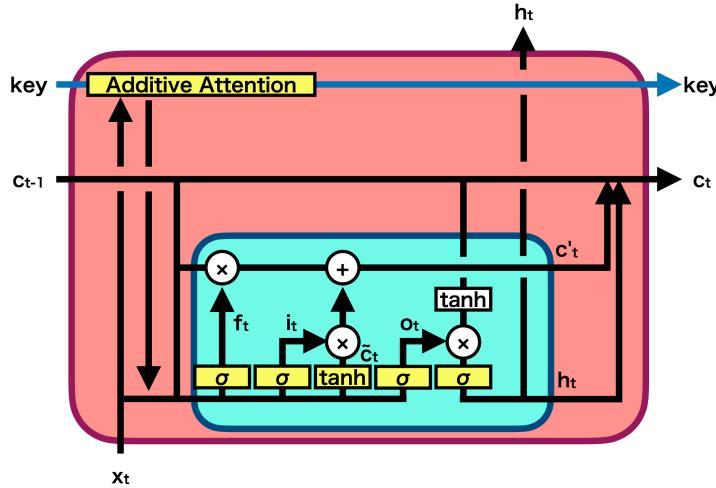


図 3.19: 独自リカレントニューラルネットワークの Attention への拡張。図 3.18 の赤丸の一つに相当している。図上部の青線は不变記憶 Key・Value の流れを表している。Attention に関する演算が加えられている点以外は図 3.16 と同様の構造である。

本研究では、エンコーダー部で抽出された情報 (Key) をリカレントニューラルネットワークの隠れ状態の一つとして入力している。また、この Key はデコーダー部の全系列について共通の値を使用しており、長期記憶・短期記憶と比べて不变記憶のような役割を果たしている。Attention Weight の計算方法として Additive Attention を採用した。 $t$  番目のコンテキスト  $\gamma_t$  は次のように計算される。

$$\begin{aligned} \gamma_t &= \alpha_t V \\ \alpha_t &= (\alpha_{t,0}, \alpha_{t,1}, \alpha_{t,2}, \dots \alpha_{t,i}, \dots) \\ &= \left( \frac{\exp(e_{t,0})}{\sum_j \exp(e_{t,j})}, \frac{\exp(e_{t,1})}{\sum_j \exp(e_{t,j})}, \frac{\exp(e_{t,2})}{\sum_j \exp(e_{t,j})}, \dots \frac{\exp(e_{t,i})}{\sum_j \exp(e_{t,j})}, \dots \right) \quad (3.6) \\ e_t &= (K U_{\text{key}} + X_t U_{\text{query}}) u_{\text{energy}} \end{aligned}$$

ここで、Key を  $K$ 、Value を  $V$  とした。Key, Value は同じエンコーダー部で抽出された情報を使用している。 $t$  番目の Query である飛跡 (ベクトル)  $x_t$  をエンコーダーの飛跡の数だけ積み上げた行列を  $X_t$ 、 $t$  番目の Attention Weight を  $\alpha_t$ 、 $t$  番目の Query についてのエネルギーを  $e_t$  とした。また、Additive Attention における重み行列をそれぞれ  $u_{\text{energy}}$ ,  $U_{\text{key}}$ ,  $U_{\text{query}}$  と置いた。添字  $i, j$  はエンコーダー部の系列、添字  $t$  はデコーダー部の系列である。本研究で

の実装では最終的に得たい情報を考慮して  $t$  番目の Query について,  $t - 1$  番目の隠れ状態では無く,  $t$  番目の飛跡 (ベクトル)  $\mathbf{x}_t$  を使用している。<sup>7</sup>

式 (3.6) の図解を図 3.20 に示す。Key, Value は飛跡一本分のエンコーダー部の出力  $\mathbf{h}$  をエンコーダー部の全ての飛跡分集めた行列である。したがって Key, Value の大きさは (エンコーダー部の飛跡の数 × エンコーダー部の出力の次元数) となる。他方, Query はデコーダー部のある飛跡  $\mathbf{x}_t$  である。ただし, Additive Attention では最終的に得たい Attention Weight を作成する為, 図 3.20 中の様にデコーダー部のある飛跡  $\mathbf{x}_t$  をエンコーダー部の飛跡の数だけ積み重ねている。ゆえに最終的な Query の大きさは (エンコーダー部の飛跡の数 × 飛跡の次元数) となる。式 (3.6) の第三式では, この Key, Query について重み行列  $\mathbf{u}_{\text{energy}}$ ,  $\mathbf{U}_{\text{key}}$ ,  $\mathbf{U}_{\text{query}}$  を掛ける事によって Query についてのエネルギー  $e_t$  を計算している。

更に第二式では, エネルギー  $e_t$  をソフトマックス関数によって規格化し, Attention Weight を計算している。Attention Weight は, あるデコーダー部の飛跡がエンコーダー部のどの飛跡に注目しているかを示しており, 第一式はエンコーダー部のどの飛跡に注意し, エンコーダー部から情報を取り出すかを計算している。よって取り出された情報 (コンテキスト  $\gamma_t$ ) はエンコーダー部の出力の次元数の大きさを持つベクトルとなる。

デコーダー部の全ての飛跡について Attention Weight を計算した時, Attention Weight は (エンコーダー部の飛跡の数 × デコーダー部の飛跡の数) の行列となる。この Attention Weight 行列はネットワーク内部を把握する上で非常に重要な情報である。図 3.19 では表現していないがオプションとして Attention Weight を出力することで, ネットワーク内部をある程度理解することができる。

得られた  $t$  番目のコンテキスト  $\gamma_t$  は, 出力  $h_t$  や更新された崩壊点  $\mathbf{c}'_t$  の計算に使用される。

$$\begin{aligned} \mathbf{c}_t &= (1 - h_t) \mathbf{c}_{t-1} + h_t \mathbf{c}'_t \\ \mathbf{c}'_t &= \mathbf{c}_{t-1} \sigma(W_f \mathbf{x}_t + R_f \mathbf{c}_{t-1} + C_f \gamma_t) \\ &\quad + \tanh(W_c \mathbf{x}_t + R_c \mathbf{c}_{t-1} + C_c \gamma_t) \sigma(W_i \mathbf{x}_t + R_i \mathbf{c}_{t-1} + C_i \gamma_t) \\ h_t &= \sigma(\mathbf{d}_h [\tanh(\mathbf{c}_{t-1}) \sigma(W_o \mathbf{x}_t + R_o \mathbf{c}_{t-1} + C_o \gamma_t)]) \end{aligned} \tag{3.7}$$

式中ではコンテキスト  $\gamma_t$  に関する各ゲートそれぞれの重み行列を  $C_f$ ,  $C_c$ ,  $C_i$ ,  $C_o$  と置いた。 $t$  番目のコンテキスト  $\gamma_t$  についての演算を加えている点以外は図 3.16 でのネットワークの演算と全く同様である。

---

<sup>7</sup> 一般に Attention が使用される機械翻訳などにおいてはデコーダー部への入力は存在しないが, 今回はデコーダー部への入力 (飛跡) が存在する為, こちらを使用した。

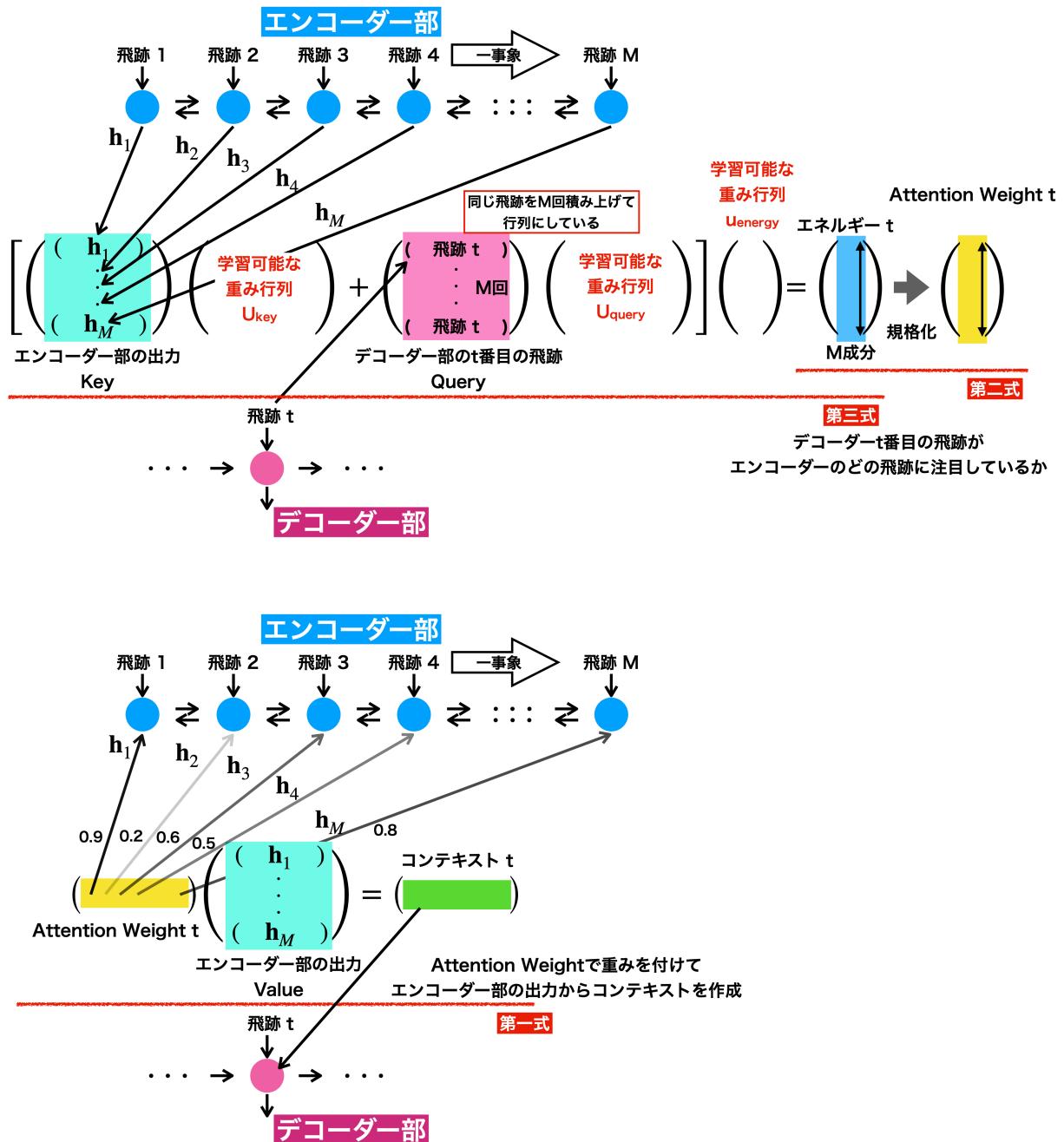


図 3.20: 本研究における Additive Attention の図解。

### 3.4.2 ネットワークの学習と戦略

訓練データとして必要な情報は、初期状態としての飛跡対（崩壊点のタネ）と事象中の全ての飛跡である。また、正解ラベルはそれぞれの飛跡が崩壊点のタネと結合しているか否かのMC情報である。推論時は、初期状態の崩壊点のタネとして非結合な飛跡対が入力される場合を考えられるが、本研究ではネットワークの学習時は崩壊点のタネとして結合している飛跡対(PV・SVCC・SVBB・TVCC)のみをMC情報によって選択し使用する。ここで、準崩壊点SVBCは崩壊点生成において雑音となりうる可能性があるため含んでいない。

リカレントニューラルネットワークでは、推論時は系列長を変えることができるが、学習時は重み更新の計算のため系列長を揃える必要がある。<sup>\*8</sup>ここで系列長は事象中の飛跡の本数であった。このため、不足している飛跡の本数をゼロ埋め(zero padding)し、ゼロ埋めした飛跡が学習に影響しないよう損失関数においてマスクしている。本研究では最も飛跡数の多い事象との兼ね合いから系列長を60本とし、それ以下の本数の事象については60本となるようにゼロ埋めしている。

最終的な入力変数の大きさを表3.7に示す。

表3.7: 任意の数の飛跡についてのネットワークの入力変数の大きさ

	PV	SV
崩壊点のタネ	(1661307, 44)	(367434, 44)
事象中の全ての飛跡	(1661307, 60, 23)	(367434, 60, 23)

ここで、第一引数はデータのサンプル数である。訓練データは全ての崩壊点のタネについて一つのサンプルが生成されるため、全ての事象( $c\bar{c} - 05, b\bar{b} - 06$ )を使用すると非常に時間がかかる。よって、本研究では全ての事象を使用して訓練データを作成した後、1エポック毎にランダムに50000サンプルを訓練に10000サンプルを検証に使用した。事象中の全ての飛跡については飛跡の本数である60本とそれぞれの飛跡について22個の変数を持っている。また、ゼロ埋めした飛跡との区別のためのマスク変数(0, 1)を一つ加えている。

崩壊点生成において、飛跡は順序を持って足されていく。短期的な順序に依存しないような独自のネットワークを構築しているが、そのような人によって決められた飛跡の順序にネットワークが依存してしまうことは、できる限り避けねばならないため、1エポック毎(2.6)に飛跡の系列順をランダムにシャッフルしている。

損失関数は二値交差エントロピー誤差を使用した。ただし、前述したようにゼロ埋めした飛跡については損失関数や正答率の計算ではマスクした。

$$L = -M_{ZP} t_C \log(y_C) - M_{ZP} (1 - t_C) \log(1 - y_C) \quad (3.8)$$

<sup>\*8</sup> 実際にはバッチサイズ毎に

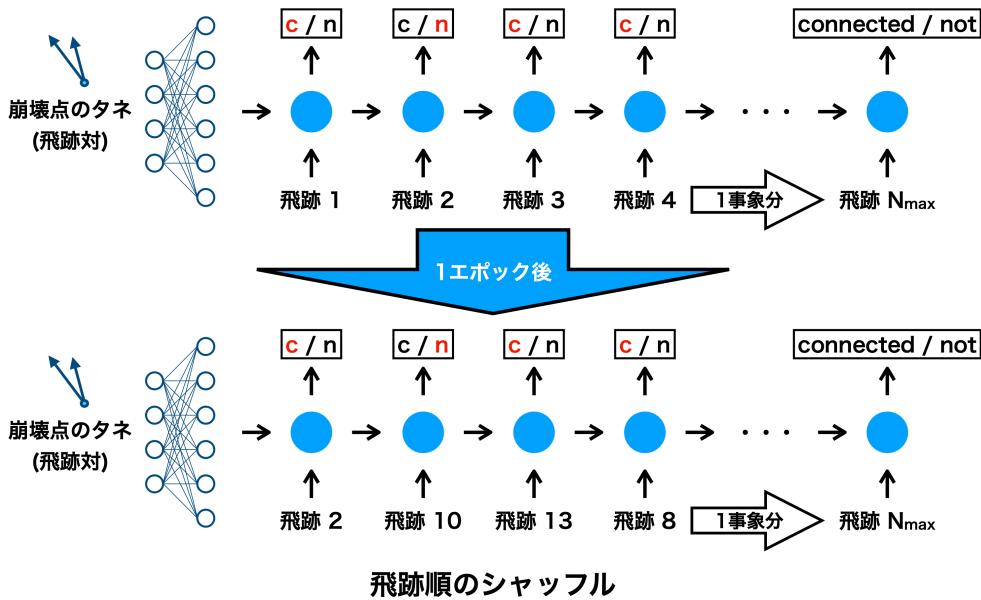


図 3.21: 飛跡順のシャッフル。1 エポック毎に系列データの飛跡順序をシャッフルし、崩壊点のタネへ足し合わせる順番への依存を取り除いた。この順序のシャッフルは学習や検証に使用する全てのデータについて同一である。

$M_{ZP}$  はゼロ埋めのためのマスク変数 (0, 1) である。

ハイパーパラメータとして、最適化手法は Adam、学習率は 0.001、エポック数は 100、バッチサイズは 32 を用いた。また、ネットワークの学習可能な重みのパラメータ数を表 3.8 に示す。

### 3.4.3 ネットワークの評価

ネットワークの評価は飛跡対についてのネットワークと同様に、1. ネットワーク間の比較と 2. ネットワーク内の理解の二つの観点で行う。任意の数の飛跡についてのネットワークでは、1. ネットワーク間の比較については標準的な LSTM と本研究で構築した独自のネットワークの比較を行う。更に、終状態の違いや崩壊点のタネの違いについて、各データ属性に特化したネットワークとそれら全てのデータを用いて学習した標準的なネットワークについての比較を行う。2. ネットワーク内の理解については任意の数の飛跡についてのネットワークは内部に Attention を持っているため、Attention Weight を確認することにより、エンコーダー部からどのような情報を抽出しているかについて調査する。

表 3.8: 任意の数の飛跡についてのネットワークにおける訓練可能なパラメータ数

層の名称	出力の形状	パラメータ数	接続先
Pair Input	(None, 44)	0	
Encoder Input	(None, 60, 23)	0	
Decoder Input	(None, None, 23)	0	
Encoder Forward Dense 1	(None, 256)	11520	Pair Input
Encoder Backward Dense 1	(None, 256)	11520	Pair Input
Encoder Forward Activation 1	(None, 256)	0	Encoder Forward Dense 1
Encoder Backward Activation 1	(None, 256)	0	Encoder Backward Dense 1
Encoder Forward Dense 2	(None, 256)	11520	Encoder Forward Activation 1
Encoder Backward Dense 2	(None, 256)	11520	Encoder Backward Activation 1
Encoder Forward Activation 2	(None, 256)	0	Encoder Forward Dense 2
Encoder Backward Activation 2	(None, 256)	0	Encoder Backward Dense 2
Encoder Embedding Dense	(None, 60, 256)	6144	Encoder Input
Bidirectional Encoder VLSTM	(None, 60, 512)	1050624	Encoder Embedding Dense Encoder Forward Activation 2 Encoder Forward Activation 2 Encoder Backward Activation 2 Encoder Backward Activation 2
Reshape Bidirectional Encoder	(None, 27136)	0	Bidirectional Encoder VLSTM
Decoder Dense 1	(None, 256)	11520	Pair Input
Decoder Activation 1	(None, 256)	0	Decoder Forward Dense 1
Decoder Dense 2	(None, 256)	11520	Decoder Forward Activation 1
Decoder Activation 2	(None, 256)	0	Decoder Forward Dense 2
Decoder Embedding Dense	(None, None, 256)	6144	Encoder Input
Decoder Attention VLSTM	(None, None, 1)	1246976	Decoder Embedding Dense Reshape Bidirectional Encoder Decoder Activation 2

## 1. ネットワーク間の比較

まず、以下の三つのネットワークの比較によって独自のネットワーク構造がどの程度効果的であるかの確認を行う。

- 標準的な LSTM：図 3.15 のネットワークの個々のステップを標準的な LSTM に置き換えたネットワーク
- 独自の LSTM：図 3.15 のネットワーク
- 独自の Attention LSTM：図 3.18 のネットワーク

これらのネットワークに関して損失と正答率のエポック毎の変化を図 3.22 に示す。ここでは全ての評価基準についてゼロ埋めした飛跡を取り除いている。また、ネットワークの構造やデータ特性を考慮し、正答率や真陽性率 (True Positive Rate, TPR), 真陰性率 (True Negative Rate, TNR) の計算から初期状態となる飛跡対と同じ飛跡を取り除いている。

標準的な LSTM は系列の順序を重視しているため学習が安定していないことが分かる。また、独自のネットワーク同士の比較においても、エンコーダー部からのコンテキストを受け取

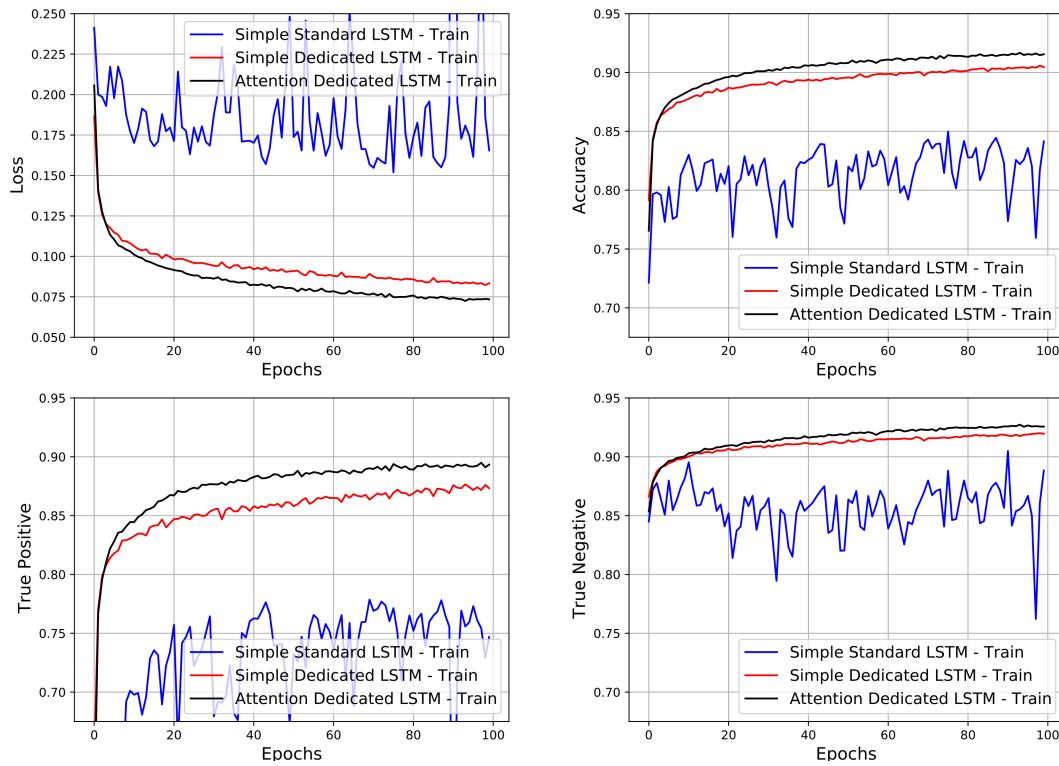


図 3.22: 標準的な LSTM と独自のネットワークの比較。左上が損失、右上が正答率、左下が TPR、右下が TNR である。それぞれ青線は標準的な LSTM、赤線は独自の LSTM、黒線は独自の Attention LSTM についての結果を表している。

ることのできる独自の Attention LSTM の性能が高くなっている。以上のことから、標準的な LSTM を単純に使うことは不適であり、独自のネットワーク構造による大幅な性能の改善が可能であると確認できる。

次に独自の Attention LSTM についての比較を行う。ここでは訓練データをデータ属性毎に分離した。ここでデータ属性とは終状態 ( $c\bar{c} \cdot b\bar{b}$ )、崩壊点のタネ (primary vertex (PV) · secondary vertex (SV)) の 4 つである。それぞれのデータ属性のみで構成された訓練データで学習した特化型のネットワークを用意した。それらのネットワークと全てのデータを使用した標準的なネットワークについての比較の結果を図 3.24 と図??に示す。

- ALL : 全てのデータを使用した標準的なネットワーク
- CC, BB : 終状態が  $c\bar{c}$ ,  $b\bar{b}$  のデータのみを使用した特化型のネットワーク
- PV, SV : 崩壊点のタネが primary vertex, secondary vertex のデータのみを使用した特化型のネットワーク

比較に関しては ROC 曲線を使用している。評価に際してのテストデータは特化型のネットワークの学習に使用した訓練データと同じ属性のものを用いた。終状態  $c\bar{c}$  や終状態  $b\bar{b}$  に関しては基本的な性能の差がほとんどないと分かる。したがって、個々の終状態のデータに関してはその特性に大きな差異は存在しないと判断できる。各崩壊点のタネに特化したネットワーク

に関しては両者とも性能の向上が見られた。特に、primary vertex に特化したネットワークは結合と非結合との分離が非常に良くできている。ROC 曲線では SV に特化したネットワークが標準的なネットワークより大幅に性能が改善されていると分かる。

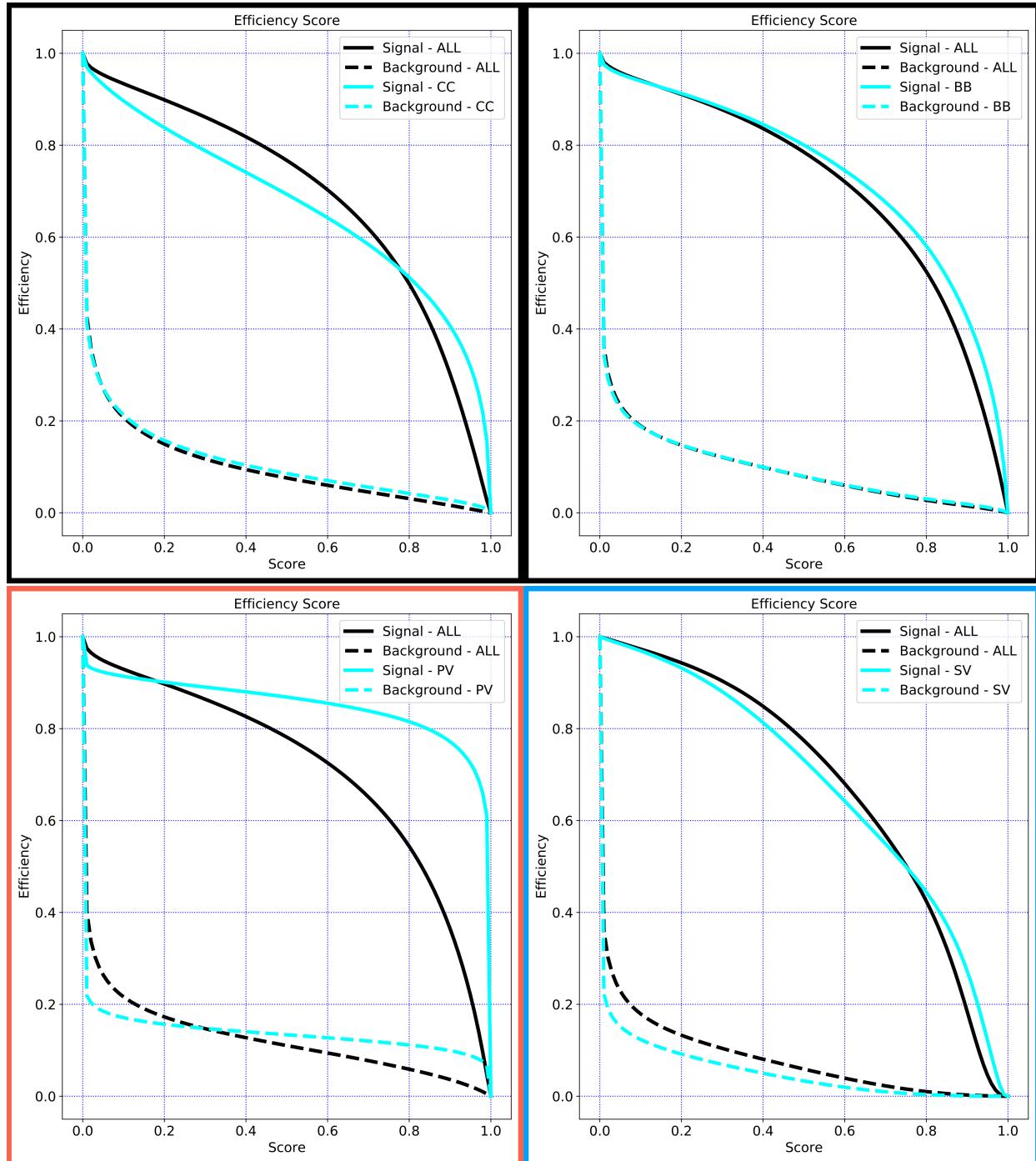


図 3.23: 各データ属性の効率とスコアの関係。縦軸は効率、横軸はネットワークのスコアについての閾値である。実線は信号効率、破線はバックグラウンド効率である。黒線は全データを使用した標準的なネットワーク、青線は個々の特化型のネットワークをそれぞれ表している。

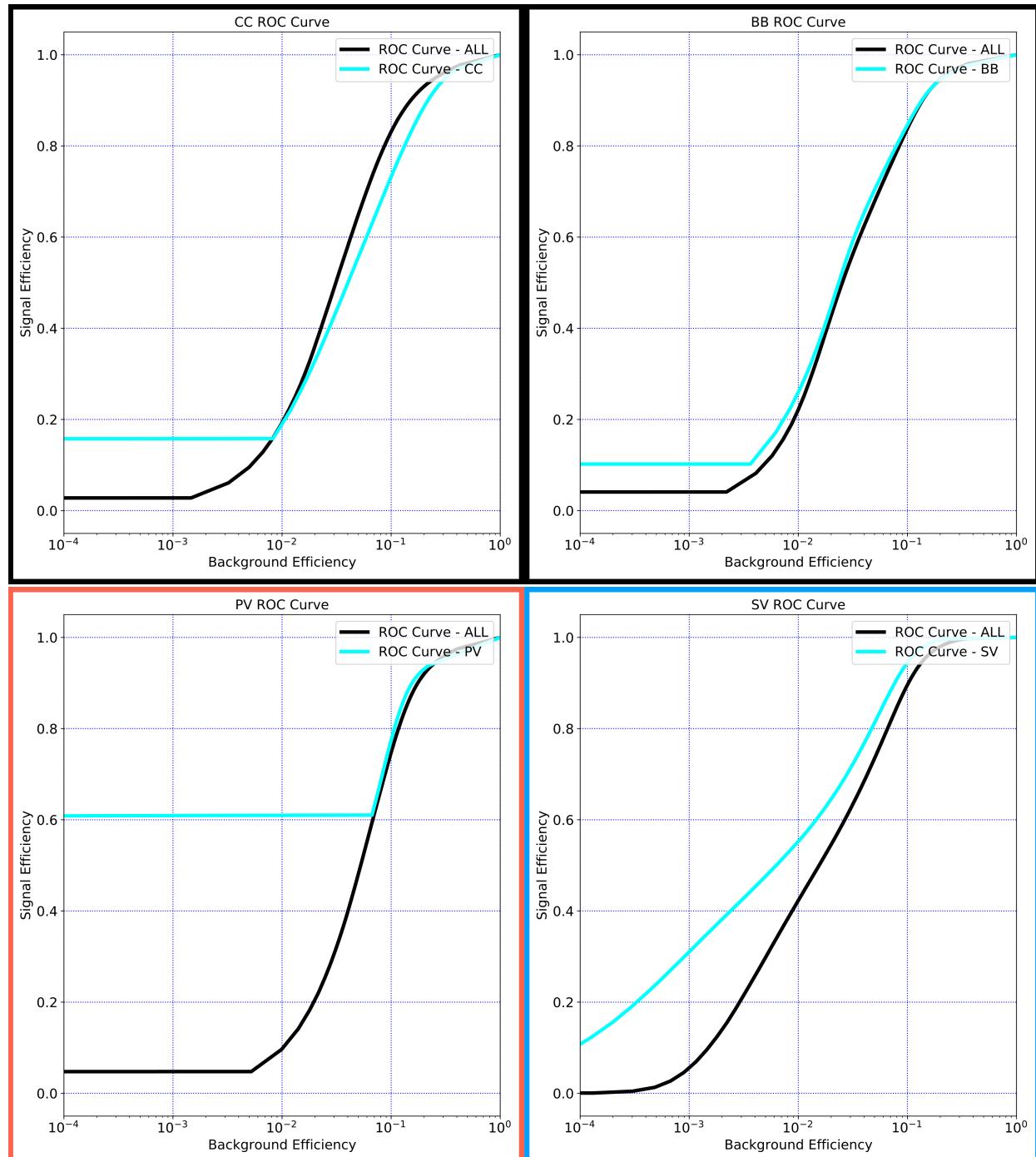


図 3.24: 各データ属性の ROC 曲線。縦軸は信号効率、横軸はバックグラウンド効率である。黒線は全データを使用した標準的なネットワーク、青線は個々の特化型のネットワークをそれぞれ表している。

## 2. ネットワーク内の理解

飛跡対についてのネットワークとは異なり、任意の数の飛跡についてのネットワークは内部に Attention を持っている。したがって、Attention Weight を確認することによって、ネットワークを把握することができる。具体的には、エンコーダー部の飛跡がデコーダー部のどのような飛跡に注意し情報を受け取っているかを確認することが可能である。そのような Attention Weight について、標準的なネットワークと PV のタネを用いて作成したサンプルを図 3.25 に示す。

図上部は横軸がデコーダー部の飛跡番号、縦軸がエンコーダー部の飛跡番号となっている。ここでは Attention Weight はカラースケールで表されている。デコーダー部の飛跡による Attention Weight の出力を縦方向に積んだような表現方法である。

図下部はエンコーダー部の飛跡を上に、デコーダー部の飛跡を下に並べた図となっている。ここでは Attention Weight は透過度で表現されている。また、デコーダー部の飛跡の内、崩壊点のタネと結合しているものを赤、結合していないものを青で描画している。デコーダー部の飛跡の数が少ないのでゼロ埋めを取り除いている為である。

これらの図から結合している飛跡はエンコーダー部の飛跡に注意し、結合していない飛跡はエンコーダー部のゼロ埋めに注意している傾向があると分かる。

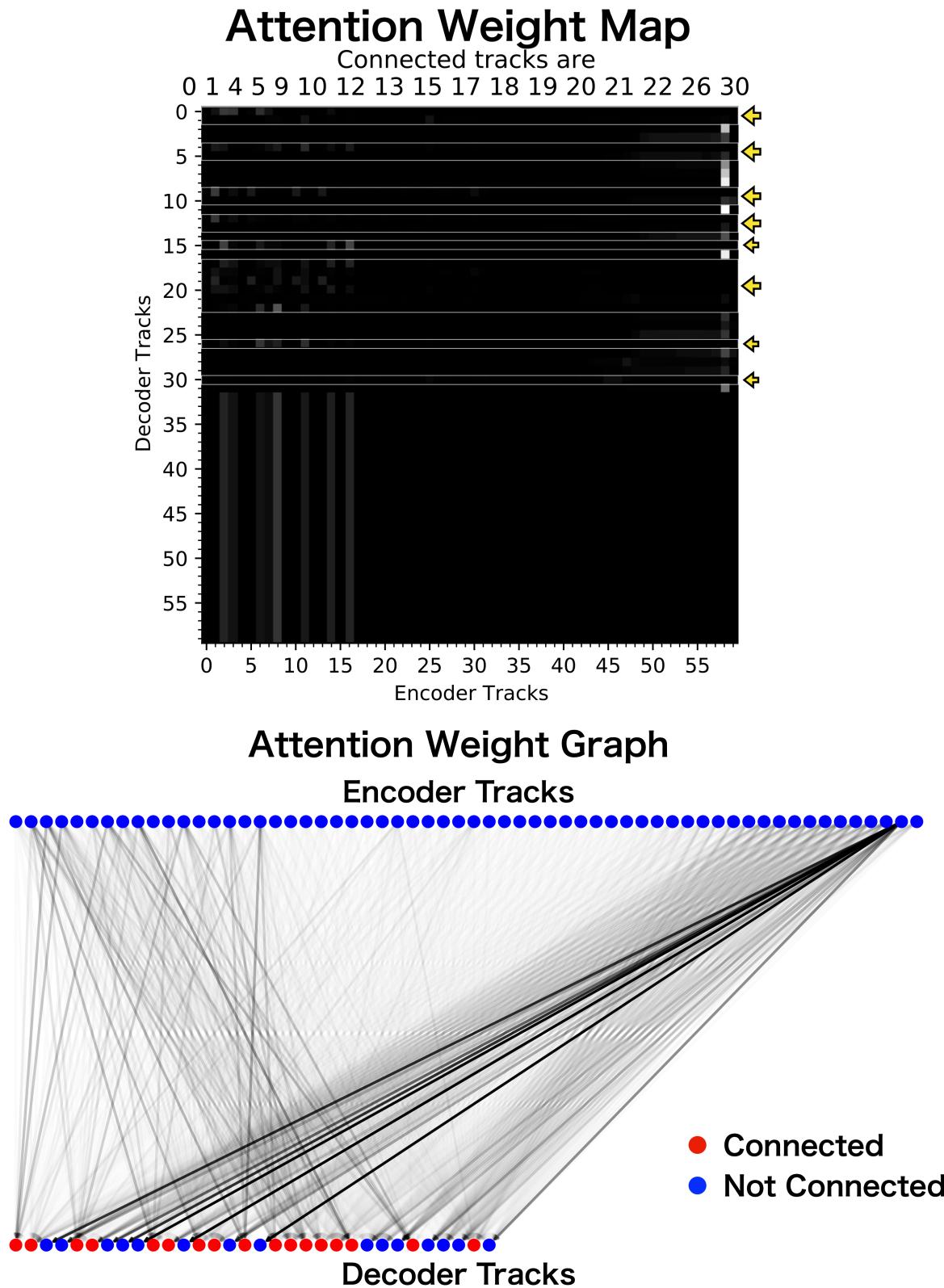


図 3.25: 図上部は Attention Weight を行列として表現した図、図下部は Attention Weight をより直感的に示した図である。図上部では縦軸がデコーダーの飛跡番号、横軸がエンコーダーの飛跡番号となっている。また、カラースケールは Attention Weight である。図下部では上段にデコーダーの飛跡、下段にデコーダーの飛跡を描画している。デコーダーの飛跡ではタネと結合している飛跡については赤丸を使用している。

## 第 4 章

# 深層学習を用いた崩壊点検出

本章では、深層学習を用いた崩壊点検出について述べる。前章では崩壊点検出のためのネットワークとして、「1. 飛跡対についてのネットワーク」、「2. 任意の数の飛跡についてのネットワーク」の二つのネットワークを導入した。しかし、これら個々のネットワーク単体では崩壊点検出を実現できないため、これらを組み合わせたアルゴリズムが必要である。そのようなアルゴリズムや前章までのネットワークについての総括を 4.1 節で行う。また、このアルゴリズムではネットワークの出力に対する閾値などの幾つかのパラメータが必要なため、4.2 節では、それらパラメータの最適化について議論する。同時に、どのような評価基準を用いて崩壊点検出の性能を判断するかについても、ここで述べる。最後に 4.3 節では、以上によって実現された崩壊点検出について改めて性能の評価をまとめる。

### 4.1 崩壊点検出アルゴリズム

前章では個々のネットワークについて、ネットワーク間の比較やネットワーク内の評価を行い単体での性能について理解を深めた。「飛跡対についてのネットワーク」では、SVCC・SVBB・TVCC・SVBC の分離は非常に困難であり、崩壊点のタネの段階での個々の secondary vertex の識別は現実的ではないということがわかった。「任意の数の飛跡についてのネットワーク」では、個々の崩壊点の種類 (primary vertex・secondary vertex) に特化したネットワークの性能が標準的なネットワークの性能よりも僅かながら高いということを示した。以上のことから、本研究では崩壊点のタネを primary vertex と secondary vertex に分け、それについて崩壊点の生成を行うこととした。

図 3.8 で示したように飛跡対の全ての組み合わせを考えた場合、そのクラスの比率は NC や PV が支配的なデータとなる。よって、図 3.12 のように NC からの汚染により、SVCC・SVBB・TVCC・SVBC が埋もれてしまうという課題があった。このような NC は、データの構成からほとんどが primary vertex と secondary vertex から一本ずつ選ばれた飛跡で構成されていると考えられる。したがって、この様な汚染は primary vertex の再構成を精度よく行う事によって大幅に削減することが可能である。また、一般に事象中において primary vertex

は必ず一つであり、かつ primary vertex 由来の飛跡の数はその他の崩壊点と比べて多いため、primary vertex から再構成する方が妥当である。これらのこと踏まえ、図 4.1 のような崩壊点検出アルゴリズムを提案する。ここで、「飛跡対についてのネットワーク」として表 3.6 で示したモデル A を用いる。また、「任意の数の飛跡についてのネットワーク」として 3.4.3 項で示した primary vertex や secondary vertex に特化したネットワークをそれぞれの崩壊点の生成に用いる。

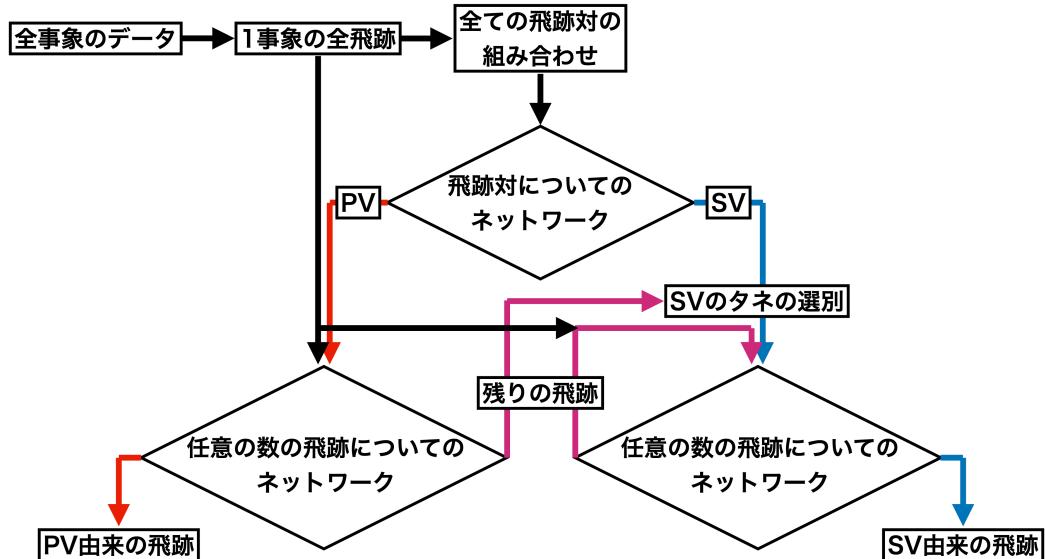


図 4.1: 崩壊点検出アルゴリズム

アルゴリズムは以下の手順で崩壊点の再構成を行う。

1. 全事象から 1 事象分のデータを取り出し、飛跡対の全ての組み合わせを考える。
2. それら飛跡対に対して、「飛跡対についてのネットワーク」を使用し、崩壊点のタネの探索を行う。
3. SVCC・SVBB・TVCC・SVBC と判定された飛跡対について、選別を行い secondary vertex のタネ (SV のタネ) を選ぶ。
4. PV と判定されたタネ (PV のタネ) について、「任意の数の飛跡についてのネットワーク」を用い primary vertex の生成を行う。
5. SV のタネと primary vertex 由来と判定された飛跡の情報を用い、「任意の数の飛跡についてのネットワーク」によって SV のタネが無くなるまで secondary vertex を生成する。

手順 1, 2 については飛跡対についてのネットワークの訓練データの作成や学習と全く同様の手順である。

手順 3 では、「飛跡対についてのネットワーク」によって得られる SVCC・SVBB・TVCC・SVBC のスコアや崩壊点の位置を用いてより純度の高い SV のタネの集合を作成する。したがって, SVCC・SVBB・TVCC・SVBC のスコアについての閾値や崩壊点の位置についての最適化が必要である。

手順 4 では、「飛跡対についてのネットワーク」によって得られる PV のスコアについて降順に並び替えた PV のタネの集合と「任意の数の飛跡についてのネットワーク」を用いて primary vertex を生成する。ここでは, PV のタネをスコアの高い方から幾つ用いるかについて最適化が必要である。また一度以上, 「任意の数の飛跡についてのネットワーク」によって結合していると判定された飛跡を primary vertex 由来であると判断している。この時の「任意の数の飛跡についてのネットワーク」によって得られる崩壊点のタネと結合しているかに関するスコアについての閾値についても, また同様に最適化が必要なパラメータである。

手順 5 について, 手順 3 で選別した SV のタネと手順 4 で得られた primary vertex 由来であると判定された飛跡の一覧を用いて secondary vertex の再構成を行う。ここでは「任意の数の飛跡についてのネットワーク」のデコーダー部に入力する飛跡から, 再構成された secondary vertex 由来の飛跡を取り除いて行くことによって, 再帰的に secondary vertex の生成が行われる。初期状態として使用する SV のタネは, 飛跡対に含まれる飛跡が primary vertex 由来の飛跡の一覧や, それまでに生成した secondary vertex 由来の飛跡の一覧に存在していた場合は除外し, 次の SV のタネへ移行している。この secondary vertex に関する「任意の数の飛跡についてのネットワーク」のスコアも最適化が必要である。また, 再構成された secondary vertex 由来の飛跡の中に primary vertex 由来の飛跡が存在した場合は「任意の数の飛跡についてのネットワーク」によって得られたスコアによって飛跡の争奪が行われる。手順 5 は SV のタネが無くなるまで行われ, 再構成された secondary vertex 由来の飛跡と PV 由来の飛跡以外の飛跡は残りの飛跡とする。

以上が崩壊点検出のためのアルゴリズムである。最適化が必要なパラメータを以下にまとめる。

- 「飛跡対についてのネットワーク」によって得られる SVCC・SVBB・TVCC・SVBC のスコアについての閾値
- 「飛跡対についてのネットワーク」によって得られる崩壊点の位置についての閾値
- 使用する PV のタネの数
- primary vertex の生成に関する「任意の数の飛跡についてのネットワーク」によって得られるスコアについての閾値
- secondary vertex の生成に関する「任意の数の飛跡についてのネットワーク」によって得られるスコアについての閾値

## 4.2 崩壊点検出の最適化と評価手法

崩壊点検出の最適化では、まず「飛跡対についてのネットワーク」によって得られる SVCC・SVBB・TVCC・SVBC のスコアと崩壊点の位置に関する閾値の最適化を行う。次に、使用する PV のタネの数、primary vertex・secondary vertex の生成における「任意の数の飛跡についてのネットワーク」のスコアに関する閾値の最適化を行う。前者は SV のタネの選別に関するパラメータ、後者は崩壊点の生成や崩壊点検出の性能についてのパラメータである。

### 4.2.1 SV のタネの選別

SV のタネを選別する上での評価基準として純度と効率を用いる。ここでは、SV のタネの効率や純度について、SVCC・SVBB・TVCC・SVBC のスコアの和に対する閾値と崩壊点の位置に関する閾値を変化させ最適な値を探る。ただし崩壊点検出アルゴリズムでは、ここから primary vertex 由来である飛跡を含むタネを除外する為、その純度は更に改善されると考えられる。また、SV のタネの選別における Pre-Selection として、「飛跡対についてのネットワーク」によって NC・PV・Others と判断された飛跡対を取り除いている。以上より、SV のタネについての効率と純度を次のように定義する。

$$\text{効率} = \frac{\text{崩壊点の位置} < \text{閾値} \wedge \text{スコアの和} > \text{閾値} \wedge \overline{\text{NC} \cdot \text{PV} \cdot \text{Others}}}{\overline{\text{NC} \cdot \text{PV} \cdot \text{Others}}} \quad (4.1)$$

$$\text{純度} = \frac{\text{崩壊点の位置} < \text{閾値} \wedge \text{スコアの和} > \text{閾値} \wedge \overline{\text{NC} \cdot \text{PV} \cdot \text{Others}}}{\text{崩壊点の位置} < \text{閾値} \wedge \text{スコアの和} > \text{閾値}}$$

ここで崩壊点の位置についての閾値は上限、スコアの和についての閾値は下限となっている。これは、より遠くの飛跡対やスコアの低い飛跡対を除外する為である。閾値と SV のタネの効率、純度の関係を図 4.2 に示す。

効率は崩壊点の位置についての閾値をより大きくすることによって改善されている。これは secondary vertex についての崩壊点の位置が 1 – 10 mm の間でピークを持つことから理解できる。一方、スコアについての閾値は位置と比較して変化が小さく、0.8 度度から徐々に効率が減少している。このことから、SVCC・SVBB・TVCC・SVBC のスコアの和は比較的大きな値を持っていることが理解できる。純度については、崩壊点の位置についての閾値が小さく、かつスコアについての閾値が大きな領域においては選択された SV のタネが存在しなかつた為、0 としている。また、全体の傾向として、崩壊点の位置についての閾値が一定で、スコアについての閾値が大きい領域で純度が高くなっている。これは、前述したような崩壊点の位置の特性や深層学習のスコアから明らかな性質である。

最適なパラメータの設定の探索のため、Precision-Recall (PR) 曲線を使用する (図 4.3)。ここで、Precision は純度、Recall は効率である。図 4.2 からも明らかな様に効率は崩壊点の位置についての閾値が 1 mm の辺りで急速に減少している。したがって、崩壊点の位置についての

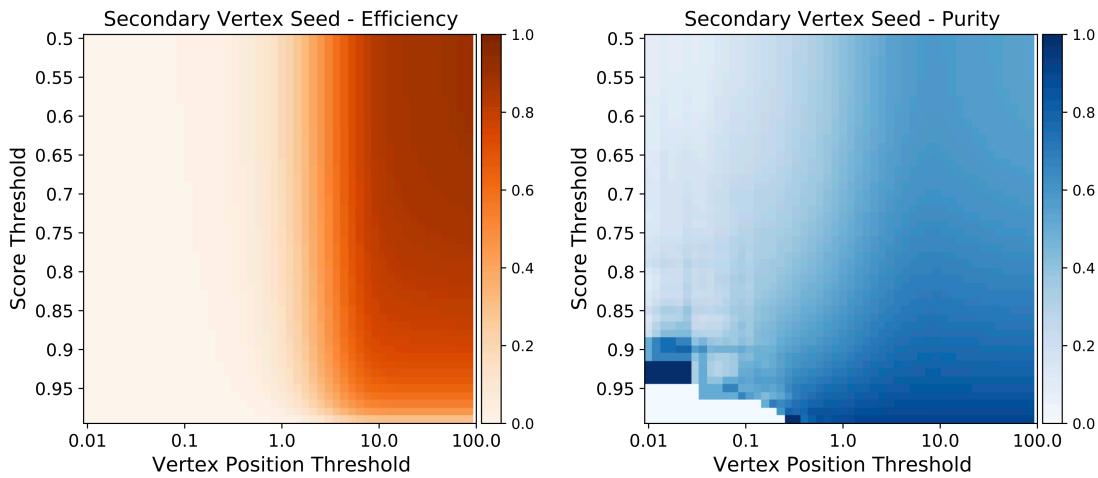


図 4.2: 閾値と SV のタネの効率、純度の関係。図左が効率についての関係、図右が純度についての関係である。縦軸は SVCC・SVBB・TVCC・SVBC のスコアの和に対しての閾値、横軸は崩壊点の位置についての閾値を示している。また、横軸については対数スケールを用いている。

閾値が 1 mm 以上の場合についてのみ考えることとする。また、性能の最大値として効率と純度の和が最大となるパラメータの組み合わせを用いる。これは ROC 曲線の際の性能の決定方法に倣ったものである。このパラメータは以降の崩壊点の生成や崩壊点検出の性能、現行の手法との比較に用いる。

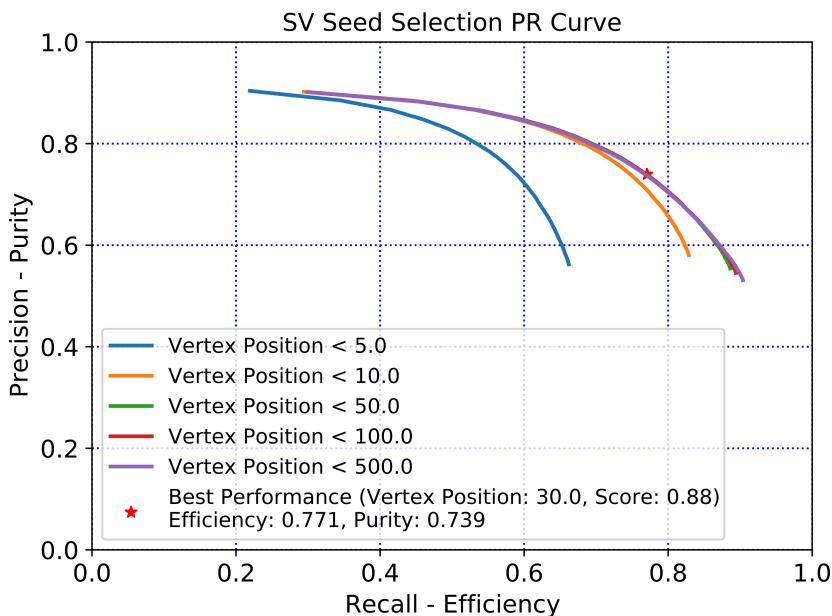


図 4.3: 閾値と SV のタネの効率、純度についての PR 曲線。縦軸は純度、横軸は効率である。崩壊点の位置についての閾値を固定し、スコアについての閾値を変化させている。崩壊点の位置について 5.0, 10.0, 50.0, 100.0, 500.0 mm のデータのみ描画している。

図 4.3 では、崩壊点の位置については 5.0, 10.0, 50.0, 100.0, 500.0 mm のデータのみ描画しているが、実際には 1.0 – 9.0 まで 1 刻みずつ, 10.0 – 90.0 まで 10 刻みずつ 100 – 900 まで 100 刻みずつのデータについて最適なパラメータの探索を行った。また、スコアについては 0.50 – 0.99 まで 0.01 刻みでの探索を行った。崩壊点の位置についての閾値は 50 度から殆ど変化はなく、それ以上の離れた位置に SV のタネがほとんど存在していない事がわかる。

### 4.2.2 崩壊点の生成

崩壊点の生成や崩壊点検出の性能についてのパラメータを最適化する為、その性能を定める評価項目を定義する必要がある。そのような評価項目として secondary vertex の再構成について飛跡段階での効率を用いる。まず以下の様な評価項目を設ける。

- Primary : MC 情報によって primary vertex 由来であるとラベルされた飛跡
- Bottom : MC 情報によってボトム・フレーバーの secondary vertex 由来であるとラベルされた飛跡
- Charm : MC 情報によってチャーム・フレーバーの secondary vertex 由来であるとラベルされた飛跡
- Others : MC 情報によって Others であるとラベルされた飛跡

これらについてネットワークがどの程度 secondary vertex と判断してしまったかを評価する。Primary や Others は secondary vertex ではない為、値が小さいほど性能が高く secondary vertex の純度が高いと判断する。Bottom や Charm は secondary vertex である為、値が大きいほど性能が高く secondary vertex の効率が高いと判断する。更に、Bottom や Charm に関しては以下の様な複数個の secondary vertex を跨いで飛跡を獲得した場合の罰則を考慮する。

- 同一の崩壊チェイン : 同一の崩壊チェイン由来の飛跡のみで構成されているか
- 同一の親粒子 : 同一の親粒子由来の飛跡のみで構成されているか

ここでは、ボトム・フレーバー、チャーム・フレーバーの secondary vertex の内、同じボトム・ハドロンから生じた secondary vertex の組みを崩壊チェインと呼び、更に細かく個々のフレーバーのハドロンを親粒子と呼んでいる（図 4.4）。同一の崩壊チェイン、同一の親粒子とはこれらの崩壊チェインや親粒子を跨いで飛跡を選択しているかどうかを判断している。

これらの評価項目は前述した現行の手法である LCFIPlus での評価項目と同じものである。崩壊点の生成についてのパラメータは使用する PV のタネの数、primary vertex · secondary vertex の生成におけるネットワークのスコアの閾値の三つである。それぞれ三つのパラメータについて、PV のタネの数を 1 から 3 個まで、primary vertex · secondary vertex のスコアの閾値を 0.50 から 0.95 まで 0.05 ずつ変化させ、各評価項目の値を確認する。データは 100 事象分の飛跡を用いた。PV のタネを一つ使用した場合の結果を図 4.5 に示す。

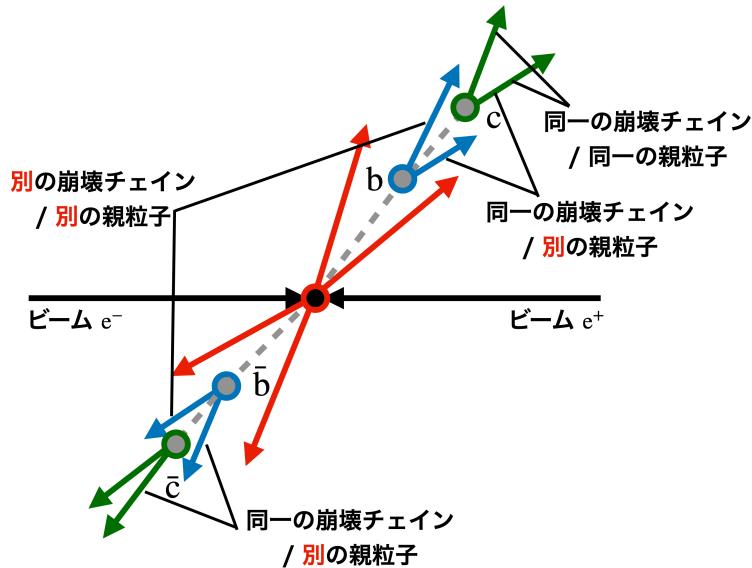


図 4.4: 同一の崩壊チェインと同一の親粒子

ここで、次章の評価の為、現行の手法である LCFIPlus の性能値より値の大きいものの数値を色付けしている。secondary vertex の生成でのネットワークのスコアの閾値について同一の親粒子に関する性能を見ると、閾値が大きいほど値が上昇し 0.85 付近で減少へ転じていることが分かる。一方、Bottom や Charm とその同一の崩壊チェインについての性能は閾値が大きいほど値が減少している。したがって、小さな閾値では一つの secondary vertex に対して様々な飛跡が結合してしまい、結果として、同一の親粒子での性能が低下していると考えられる。ただし、同一の崩壊チェインについては常に数 % 程度の差異で追従しており、本研究のネットワークが正常に動作していると理解できる。

Primary や Others は非常に値が小さく、Primary は primary vertex · secondary vertex の生成でのネットワークのスコアの閾値の両方に感度を持っていることが分かる。これは primary vertex の生成に対して高い閾値を設けた場合、取り逃がした真の primary vertex 由来の飛跡が secondary vertex 内に混入してしまうことを示している。Others については、primary vertex の生成に対しての閾値について大きな影響を受けていない。このことから primary vertex は純度が高く Others な飛跡をほとんど含んでいないと考えられる。

### 4.3 崩壊点検出の性能

本章では、現行の手法である LCFIPlus の評価項目に則って深層学習を用いた崩壊点検出の性能を調査した。これは次章の LCFIPlus との比較を行う為である。

最後にデータサンプル  $b\bar{b} - 08$ ,  $c\bar{c} - 08$  での全事象を用いた性能をそれぞれ表 4.1, 4.2 に示す。ここでは、前節で最適化を行った閾値の値として以下の組みを使用した。ただし、これらの性能値は 5 章で追加されるアルゴリズムを含んでいない。したがって、崩壊点検出単体での

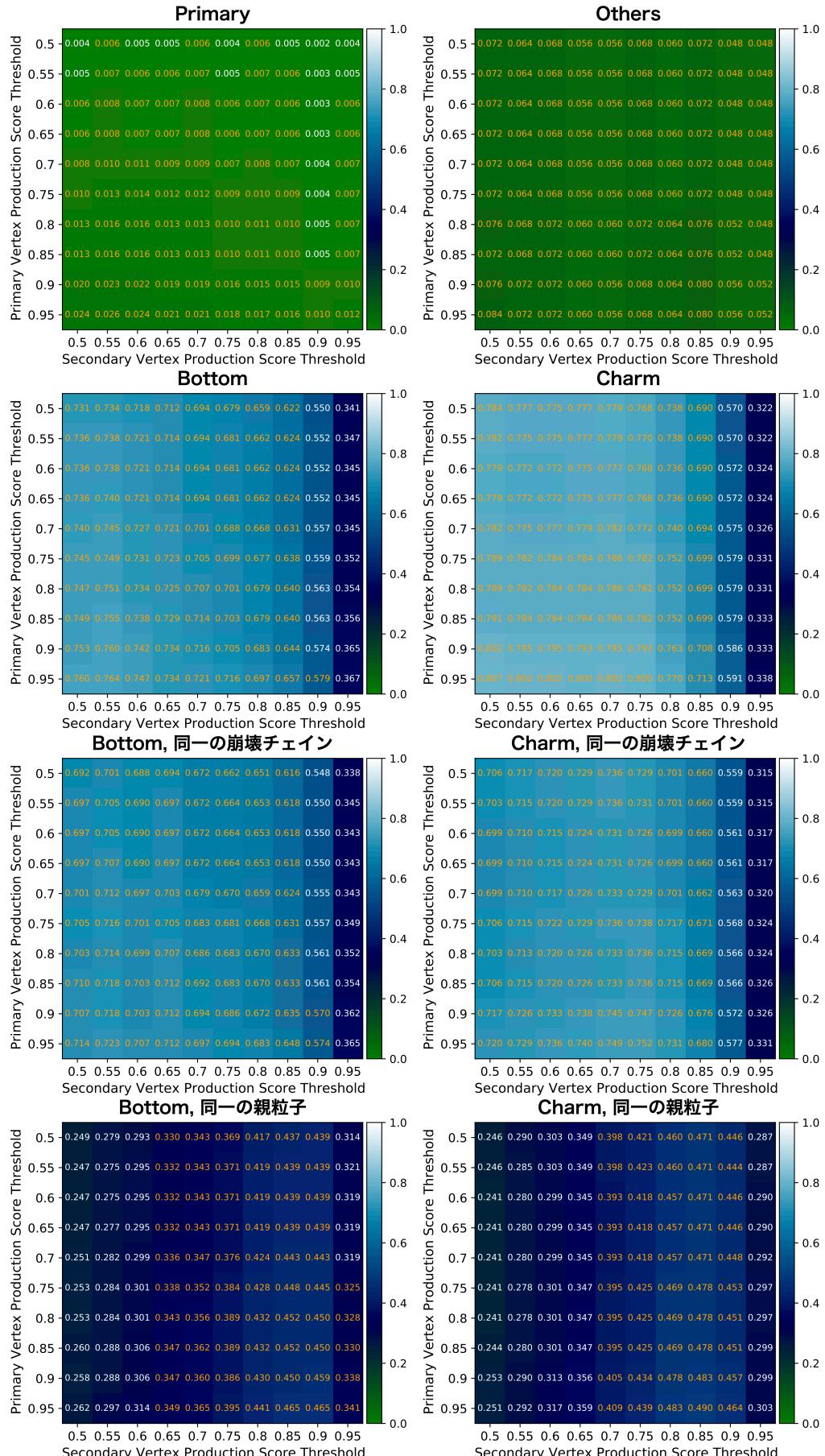


図 4.5: 閾値と崩壊点検出の性能の関係。全ての図について縦軸・横軸は primary vertex · secondary vertex のスコアの閾値である。Primary, Others, Bottom, Charm はそれぞれの飛跡がネットワークによって secondary vertex と判断された割合を示している。同一の崩壊チェイン, 同一の親粒子は再構成された secondary vertex 内に異なる崩壊チェインや親粒子由来の飛跡を含んでいないか評価している。

性能値であり、他のアルゴリズムと連結された場合については、この限りではない点に留意しなければならない。

- SV のタネの選別 - 飛跡対についてのネットワーク
  - SVCC・SVBB・TVCC・SVBC のスコアの和 : 0.88
  - 崩壊点の位置 : 30.0 mm
- 崩壊点の生成 - 任意の数の飛跡についてのネットワーク
  - PV のタネの数 : 3
  - primary vertex の生成に関するスコア : 0.50
  - secondary vertex の生成に関するスコア : 0.75

表 4.1 より、事象数を増やした場合においても本研究で作成した崩壊点検出アルゴリズムがおよそ 65 – 75% 程度の割合で secondary vertex を再構成できていることがわかる。更に、それら secondary vertex について 35 – 40% 程度の割合で各崩壊点の分離に成功している。一方で、Primary, Others については数 % 程度に抑えられ、純度に関しても非常に高い結果を得られた。

表 4.2 では、データサンプル  $b\bar{b} - 08$  での結果と比較して Primary の値が悪化している。これは図 3.2 からもわかる様に、終状態  $c\bar{c}$  では終状態  $b\bar{b}$  と比較して primary vertex 由来の飛跡の数が多いためである。

表 4.1: データサンプル  $b\bar{b} - 08$  を用いた性能

真の飛跡の種類	Primary	Bottom	Charm	Others
全飛跡の数	307649	167161	152314	86225
secondary vertex と判定された飛跡の割合	1.2%	66.8%	74.7%	6.9%
... 同一の崩壊チェイン	-	64.8%	69.1%	-
... 同一の親粒子	-	37.9%	40.5%	-

表 4.2: データサンプル  $c\bar{c} - 08$  を用いた性能

真の飛跡の種類	Primary	Charm	Others
全飛跡の数	491035	147449	93681
secondary vertex と判定された飛跡の割合	1.5%	59.0%	4.8%
... 同一の親粒子	-	55.7%	-

## 第 5 章

# 現行の手法 (LCFIPlus) との比較

本研究での崩壊点検出の性能の可否についての判断は現行の手法と比較することによって行う。本章では、そのような比較について議論を行う。まず 5.1 節では崩壊点検出単体での性能を比較する。

崩壊点検出はジェットの再構成における最初段のアルゴリズムである。したがって、最終的な物理解析への影響や性能を確認するには、本研究で作成した崩壊点検出を用いたフレーバー識別の性能を評価する必要がある。このようなフレーバー識別の性能を確認する為、本研究で作成した崩壊点検出アルゴリズムを LCFIPlus, Marlin で動作可能な形で実装した。5.2 節ではフレーバー識別を用いたより詳細な比較について述べる。

### 5.1 崩壊点検出単体での比較

本研究における崩壊点検出の性能を表 4.1 に示した。表 5.1 に LCFIPlus で使用されている崩壊点検出の性能である文献 [9] の値を示す。これは本研究の目標値である。4.2.2 項でも述べたように、この表にあるそれぞれの割合は Primary や Others については低く、Bottom や Charm については高ければ良い性能であるとみなせる。崩壊点検出での比較では、本研究の崩壊点検出は LCFIPlus に比べ、Bottom や Charm の再構成効率が 10% 程度向上しているとわかる。一方で、0.5 より高い閾値を「任意の数の飛跡についてのネットワーク」の出力に設け、より純度の高い secondary vertex の生成を行っているが、Primary や Others に関しては誤識別の割合が上昇してしまっている。LCFIPlus では Bottom や Charm の効率と同一の崩壊チェイン由来の差は 1% 程度であるのに対し、本研究の性能では 2 – 5% 程度の差ができる。このことから、本研究の崩壊点検出アルゴリズムは LCFIPlus と比較して別の崩壊チェインの飛跡を獲得しやすい傾向にあることがわかる。Others の飛跡の数について、他の項目と比較した時の割合が LCFIPlus と本アルゴリズムでおよそ 2.5 倍程度異なっている。これは Others の定義によるものであると考えれるが、本研究の解析において具体的な原因を突き止めるに至らなかった。

崩壊点検出アルゴリズムの性能を一意に決めるのは容易ではないため、より詳細な比較はジェット再構成アルゴリズムの最後段であるフレーバータギングを用いて行うべきである。

表 5.1: LCFIPlus での性能値

真の飛跡の種類	Primary	Bottom	Charm	Others
全飛跡の数	496897	258299	247352	56432
secondary vertex と判定された飛跡の割合	0.6%	57.5%	64.3%	2.5%
... 同一の崩壊チェイン	-	56.6%	63.4%	1.9%
... 同一の親粒子	-	32.2%	38.9%	1.2%

## 5.2 より詳細な比較

本研究のネットワークは Tensorflow/Keras によって構築されており、それらは Python を用いて記述されている。しかし LCFIPlus は前述したように Marlin のプロセッサーであり C++ で動作しているため、LCFIPlus 内のフレーバータギングの使用には Python 環境から C++ 環境への移植が必要である。本研究では Tensorflow C++ API を用いて作成したネットワークを C++ 上で動作させ、本研究の崩壊点検出アルゴリズムを LCFIPlus 内のアルゴリズムとして実装した。したがって、本アルゴリズムは LCFIPlus のアルゴリズムと容易に置換でき、また Tensorflow/Keras で作成したネットワークであれば簡単な手順によって導入が可能である。使用したソフトウェアの環境を次の表 5.2 にまとめる。詳細な実装方法については著者の GitHub にまとめている [41]。

フレーバータギングでの比較にあたって性能を対照的に評価するため、ここでは LCFIPlus の崩壊点検出アルゴリズムのみを深層学習を用いた手法に置き換え、その他の LCFIPlus アルゴリズムについては同様の設定のものを使用した。このため、他の LCFIPlus アルゴリズムとの整合性を保つためアルゴリズムの改変が必要となった。本アルゴリズムに加えた新たなアルゴリズムについては 5.2.1 項で述べる。フレーバータギングの性能について 5.2.2 項で述べ比較を行う。

LCFIPlus の性能として前節では文献値 [9] を用いたが、ここではデータの量やアルゴリズムの詳細なパラメータを一致させるため文献と同様の手順で再度ジェット再構成を行い、得られた性能値と比較する。

表 5.2: 崩壊点検出のソフトウェア動作環境

ソフトウェア	バージョン
Bazel	0.29.1
Tensorflow C++ API	2.1.0
CUDA	10.1
cuDNN	7
Eigen	3.3.90
Protobuf	3.8
g++	8.4.0
iLCSoft	02 – 02

### 5.2.1 アルゴリズムの修正

LCFIPlus では、後段のアルゴリズムに前段のアルゴリズムからの情報を使用しており、特にジェットクラスタリングでは LCFIPlus の崩壊点検出アルゴリズムによって得られる崩壊点の情報を使用している。LCFIPlus の崩壊点検出アルゴリズムを本研究で作成したアルゴリズムに置換した場合においても、この崩壊点の情報をジェットクラスタリングへ提供しなければならない。したがって、本アルゴリズムによって生成された崩壊点についても再度 LCFIPlus のフィッターを用いた情報の抽出が必要である。しかし、本研究の崩壊点検出アルゴリズムではその性質上、飛跡が一本しか含まれていない secondary vertex が生成されてしまう場合があり、この様な「単独の飛跡」について、LCFIPlus のフィッターは上手く動作しないため何らかの処理が必要である。更に、ジェットクラスタリング以降は LCFIPlus のフィッターで得られた情報を信用しており、フィッターによって生成された崩壊点 ( $\chi^2$  値が正常な崩壊点) であることを前提にしている。以上の都合により、4.1 項で作成した崩壊点検出アルゴリズムの後に以下の手順を追加した。

1. LCFIPlus のフィッターによって、深層学習で予想される secondary vertex が正常な崩壊点であるか評価し、正常でない崩壊点を単独の飛跡に分解する。
2. 単独の飛跡について、深層学習で予想された正常な secondary vertex や他の単独の飛跡と結合するかどうかを LCFIPlus のフィッターによって評価する。
3. 生成された崩壊点について、LCFIPlus のフィッターを用いて情報の抽出を行う。

手順 1 によって本アルゴリズムが再構成した崩壊点に正常であるか評価を行い、異常であった場合は単独の飛跡に分解される。この評価は LCFIPlus のフィッターで得られる  $\chi^2$  値を用いている。手順 2 では単独の飛跡と深層学習で予想される secondary vertex が結合するかを LCFIPlus のフィッターを用いることで評価している。手順 3 では生成された崩壊点について、LCFIPlus のフィッターを用いてジェットクラスタリングのための情報を抽出している。

以上の手順をアルゴリズムに追加し崩壊点の探索を行う。手順 1, 2 で使用した各パラメータの閾値を次に示す。これらの閾値は LCFIPplus で使用している値を参考にした。特に手順 2 では LCFIPplus と同等の選別を行っている。

ただし, primary vertex や secondary vertex の飛跡を選ぶ段階では純粹に深層学習のみを使用している事に言及しておく。ここで加えられた手順は崩壊点検出アルゴリズムの後のジェット再構成アルゴリズムの為の選別であり, 品質の悪い (ヒット数の少ない飛跡再構成についての誤差の大きな) 飛跡を含んだ崩壊点が後段のアルゴリズムへ影響しない様にする為の処理である。

- 手順 1
  - 最も  $\chi^2$  値への寄与の大きい飛跡の  $\chi^2$  値の上限 : 9
- 手順 2
  - 最も  $\chi^2$  値への寄与の大きい飛跡の  $\chi^2$  値の上限 : 9
  - 結合された崩壊点の質量の上限 : 10 GeV

### 5.2.2 本研究の崩壊点検出を用いたジェット再構成の性能

崩壊点検出アルゴリズムの後のジェット再構成アルゴリズムについては, 参考文献 [9] に倣った設定値を用いた。まず, jet vertex refiner アルゴリズム (1.5.3 項) によって再構成された崩壊点と疑似崩壊点の情報を表 5.4 にまとめる。括弧内は LCFIPplus の性能との相対値である。

表 5.3: (差し替え) 再構成された崩壊点と疑似崩壊点の個数

(崩壊点, 疑似崩壊点)	ボトム ( $b$ )	チャーム ( $c$ )	アップ, ダウン, ストレング ( $uds$ )
(0, 0)	12.7% (-8.6)	49.1% (-10.2)	93.8% (-4.3)
(0, 1)	1.25% (-0.36)	0.41% (-2.4)	0.10% (0.9)
(1, 0)	47.9% (10.2)	48.1% (8.3)	5.96% (4.16)
(1, 1)	9.41% (-4.1)	0.90% (0.36)	0.03% (0.01)
(2, 0)	28.7% (4.9)	1.46% (1.27)	0.07% (0.03)

LCFIPplus と比較して本研究の崩壊点検出は疑似崩壊点ではなく崩壊点による項目の値が大きくなってしまっており, 崩壊点検出により, より多くの崩壊点の候補を発見しているとわかる。一方, アップ, ダウン, ストレング・フレーバーのジェット ( $uds$ ・ジェット) に関しても偽の崩壊点を発見してしまっており, フレーバー識別の性能低下につながる可能性がある。

フレーバータギングでは, ジェットに含まれる崩壊点や疑似崩壊点の個数によって以下の 4 つのカテゴリにデータを分離した。

フレーバータギングの性能は ROC 曲線を用いた評価を行う。ここではボトム・ジェット ( $b$ ・ジェット) についての識別効率とチャーム・ジェット ( $c$ ・ジェット) についての識別効率

表 5.4: フレーバータギングのカテゴリー

カテゴリー	1	2	3	4
崩壊点の個数	0	1	1	2
疑似崩壊点の個数	0-2	0	1	0

について、それぞれ ROC 曲線を描画した。図 5.1 では横軸に信号効率、縦軸に対数スケールの背景効率を示している。 $b$ ・ジェットの識別に関する背景事象は  $c$ ・ジェット,  $uds$ ・ジェットが考えられ、 $c$ ・ジェットの識別については  $b$ ・ジェット,  $uds$ ・ジェットが考えられる。

残念ながら、本研究で作成した崩壊点検出アルゴリズムに置換した場合の LCFIPlus のフレーバー識別性能は、LCFIPlus のアルゴリズムと比較して大幅に悪化してしまった。これは深層学習では、品質の悪い飛跡を含んでしまう事が要因の一つとして考えられる。通常、LCFIPlus のフィッターによって生成される secondary vertex では、誤差の大きな飛跡は取り除かれ影響は僅かであると考えられるが、深層学習においては、この様な誤差の取り扱いは非常に困難である。更に、LCFIPlus のジェット再構成では LCFIPlus のフィッターで得られた情報を後段のアルゴリズムに使用する為、LCFIPlus のフィッターで得られた情報は必ず正常である事が要求される。従って、LCFIPlus のフィッターで得られない様な飛跡を深層学習によって再構成できたとしても、その様な飛跡に対して LCFIPlus のフィッターはうまく動作しない為、後の解析において雑音になり得る。以上の要因から本アルゴリズムは LCFIPlus でのフレーバータギングでは性能が悪化してしまったと考えられる。

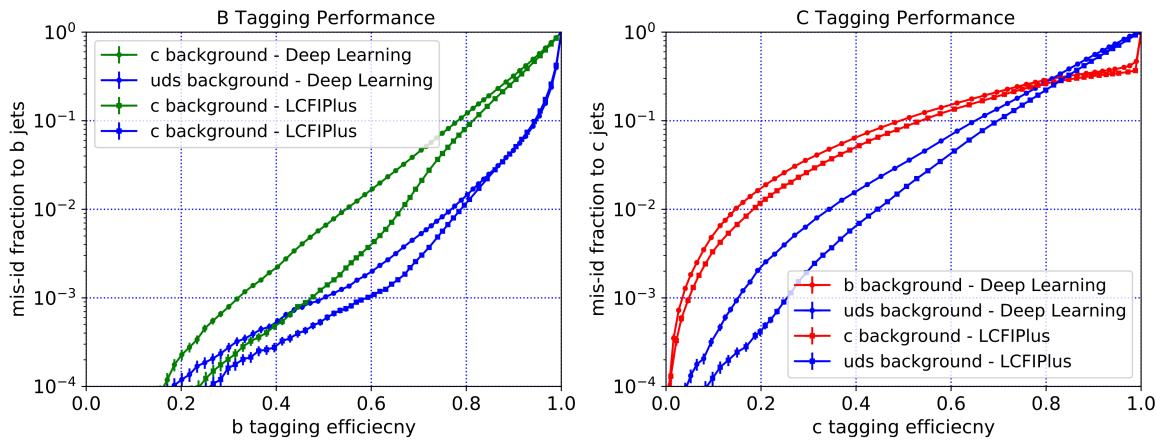


図 5.1: (差し替え) フレーバー識別の性能に関する ROC 曲線。左図が  $b$ ・ジェットについての識別効率、右図が  $c$ ・ジェットについての識別効率を示している。それぞれ縦軸が背景効率、横軸が信号効率である。赤が  $b$ ・フレーバー、緑が  $c$ ・フレーバー、青が  $uds$ ・フレーバーの結果を示している。

実際に LCFIPlus と同等の閾値に基づく選別を 5.2.1 項の手順 1 に加え、品質の悪い飛跡を事前に取り除いた場合の結果では性能が改善していることが分かる（図 5.2）。ここでは、より多くの飛跡を獲得できる様にする為、「任意の数の飛跡についてのネットワーク」に関する閾

値を以下の様に変更している。

図5.2では、 $c \cdot$  ジェットについての識別において LCFIPplus と同等の性能となっている事がわかる。

- 崩壊点の生成 - 任意の数の飛跡についてのネットワーク
  - PV のタネの数 : 3
  - primary vertex の生成に関するスコア : 0.50
  - secondary vertex の生成に関するスコア : 0.50
- ヒット数, 誤差, エネルギーによる飛跡の選別
- 手順 1
  - 最も  $\chi^2$  値への寄与の大きい飛跡の  $\chi^2$  値の上限 : 9
  - 結合された崩壊点の質量の上限 : 10 GeV
  - 崩壊点中の飛跡の合成運動量がビーム衝突点方向ではない

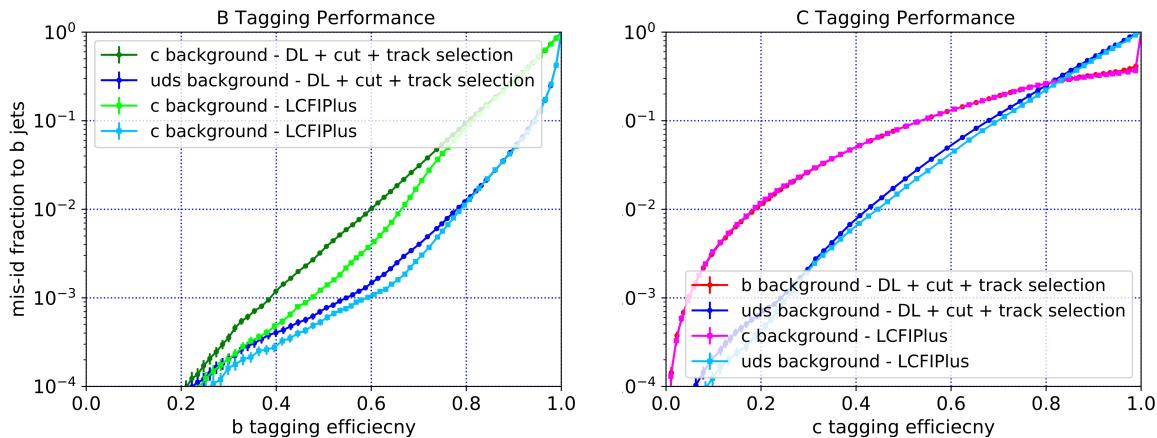


図5.2: LCFIPplus と同等の選別を行った後の性能。左図がボトム・ジェットについての識別効率、右図がチャーム・ジェットについての識別効率を示している。それぞれ縦軸が背景効率、横軸が信号効率である。赤が  $b \cdot$  フレーバー、緑が  $c \cdot$  フレーバー、青が  $uds \cdot$  フレーバーの結果を示している。

## 第 6 章

# まとめと今後の展望

本研究では、深層学習を用いた崩壊点検出アルゴリズムの開発と実装を行った。ジェットの再構成は、加速器実験における新物理の手掛かりを探索する上で非常に重要な役割を果たしている。ILCにおいては、ボトム・フレーバーとチャーム・フレーバーのジェットの高性能な分離が期待され、現在は LCFIPlus のジェット再構成アルゴリズムが使用されている。事象再構成アルゴリズムにはパターン認識技術が使用される場合も多く、深層学習のような機械学習技術の発展による物理解析の精度向上が期待されている。しかし、崩壊点検出アルゴリズムについては深層学習を用いたアルゴリズムの研究は殆ど行われておらず、本研究がその先駆けとなっている。

深層学習技術として一般的なフィードフォワードニューラルネットワークに加え、リカレントニューラルネットワークの一種である LSTM を独自に拡張したネットワークや Attention など最新のテクニックを活用した。崩壊点検出アルゴリズムの実現に当たって二種類のネットワーク「1. 飛跡対についてのネットワーク」、「2. 任意の数の飛跡についてのネットワーク」を Tensorflow/Keras を用いて実装した。ネットワークのための訓練データとして  $e^+e^- \rightarrow b\bar{b}/c\bar{c}$  事象の MC シミュレーションデータを使用し、正解ラベルの作成には MC 情報や LCFIPlus のフィッターによる計算値を用いた。

これらのネットワークを組み合わせたアルゴリズムを作成し、性能の評価を飛跡段階での効率を用いて行った。性能の可否については LCFIPlus と比較する事によって評価し、まず崩壊点検出アルゴリズム単体での比較を行った後、ジェット再構成の最後段であるフレーバータギングでの比較を行った。フレーバータギングの比較のため、本ネットワーク・アルゴリズムの LCFIPlus, Marlin への実装を行った。本アルゴリズムは LCFIPlus のアルゴリズムと置換可能な状態であり、iLCSoft 内での深層学習の活用について示すことができた。

崩壊点検出アルゴリズム単体での比較において、本研究のアルゴリズムは secondary vertex の再構成効率において LCFIPlus のアルゴリズムを超えることができた（表 4.1）。一方で、誤識別である Primary, Others の値が悪化してしまっており単純な性能比較では性能の優劣を決めることが出来なかった。

フレーバータギングの比較において、本研究で作成した崩壊点検出アルゴリズムは LCFI-

Plus の性能を超える事ができなかった。これは、深層学習で再構成された崩壊点に誤差の大きな飛跡が混入してしまい雑音となってしまうからであると考えられる。深層学習ではその様な誤差の情報を上手く取り入れる事が困難である為、適切な損失関数やネットワークの作成が必要であるが、これらの研究は今後の課題としたい。

今後の展望として、崩壊点検出アルゴリズムについては 2 点改善の余地が存在するを考えている。

一つは深層学習の理解からの改良である。例えば、現在、「任意の飛跡についてのネットワーク」はエンコーダー・デコーダー部に対してリカレントニューラルネットワークの技術を使用しているが、リカレントニューラルネットワークは並列化しづらく、学習・推論に時間がかかるという課題がある。特にエンコーダー部にはより並列処理に互換性のある構造を取り入れることでより高速な計算が可能になると想定している。更に、本研究では飛跡について空間方向の依存性を殆ど取り入れられていない為、「畳み込みニューラルネットワーク」や「グラフニューラルネットワーク」などを活用する事によって改善が期待できる。

もう一つは物理的な性質からの改良である。これは損失関数やネットワークの構造に対して、エネルギーなどの物理的性質を組み込む事による改善である。本ネットワークはその様な物理的性質からの学習の方向付けを行なっておらず、事象についての情報を十全に抽出できていなかつたのではないかと考えられる。

本研究では、残念ながらフレーバータギングの性能において従来の手法を上回ることが出来なかつたが、これは深層学習としての利点を十分に活かせていない事もその要因として考えられる。深層学習では中間層の情報の使用や転移学習といった様々な手段によって後段のアルゴリズムに情報を提供することが出来る。今回の比較においては、深層学習は単に推論にのみ使いし、この様な情報は全て捨ててしまっている為、深層学習の真の価値を活かし切れたとは言いづらく、後段のアルゴリズムを含めた全てのジエット再構成アルゴリズムを深層学習や機械学習に置き換える事でこれらの情報を活用することができると言えている。

# 謝辞

# 付録 A

## ソースコード

ここでは本研究で使用したコードについて記述する。ここで紹介する実装方法は Tensorflow/Keras を用いた一例である。実装の解説については要点のみ述べる。詳細なコードについては参考文献 [36] を参照して頂きたい。

### A.1 任意の数の飛跡についてのネットワーク

#### A.1.1 独自のリカレントニューラルネットワーク

まず、3.4.3 で比較した 3 つのネットワークについて、使用したコードを記述する。

**Listing A.1:** 標準的な LSTM

---

```

1 def LSTMModelSimple(pair, tracks, UNITS=256):
2
3     MAX_TRACK_NUM = tracks.shape[1]
4     INPUT_DIM = tracks.shape[2]
5     PAIR_DIM = pair.shape[1]
6
7     pair_input = Input(shape=(PAIR_DIM,), name='Pair_Input')
8
9     track_input = Input(shape=(None, INPUT_DIM), name='Input')
10    track_embedd = TimeDistributed(Dense(UNITS, name='Embedding_Dense',
11                                         activation='relu'))(track_input)
12
13    init_state = Dense(UNITS, name='Init_State_Dense_1')(pair_input)
14    init_state = BatchNormalization(name='Init_State_BatchNorm_1')(init_state)
15    init_state = Activation('relu', name='Init_State_Activation_1')(init_state)
16    init_state = Dense(UNITS, name='Init_State_Dense_2')(init_state)
17    init_state = BatchNormalization(name='Init_State_BatchNorm_2')(init_state)
```

```
17     init_state = Activation('relu', name='Init_State_Activation_2')(init_state)
18
19     rnn = LSTM(UNITS, return_sequences=True, name='LSTM_Simple')(track_embedd,
20                   initial_state=[init_state, init_state])
21
22     rnn = TimeDistributed(Dense(1, name='Last_Dense', activation='relu'))(rnn)
23
24     model = Model(inputs=[pair_input, track_input], outputs=rnn)
25
26     model.summary()
27
28     return model
```

Listing A.2: 独自の LSTM

```
1 def VLSTMModelSimple(pair, tracks, UNITS=256):
2
3     MAX_TRACK_NUM = tracks.shape[1]
4     INPUT_DIM = tracks.shape[2]
5     PAIR_DIM = pair.shape[1]
6
7     pair_input = Input(shape=(PAIR_DIM,), name='Pair_Input')
8
9     track_input = Input(shape=(None, INPUT_DIM), name='Input')
10    track_embedd = TimeDistributed(Dense(UNITS, name='Embedding_Dense',
11                                    activation='relu'))(track_input)
12
13    init_state = Dense(UNITS, name='Init_State_Dense_1')(pair_input)
14    init_state = BatchNormalization(name='Init_State_BatchNorm_1')(init_state)
15    init_state = Activation('relu', name='Init_State_Activation_1')(init_state)
16    init_state = Dense(UNITS, name='Init_State_Dense_2')(init_state)
17    init_state = BatchNormalization(name='Init_State_BatchNorm_2')(init_state)
18    init_state = Activation('relu', name='Init_State_Activation_2')(init_state)
19
20    cell = layers.VLSTMCellSimple(UNITS, 1)
21
22    rnn = RNN(cell, return_sequences=True, name='Vertex_LSTM_Simple')(
23                  track_embedd, initial_state=[init_state, init_state])
24
25    model = Model(inputs=[pair_input, track_input], outputs=rnn)
26
27    model.summary()
```

```
26  
27     return model
```

---

Listing A.3: 独自の Attention LSTM

---

```
1 def AttentionVLSTMModel(pair, tracks, ENCODER_UNITS=256, DECODER_UNITS=256):  
2  
3     MAX_TRACK_NUM = tracks.shape[1]  
4     INPUT_DIM = tracks.shape[2]  
5     PAIR_DIM = pair.shape[1]  
6  
7     pair_input = Input(shape=(PAIR_DIM,), name='Pair_Input')  
8  
9     encoder_input = Input(shape=(MAX_TRACK_NUM, INPUT_DIM), name='Encoder_Input'  
10        )  
11  
12    encoder_embedd = TimeDistributed(Dense(ENCODER_UNITS, name='  
13        Encoder_Embbedding_Dense', activation='relu'))(encoder_input)  
14  
15    encoder_init_state_f = Dense(ENCODER_UNITS, name='Encoder_Forward_Dense_1')(  
16        pair_input)  
17    encoder_init_state_f = BatchNormalization(name='Encoder_Forward_BatchNorm_1')(  
18        encoder_init_state_f)  
19    encoder_init_state_f = Activation('relu', name='Encoder_Forward_Activation_1')(  
20        encoder_init_state_f)  
21    encoder_init_state_f = Dense(ENCODER_UNITS, name='Encoder_Forward_Dense_2')(  
22        encoder_init_state_f)  
23    encoder_init_state_f = BatchNormalization(name='Encoder_Forward_BatchNorm_2')(  
24        encoder_init_state_f)  
25    encoder_init_state_f = Activation('relu', name='Encoder_Forward_Activation_2')(  
26        encoder_init_state_f)  
27  
28    encoder_init_state_b = Dense(ENCODER_UNITS, name='Encoder_Backward_Dense_1')(  
29        pair_input)  
30    encoder_init_state_b = BatchNormalization(name='Encoder_Backward_BatchNorm_1')(  
31        encoder_init_state_b)  
32    encoder_init_state_b = Activation('relu', name='Encoder_Backward_Activation_1')(  
33        encoder_init_state_b)
```

```
25     encoder_init_state_b = Dense(ENCODER_UNITS, name='Encoder_Backward_Dense_2')(
26         encoder_init_state_b)
27     encoder_init_state_b = BatchNormalization(name='Encoder_Backward_BatchNorm_2',
28         )(encoder_init_state_b)
29     encoder_init_state_b = Activation('relu', name='Encoder_Backward_Activation_2',
30         )(encoder_init_state_b)
31
32     decoder_init_state = Dense(DECODER_UNITS, name='Decoder_Dense_1')(pair_input)
33     denoder_init_state = BatchNormalization(name='Decoder_BatchNorm_1')(
34         decoder_init_state)
35     decoder_init_state = Activation('relu', name='Decoder_Activation_1')(
36         decoder_init_state)
37     decoder_init_state = Dense(DECODER_UNITS, name='Decoder_Dense_2')(
38         decoder_init_state)
39     denoder_init_state = BatchNormalization(name='Decoder_BatchNorm_2')(
40         decoder_init_state)
41     decoder_init_state = Activation('relu', name='Decoder_Activation_2')(
42         decoder_init_state)
43
44     vlstm_cell_f = layers.VLSTMCellEncoder(ENCODER_UNITS)
45     vlstm_cell_b = layers.VLSTMCellEncoder(ENCODER_UNITS)
46     encoder_f = RNN(vlstm_cell_f, return_sequences=True, name='
47         Encoder_Forward_VLSTM', go_backwards=False)
48     encoder_b = RNN(vlstm_cell_b, return_sequences=True, name='
49         Encoder_Backward_VLSTM', go_backwards=True)
50
51     with CustomObjectScope({'VLSTMCellEncoder': layers.VLSTMCellEncoder}):
52         biencoder = Bidirectional(encoder_f, backward_layer=encoder_b, name='
53             Bidirectional_Encoder_VLSTM')(encoder_embedd, initial_state=[
54                 encoder_init_state_f, encoder_init_state_f, encoder_init_state_b,
55                 encoder_init_state_b])
56
57     biencoder = Reshape(target_shape=(MAX_TRACK_NUM*ENCODER_UNITS*2,), name='
58         Reshape_Bidirectional_Encoder')(biencoder)
59
60     # DECODER_UNITS, ENCODER_UNITS, DECODER_OUTPUT, MAX_TRACK_NUM
61     attentionvlstm_cell = layers.AttentionVLSTMCell(DECODER_UNITS, ENCODER_UNITS
62             *2, 1, MAX_TRACK_NUM)
63
64     decoder, attention = RNN(attentionvlstm_cell, return_sequences=True, name='
```

---

```

        Decoder_Attention_VLSTM')(decoder_embedd, initial_state=[biencoder,
decoder_init_state])

50
51     model = Model(inputs=[pair_input, encoder_input, decoder_input], outputs=
decoder)

52
53     model.summary()

54
55     return model

```

---

コード A.1 では特に独自機構は使用しておらず、基本的には Tensorflow.Keras の標準的なライブラリで動作している。本研究では RNNCell を用いて独自のリカレントニューラルネットワーク構造を定義している。コード A.2, A.3 では以下の様に Cell を読み出し、使用している。

#### Listing A.4: 独自の LSTM

---

```

1     cell = layers.VLSTMCellSimple(UNITS, 1)
2
3     rnn = RNN(cell, return_sequences=True, name='Vertex_LSTM_Simple')(
track_embedd, initial_state=[init_state, init_state])

```

---

#### Listing A.5: 独自の Attention LSTM

---

```

1     vlstm_cell_f = layers.VLSTMCellEncoder(ENCODER_UNITS)
2     vlstm_cell_b = layers.VLSTMCellEncoder(ENCODER_UNITS)
3     encoder_f = RNN(vlstm_cell_f, return_sequences=True, name="Encoder_Forward_VLSTM",
go_backwards=False)
4     encoder_b = RNN(vlstm_cell_b, return_sequences=True, name="Encoder_Backward_VLSTM",
go_backwards=True)
5
6     with CustomObjectScope({'VLSTMCellEncoder': layers.VLSTMCellEncoder}):
7         biencoder = Bidirectional(encoder_f, backward_layer=encoder_b, name='
Bidirectional_Encoder_VLSTM')(encoder_embedd, initial_state=[encoder_init_state_f, encoder_init_state_f, encoder_init_state_b,
encoder_init_state_b])

```

---

```

1     attentionvlstm_cell = layers.AttentionVLSTMCell(DECODER_UNITS, ENCODER_UNITS
*2, 1, MAX_TRACK_NUM)
2
3     decoder, attention = RNN(attentionvlstm_cell, return_sequences=True, name='
Decoder_Attention_VLSTM')(decoder_embedd, initial_state=[biencoder,

```

---

```
decoder_init_state])
```

---

コード A.2 では”VLSTMCellSimple()”を、コード A.3 では”VLSTMCellEncoder()”, ”AttentionVLSTMCell()”をそれぞれ呼び出している。”VLSTMCellSimple()”は図 3.17 を示しており、以下の様に実装している。

---

**Listing A.6:** 独自のリカレントニューラルネットワーク構造の 1 ステップ

---

```

1 class VLSTMCellSimple(AbstractRNNCell):
2
3     def __init__(self,
4                  units,
5                  units_out,
6                  activation='tanh',
7                  recurrent_activation='hard_sigmoid',
8                  dense_activation='sigmoid',
9                  use_bias=True,
10                 kernel_initializer='glorot_uniform',
11                 recurrent_initializer='orthogonal',
12                 bias_initializer='zeros',
13                 unit_forget_bias=True,
14                 kernel_regularizer=None,
15                 recurrent_regularizer=None,
16                 bias_regularizer=None,
17                 kernel_constraint=None,
18                 recurrent_constraint=None,
19                 bias_constraint=None,
20                 implementation=1,
21                 **kwargs):
22
23     super(VLSTMCellSimple, self).__init__(**kwargs)
24     self.units = units
25     self.units_out = units_out
26     self.activation = activations.get(activation)
27     self.recurrent_activation = activations.get(recurrent_activation)
28     self.dense_activation = activations.get(dense_activation)
29     self.use_bias = use_bias
30
31     self.kernel_initializer = initializers.get(kernel_initializer)
32     self.recurrent_initializer = initializers.get(recurrent_initializer)
33     self.bias_initializer = initializers.get(bias_initializer)
```

```
34     self.unit_forget_bias = unit_forget_bias
35
36     self.kernel_regularizer = regularizers.get(kernel_regularizer)
37     self.recurrent_regularizer = regularizers.get(recurrent_regularizer)
38     self.bias_regularizer = regularizers.get(bias_regularizer)
39
40     self.kernel_constraint = constraints.get(kernel_constraint)
41     self.recurrent_constraint = constraints.get(recurrent_constraint)
42     self.bias_constraint = constraints.get(bias_constraint)
43
44     if implementation != 1:
45         logging.debug(RECURRENT_DROPOUT_WARNING_MSG)
46         self.implementation = 1
47     else:
48         self.implementation = implementation
49
50     @property
51     def state_size(self):
52         return [self.units, self.units]
53
54     def build(self, input_shape): # definition of the weights
55         input_dim = input_shape[-1]
56         self.kernel = self.add_weight( # W
57             shape=(input_dim, self.units * 4), # "* 4" means "o, f, i, z"
58             name='kernel',
59             initializer=self.kernel_initializer,
60             regularizer=self.kernel_regularizer,
61             constraint=self.kernel_constraint)
62         self.recurrent_kernel = self.add_weight( # R
63             shape=(self.units, self.units * 4),
64             name='recurrent_kernel',
65             initializer=self.recurrent_initializer,
66             regularizer=self.recurrent_regularizer,
67             constraint=self.recurrent_constraint)
68         self.dense_kernel = self.add_weight( # Last Dense kernel
69             shape=(self.units * 1, self.units_out),
70             name='dense_kernel',
71             initializer=self.recurrent_initializer,
72             regularizer=self.recurrent_regularizer,
73             constraint=self.recurrent_constraint)
```

```
74
75     if self.use_bias:
76         if self.unit_forget_bias:
77
78             def bias_initializer(_, *args, **kwargs):
79                 return K.concatenate([
80                     self.bias_initializer((self.units,), *args, **kwargs),
81                     initializers.Ones()((self.units,), *args, **kwargs),
82                     self.bias_initializer((self.units * 2,), *args, **kwargs),
83                 ])
84
85         else:
86             bias_initializer = self.bias_initializer
87             self.bias = self.add_weight( # b
88                 shape=(self.units * 4,),
89                 name='bias',
90                 initializer=bias_initializer,
91                 regularizer=self.bias_regularizer,
92                 constraint=self.bias_constraint)
93
94     else:
95         self.bias = None
96     self.built = True
97
98     def _compute_update_vertex(self, x, V_tm1):
99         """Computes carry and output using split kernels."""
100        # x = W * track
101        x_i, x_f, x_c, x_o = x
102        V_tm1_i, V_tm1_f, V_tm1_c, V_tm1_o, V_tm1_o2, V_tm1_u, V_tm1_v = V_tm1
103        # i = x_i + V_tm1_i * R_i
104        # = W_i * track + V_tm1_1 * R_i
105
106        i = self.recurrent_activation(
107            x_i + K.dot(V_tm1_i, self.recurrent_kernel[:, :self.units]))
108        f = self.recurrent_activation(
109            x_f + K.dot(V_tm1_f, self.recurrent_kernel[:, self.units:self.units *
110                2]))
111        c = self.activation(
112            x_c + K.dot(V_tm1_c, self.recurrent_kernel[:, self.units * 2:self.units
113                * 3]))
114
115        # U = update vertex
```

```
112     U = f * V_tm1_u + i * c
113
114     o = self.recurrent_activation(
115         x_o + K.dot(V_tm1_o, self.recurrent_kernel[:, self.units * 3:])))
116
117     h_temp = o * self.activation(V_tm1_o2)
118     # h size [self.units]
119     h = self.dense_activation(K.dot(h_temp, self.dense_kernel))
120     # h size [1] activated sigmoid
121
122     V = h * U + (1-h) * V_tm1_v
123     return h, V
124
125 def call(self, inputs, states, training=None):
126     V_tm1 = states[0] # previous Vertex state
127
128     if self.implementation == 1:
129         # input = track
130         inputs_i = inputs
131         inputs_f = inputs
132         inputs_c = inputs
133         inputs_o = inputs
134         # k = W
135         k_i, k_f, k_c, k_o = array_ops.split(
136             self.kernel, num_or_size_splits=4, axis=1)
137         x_i = K.dot(inputs_i, k_i)
138         x_f = K.dot(inputs_f, k_f)
139         x_c = K.dot(inputs_c, k_c)
140         x_o = K.dot(inputs_o, k_o)
141         if self.use_bias:
142             b_i, b_f, b_c, b_o = array_ops.split(
143                 self.bias, num_or_size_splits=4, axis=0)
144             x_i = K.bias_add(x_i, b_i)
145             x_f = K.bias_add(x_f, b_f)
146             x_c = K.bias_add(x_c, b_c)
147             x_o = K.bias_add(x_o, b_o)
148
149     V_tm1_i = V_tm1
150     V_tm1_f = V_tm1
151     V_tm1_c = V_tm1
```

```
152     V_tm1_o = V_tm1
153     V_tm1_o2 = V_tm1
154     V_tm1_u = V_tm1
155     V_tm1_v = V_tm1
156     x = (x_i, x_f, x_c, x_o)
157
158     V_tm1 = (V_tm1_i, V_tm1_f, V_tm1_c, V_tm1_o, V_tm1_o2, V_tm1_u, V_tm1_v)
159     h, V = self._compute_update_vertex(x, V_tm1)
160
161     return h, [V, V]
162
163 def get_config(self):
164     config = {
165         'units':
166             self.units,
167         'units_out':
168             self.units_out,
169         'activation':
170             activations.serialize(self.activation),
171         'recurrent_activation':
172             activations.serialize(self.recurrent_activation),
173         'use_bias':
174             self.use_bias,
175         'kernel_initializer':
176             initializers.serialize(self.kernel_initializer),
177         'recurrent_initializer':
178             initializers.serialize(self.recurrent_initializer),
179         'bias_initializer':
180             initializers.serialize(self.bias_initializer),
181         'unit_forget_bias':
182             self.unit_forget_bias,
183         'kernel_regularizer':
184             regularizers.serialize(self.kernel_regularizer),
185         'recurrent_regularizer':
186             regularizers.serialize(self.recurrent_regularizer),
187         'bias_regularizer':
188             regularizers.serialize(self.bias_regularizer),
189         'kernel_constraint':
190             constraints.serialize(self.kernel_constraint),
191         'recurrent_constraint':
```

```

192     constraints.serialize(self.recurrent_constraint),
193     'bias_constraint':
194         constraints.serialize(self.bias_constraint),
195     'implementation':
196         self.implementation
197 }
198 base_config = super(VLSTMCellSimple, self).get_config()
199 return dict(list(base_config.items()) + list(config.items()))

```

ここでは、学習可能な重み行列  $W_f, W_i, W_c, W_o, R_f, R_i, R_c, R_o, \mathbf{d}_h$  は以下で定義されている。

**Listing A.7:** 独自のリカレントニューラルネットワーク構造の 1 ステップ

```

1 def build(self, input_shape): # definition of the weights
2     input_dim = input_shape[-1]
3     self.kernel = self.add_weight( # W
4         shape=(input_dim, self.units * 4), # "* 4" means "o, f, i, z"
5         name='kernel',
6         initializer=self.kernel_initializer,
7         regularizer=self.kernel_regularizer,
8         constraint=self.kernel_constraint)
9     self.recurrent_kernel = self.add_weight( # R
10        shape=(self.units, self.units * 4),
11        name='recurrent_kernel',
12        initializer=self.recurrent_initializer,
13        regularizer=self.recurrent_regularizer,
14        constraint=self.recurrent_constraint)
15     self.dense_kernel = self.add_weight( # Last Dense kernel
16        shape=(self.units * 1, self.units_out),
17        name='dense_kernel',
18        initializer=self.recurrent_initializer,
19        regularizer=self.recurrent_regularizer,
20        constraint=self.recurrent_constraint)
21
22     if self.use_bias:
23         if self.unit_forget_bias:
24
25             def bias_initializer(_, *args, **kwargs):
26                 return K.concatenate([
27                     self.bias_initializer((self.units,), *args, **kwargs),

```

```

28         initializers.Ones()((self.units,), *args, **kwargs),
29         self.bias_initializer((self.units * 2,), *args, **kwargs),
30     ])
31     else:
32         bias_initializer = self.bias_initializer
33         self.bias = self.add_weight( # b
34             shape=(self.units * 4,),
35             name='bias',
36             initializer=bias_initializer,
37             regularizer=self.bias_regularizer,
38             constraint=self.bias_constraint)
39     else:
40         self.bias = None
41     self.built = True

```

---

また、1ステップの演算は以下の様に行われる。

**Listing A.8:** 独自のリカレントニューラルネットワーク構造の1ステップ

```

1 def _compute_update_vertex(self, x, V_tm1):
2     """Computes carry and output using split kernels."""
3     # x = W * track
4     x_i, x_f, x_c, x_o = x
5     V_tm1_i, V_tm1_f, V_tm1_c, V_tm1_o, V_tm1_o2, V_tm1_u, V_tm1_v = V_tm1
6     # i = x_i + V_tm1_i * R_i
7     # = W_i * track + V_tm1_1 * R_i
8
9     i = self.recurrent_activation(
10        x_i + K.dot(V_tm1_i, self.recurrent_kernel[:, :self.units]))
11     f = self.recurrent_activation(
12        x_f + K.dot(V_tm1_f, self.recurrent_kernel[:, self.units:self.units *
13            2]))
13     c = self.activation(
14        x_c + K.dot(V_tm1_c, self.recurrent_kernel[:, self.units * 2:self.units
15            * 3]))
15
16     # U = update vertex
17     U = f * V_tm1_u + i * c
18
19     o = self.recurrent_activation(
20        x_o + K.dot(V_tm1_o, self.recurrent_kernel[:, self.units * 3:])))
21

```

```
22     h_temp = o * self.activation(V_tm1_o2)
23     # h size [self.units]
24     h = self.dense_activation(K.dot(h_temp, self.dense_kernel))
25     # h size [1] activated sigmoid
26
27     V = h * U + (1-h) * V_tm1_v
28     return h, V
29
30 def call(self, inputs, states, training=None):
31     V_tm1 = states[0] # previous Vertex state
32
33     if self.implementation == 1:
34         # input = track
35         inputs_i = inputs
36         inputs_f = inputs
37         inputs_c = inputs
38         inputs_o = inputs
39         # k = W
40         k_i, k_f, k_c, k_o = array_ops.split(
41             self.kernel, num_or_size_splits=4, axis=1)
42         x_i = K.dot(inputs_i, k_i)
43         x_f = K.dot(inputs_f, k_f)
44         x_c = K.dot(inputs_c, k_c)
45         x_o = K.dot(inputs_o, k_o)
46         if self.use_bias:
47             b_i, b_f, b_c, b_o = array_ops.split(
48                 self.bias, num_or_size_splits=4, axis=0)
49             x_i = K.bias_add(x_i, b_i)
50             x_f = K.bias_add(x_f, b_f)
51             x_c = K.bias_add(x_c, b_c)
52             x_o = K.bias_add(x_o, b_o)
53
54     V_tm1_i = V_tm1
55     V_tm1_f = V_tm1
56     V_tm1_c = V_tm1
57     V_tm1_o = V_tm1
58     V_tm1_o2 = V_tm1
59     V_tm1_u = V_tm1
60     V_tm1_v = V_tm1
61     x = (x_i, x_f, x_c, x_o)
```

```

62
63     V_tm1 = (V_tm1_i, V_tm1_f, V_tm1_c, V_tm1_o, V_tm1_o2, V_tm1_u, V_tm1_v)
64     h, V = self._compute_update_vertex(x, V_tm1)
65
66     return h, [V, V]

```

---

以上が図 3.17 に示したネットワークである。次に、これらをエンコーダー・デコーダーモデルへと拡張した際の変更点について実装をまとめ。まずエンコーダー部については、デコーダー部へより多くの情報を受け渡す為、重み  $d_h$  を取り除いている。

**Listing A.9:** エンコーダー部の変更点

```

1 class VLSTMCellEncoder(AbstractRNNCell):
2     def build(self, input_shape): # definition of the weights
3         input_dim = input_shape[-1]
4         self.batch_size = input_shape[0]
5         self.kernel = self.add_weight( # W
6             shape=(input_dim, self.units * 4), # "* 4" means "o, f, i, z"
7             name='kernel',
8             initializer=self.kernel_initializer,
9             regularizer=self.kernel_regularizer,
10            constraint=self.kernel_constraint)
11        self.recurrent_kernel = self.add_weight( # R
12            shape=(self.units, self.units * 4),
13            name='recurrent_kernel',
14            initializer=self.recurrent_initializer,
15            regularizer=self.recurrent_regularizer,
16            constraint=self.recurrent_constraint)
17
18        if self.use_bias:
19            if self.unit_forget_bias:
20
21                def bias_initializer(_, *args, **kwargs):
22                    return K.concatenate([
23                        self.bias_initializer((self.units,), *args, **kwargs),
24                        initializers.Ones()((self.units,), *args, **kwargs),
25                        self.bias_initializer((self.units * 2,), *args, **kwargs),
26                    ])
27            else:
28                bias_initializer = self.bias_initializer
29                self.bias = self.add_weight( # b

```

```
30         shape=(self.units * 4,),
31         name='bias',
32         initializer=bias_initializer,
33         regularizer=self.bias_regularizer,
34         constraint=self.bias_constraint)
35     else:
36         self.bias = None
37     self.built = True
38
39     def _compute_update_vertex(self, x, V_tm1):
40         """Computes carry and output using split kernels."""
41         x_i, x_f, x_c, x_o = x
42         V_tm1_i, V_tm1_f, V_tm1_c, V_tm1_o, V_tm1_o2, V_tm1_u, V_tm1_v = V_tm1
43
44         i = self.recurrent_activation(
45             x_i + K.dot(V_tm1_i, self.recurrent_kernel[:, :self.units]))
46         f = self.recurrent_activation(
47             x_f + K.dot(V_tm1_f, self.recurrent_kernel[:, self.units:self.units *
48                 2]))
48         c = self.activation(
49             x_c + K.dot(V_tm1_c, self.recurrent_kernel[:, self.units * 2:self.units
50                 * 3]))
50
51         U = f * V_tm1_u + i * c
52
53         o = self.recurrent_activation(
54             x_o + K.dot(V_tm1_o, self.recurrent_kernel[:, self.units * 3:]))
55
56         h = o * self.activation(V_tm1_o2)
57
58         V = h * U + (1-h) * V_tm1_v
59         #the size of h is [self.units]
60         return h, V
61
62     def call(self, inputs, states, training=None):
63         V_tm1 = states[0] # previous Vertex state
64
65
66         if self.implementation == 1:
67             inputs_i = inputs
```

```

68     inputs_f = inputs
69     inputs_c = inputs
70     inputs_o = inputs
71     k_i, k_f, k_c, k_o = array_ops.split(
72         self.kernel, num_or_size_splits=4, axis=1)
73     x_i = K.dot(inputs_i, k_i)
74     x_f = K.dot(inputs_f, k_f)
75     x_c = K.dot(inputs_c, k_c)
76     x_o = K.dot(inputs_o, k_o)
77     if self.use_bias:
78         b_i, b_f, b_c, b_o = array_ops.split(
79             self.bias, num_or_size_splits=4, axis=0)
80         x_i = K.bias_add(x_i, b_i)
81         x_f = K.bias_add(x_f, b_f)
82         x_c = K.bias_add(x_c, b_c)
83         x_o = K.bias_add(x_o, b_o)
84
85     V_tm1_i = V_tm1
86     V_tm1_f = V_tm1
87     V_tm1_c = V_tm1
88     V_tm1_o = V_tm1
89     V_tm1_o2 = V_tm1
90     V_tm1_u = V_tm1
91     V_tm1_v = V_tm1
92     x = (x_i, x_f, x_c, x_o)
93
94     V_tm1 = (V_tm1_i, V_tm1_f, V_tm1_c, V_tm1_o, V_tm1_o2, V_tm1_u, V_tm1_v)
95     h, V = self._compute_update_vertex(x, V_tm1)
96
97     return h, [V, V]

```

デコーダー部では Attention へ拡張する為、重み行列  $u_{\text{energy}}$ ,  $U_{\text{key}}$ ,  $U_{\text{query}}$ ,  $C_f$ ,  $C_c$ ,  $C_i$ ,  $C_o$  を追加している。

**Listing A.10:** デコーダー部の変更点

```

1  def build(self, input_shape): # definition of the weights
2      self.feature_dim = input_shape[-1]
3      self.batch_size = input_shape[0]
4
5      self.attention_kernel_V = self.add_weight (# u_energy

```

```
6      shape=(self.units,) ,
7      name='attention_kernel_V' ,
8      initializer=self.kernel_initializer,
9      regularizer=self.kernel_regularizer,
10     constraint=self.kernel_constraint)

11
12     self.attention_kernel_W = self.add_weight( # U_Query
13         shape=(self.feature_dim, self.units),
14         name='attention_kernel_W',
15         initializer=self.kernel_initializer,
16         regularizer=self.kernel_regularizer,
17         constraint=self.kernel_constraint)

18
19     self.attention_kernel_U = self.add_weight( # U_Key
20         shape=(self.att_input_dim, self.units),
21         name='attention_kernel_U',
22         initializer=self.kernel_initializer,
23         regularizer=self.kernel_regularizer,
24         constraint=self.kernel_constraint)

25
26     self.attention_kernel_b = self.add_weight( # U_Key bias
27         shape=(self.units,) ,
28         name='attention_kernel_b' ,
29         initializer=self.bias_initializer,
30         regularizer=self.bias_regularizer,
31         constraint=self.bias_constraint)

32
33     self.kernel = self.add_weight( # W
34         shape=(self.feature_dim, self.units * 4), # "* 4" means "o, f, i, z"
35         name='att_kernel' ,
36         initializer=self.kernel_initializer,
37         regularizer=self.kernel_regularizer,
38         constraint=self.kernel_constraint)

39
40     self.recurrent_kernel = self.add_weight( # R
41         shape=(self.units, self.units * 4),
42         name='att_recurrent_kernel' ,
43         initializer=self.recurrent_initializer,
44         regularizer=self.recurrent_regularizer,
45         constraint=self.recurrent_constraint)
```

```
46
47     self.dense_kernel = self.add_weight( # Last Dense kernel
48         shape=(self.units * 1, self.output_dim),
49         name='att_dense_kernel',
50         initializer=self.recurrent_initializer,
51         regularizer=self.recurrent_regularizer,
52         constraint=self.recurrent_constraint)
53
54     self.context_kernel = self.add_weight( # C
55         shape=(self.att_input_dim, self.units * 4),
56         name='att_recurrent_kernel',
57         initializer=self.recurrent_initializer,
58         regularizer=self.recurrent_regularizer,
59         constraint=self.recurrent_constraint)
60
61     if self.use_bias:
62         if self.unit_forget_bias:
63
64             def bias_initializer(_, *args, **kwargs):
65                 return K.concatenate([
66                     self.bias_initializer((self.units,), *args, **kwargs),
67                     initializers.Ones()((self.units,), *args, **kwargs),
68                     self.bias_initializer((self.units * 2,), *args, **kwargs),
69                 ])
70
71         else:
72             bias_initializer = self.bias_initializer
73             self.bias = self.add_weight( # b
74                 shape=(self.units * 4,),
75                 name='att_bias',
76                 initializer=bias_initializer,
77                 regularizer=self.bias_regularizer,
78                 constraint=self.bias_constraint)
79
80         self.bias = None
81         self.built = True
```

演算についても以下の様に変更している。

```

4                         timesteps=None, training=None):
5     if input_dim is None:
6         input_dim = K.shape(x)[2]
7     if timesteps is None:
8         timesteps = K.shape(x)[1]
9     if output_dim is None:
10        output_dim = K.shape(w)[1]
11
12    if dropout is not None and 0. < dropout < 1.:
13        # apply the same dropout pattern at every timestep
14        ones = K.ones_like(K.reshape(x[:, 0, :], (-1, input_dim)))
15        dropout_matrix = K.dropout(ones, dropout)
16        expanded_dropout_matrix = K.repeat(dropout_matrix, timesteps)
17        x = K.in_train_phase(x * expanded_dropout_matrix, x, training=training)
18
19    # collapse time dimension and batch dimension together
20    x = K.reshape(x, (-1, input_dim))
21    x = K.dot(x, w)
22    if b is not None:
23        x = K.bias_add(x, b)
24    # reshape to 3D tensor
25    if K.backend() == 'tensorflow':
26        x = K.reshape(x, K.stack([-1, timesteps, output_dim]))
27        x.set_shape([None, None, output_dim])
28    else:
29        x = K.reshape(x, (-1, timesteps, output_dim))
30    return x
31
32 def _compute_update_vertex(self, x, V_tm1, c):
33     """Computes carry and output using split kernels."""
34     x_i, x_f, x_c, x_o = x
35     V_tm1_i, V_tm1_f, V_tm1_c, V_tm1_o, V_tm1_o2, V_tm1_u, V_tm1_v = V_tm1
36     c_i, c_f, c_c, c_o = c
37
38     i = self.recurrent_activation(
39         x_i
40         + K.dot(V_tm1_i, self.recurrent_kernel[:, :self.units])
41         + K.dot(c_i, self.context_kernel[:, :self.units])) # Attention
42         information

```

```

43     f = self.recurrent_activation(
44         x_f
45         + K.dot(V_tm1_f, self.recurrent_kernel[:, self.units:self.units * 2])
46         + K.dot(c_f, self.context_kernel[:, self.units:self.units * 2]))
47
48     c = self.activation(
49         x_c
50         + K.dot(V_tm1_c, self.recurrent_kernel[:, self.units * 2:self.units *
51             3])
52         + K.dot(c_c, self.context_kernel[:, self.units * 2:self.units * 3]))
53
54     U = f * V_tm1_u + i * c
55
56     o = self.recurrent_activation(
57         x_o
58         + K.dot(V_tm1_o, self.recurrent_kernel[:, self.units * 3:])
59         + K.dot(c_o, self.context_kernel[:, self.units * 3:]))
60
61     h_temp = o * self.activation(V_tm1_o2)
62     # h size [self.units]
63     h = self.dense_activation(K.dot(h_temp, self.dense_kernel))
64     # h size [1] activated sigmoid
65
66     V = h * U + (1-h) * V_tm1_v
67     return h, V
68
69     def call(self, inputs, states, training=None):
70         # store the whole sequence so we can "attend" to it at each timestep
71
72         att = states[0] # Attention input (track num, input dim) / key
73         self.x_seq = K.reshape(att, (-1, self.timestep, self.att_input_dim)) #
74             # Attention input (track num, input dim)
75         V_tm1 = states[1] # previous Vertex state (units)
76
77         # Additive Attention Bahdanau et al., 2015
78         self._uxpb = self._time_distributed_dense(self.x_seq,
79                                         self.attention_kernel_U,
80                                         b=self.attention_kernel_b,
81                                         input_dim=self.att_input_dim,
82                                         timesteps=self.timestep,
83                                         training=training)
84
85         # Compute context vector
86         context = K.dot(self.x_seq, self.attention_kernel_U)
87         context = context + self.attention_kernel_b
88         context = self.attention_activation(context)
89
90         # Compute output
91         o = self.dense_activation(K.dot(context, self.dense_kernel))
92
93         return o

```

```
81                                         output_dim=self.units)
82
83     # repeat the input track to the length of the sequence (track num, feature dim
84     ))
85
86     _tt = K.repeat(inputs, self.timestep) # inputs / query
87     _Wxtt = K.dot(_tt, self.attention_kernel_W)
88     et = K.dot(activations.tanh(_Wxtt + self._uxpb), K.expand_dims(self.
89                     attention_kernel_V)) # Energy
90
91     """
92
93     #Dot-Product Attention Luong et al., 2015 / Scaled Dot-Product Attention
94     Vaswani 2017
95
96     self.x_seq /= np.sqrt(self.att_input_dim)
97     et = K.batch_dot(K.expand_dims(inputs), self.x_seq, axes=[1, 2])
98     et = K.reshape(et, (-1, self.timestep, 1))
99     """
100
101
102     at = K.exp(et)
103     at_sum = K.sum(at, axis=1)
104     at_sum_repeated = K.repeat(at_sum, self.timestep)
105     at /= at_sum_repeated # attention weights ({batchsize}, track num, 1)
106     context = K.squeeze(K.batch_dot(at, self.x_seq, axes=1), axis=1) # context
107
108
109     if self.implementation == 1:
110         inputs_i = inputs
111         inputs_f = inputs
112         inputs_c = inputs
113         inputs_o = inputs
114         k_i, k_f, k_c, k_o = array_ops.split(
115             self.kernel, num_or_size_splits=4, axis=1)
116         x_i = K.dot(inputs_i, k_i)
117         x_f = K.dot(inputs_f, k_f)
118         x_c = K.dot(inputs_c, k_c)
119         x_o = K.dot(inputs_o, k_o)
120
121         if self.use_bias:
122             b_i, b_f, b_c, b_o = array_ops.split(
123                 self.bias, num_or_size_splits=4, axis=0)
124             x_i = K.bias_add(x_i, b_i)
125             x_f = K.bias_add(x_f, b_f)
```

```

118     x_c = K.bias_add(x_c, b_c)
119     x_o = K.bias_add(x_o, b_o)
120
121     V_tm1_i = V_tm1
122     V_tm1_f = V_tm1
123     V_tm1_c = V_tm1
124     V_tm1_o = V_tm1
125     V_tm1_o2 = V_tm1
126     V_tm1_u = V_tm1
127     V_tm1_v = V_tm1
128
129     c_i = context
130     c_f = context
131     c_c = context
132     c_o = context
133
134     x = (x_i, x_f, x_c, x_o)
135     V_tm1 = (V_tm1_i, V_tm1_f, V_tm1_c, V_tm1_o, V_tm1_o2, V_tm1_u, V_tm1_v)
136     c = (c_i, c_f, c_c, c_o)
137
138     h, V = self._compute_update_vertex(x, V_tm1, c)
139
140     return [h, at], [att, V]

```

---

## A.2 崩壊点の再構成

ここでは崩壊点検出アルゴリズムの内、Python を用いた実装について解説する。

●

---

```

1 def ModelsLoad(pair_model_name, lstm_model_name, slstm_model_name):
2
3     pair_model = modeltools.LoadPairModel(pair_model_name)
4     lstm_model = modeltools.LoadAttentionVLSTMModel(lstm_model_name)
5     slstm_model = modeltools.LoadAttentionVLSTMModel(slstm_model_name)
6     pair_model.compile(loss={'Vertex_Output': 'categorical_crossentropy',
7                             'Position_Output': 'mean_squared_logarithmic_error'},
8                         optimizer='SGD',
9                         metrics=['accuracy', 'mae'])
10    lstm_model.compile(loss='binary_crossentropy',
11                         optimizer='SGD',
12                         metrics=['accuracy'])

```

```
10         optimizer="Adam",
11         metrics=['accuracy'])
12     slstm_model.compile(loss='binary_crossentropy',
13                           optimizer="Adam",
14                           metrics=['accuracy'])
15
16     return pair_model, lstm_model, slstm_model
17
18
19 def PairInference(pair_model, variables):
20
21     predict_vertex, predict_position = pair_model.predict([variables], verbose
22     =1)
23
24     return predict_vertex, predict_position
25
26 def GetEncoderDecoderTracksandTrue(debug, event_data, NTrack, MaxTrack):
27
28     tracks = []
29     true_label = []
30     chain_lists = []
31     particle_lists = []
32     chain_label = [0 for i in range(int(NTrack))]
33     vertex_mat_tbbcc = [[0 for j in range(int(NTrack))] for i in range(int(
34         NTrack))]
35     vertex_mat_tbc = [[0 for j in range(int(NTrack))] for i in range(int(NTrack
36         ))]]
37
38     for event_datum in event_data:
39         if event_datum[57] == 2 or event_datum[57] == 3 or event_datum[57] ==
40             4:
41             vertex_mat_tbbcc[int(event_datum[1])][int(event_datum[2])] = 1
42             vertex_mat_tbbcc[int(event_datum[2])][int(event_datum[1])] = 1
43             if event_datum[57] == 3 or event_datum[57] == 4 or event_datum[57] ==
44                 5:
45                 vertex_mat_tbc[int(event_datum[1])][int(event_datum[2])] = 1
46                 vertex_mat_tbc[int(event_datum[2])][int(event_datum[1])] = 1
47
48         if int(event_datum[1]) == int(NTrack)-1:
```

```
45
46     if(event_datum[71]==1): true_label.append("c")
47     elif(event_datum[72]==1): true_label.append("b")
48     elif(event_datum[73]==1): true_label.append("o")
49     elif(event_datum[74]==1): true_label.append("p")
50
51     tracks.append(np.concatenate([[1], event_datum[25:47]]))
52     if int(event_datum[2]) == int(NTtrack)-2:
53
54         if(event_datum[63]==1): true_label.append("c")
55         elif(event_datum[64]==1): true_label.append("b")
56         elif(event_datum[65]==1): true_label.append("o")
57         elif(event_datum[66]==1): true_label.append("p")
58
59     tracks.append(np.concatenate([[1], event_datum[3:25]]))
60
61 decoder_tracks = np.array(deepcopy(tracks))
62 encoder_tracks = np.pad(np.array(deepcopy(tracks)), [(0, int(MaxTrack-NTtrack
    )), (0, 0)])
63
64 vertex_mat_tbbcc = np.array(vertex_mat_tbbcc)
65 vertex_mat_tbc = np.array(vertex_mat_tbc)
66
67 for t in range(int(NTtrack)):
68     vertex_mat_tbbcc[t][t] = 1
69     vertex_mat_tbc[t][t] = 1
70     tmp_particle_lists= [part for particle in particle_lists for part in
71                         particle]
72     tmp_chain_lists= [ch for chain in chain_lists for ch in chain]
73     particle_list = [i for i, x in enumerate(vertex_mat_tbbcc[t, :]) if x
74                      == 1]
75     chain_list = [i for i, x in enumerate(vertex_mat_tbc[t, :]) if x == 1]
76     if len(particle_list) > 1 and (t not in tmp_particle_lists):
77         particle_lists.append(particle_list)
78     if len(chain_list) > 1 and (t not in tmp_chain_lists): chain_lists.
79         append(chain_list)
80
81 for i, particle_list in enumerate(particle_lists):
82     for particle in particle_list:
83         chain_label[int(particle)] = -(i+1)
```

```
80
81     for i, chain_list in enumerate(chain_lists):
82         c = i + 1
83         for chain in chain_list:
84             if chain_label[int(chain)] > 0: c = chain_label[int(chain)]
85         for chain in chain_list:
86             chain_label[int(chain)] = c
87
88     if debug==True:
89         print("Encoder_Trach_Shape" + str(encoder_tracks.shape))
90         print("Decoder_Trach_Shape" + str(decoder_tracks.shape))
91         print("True_Label" + str(true_label))
92         print("Chain_Label" + str(chain_label))
93         print(list(particle_lists))
94         print(list(chain_lists))
95
96     return encoder_tracks, decoder_tracks, true_label, chain_label
97
98
99 def SecondarySeedSelectionOne(debug, event_data, ThresholdPairSecondaryScore,
100                               ThresholdPairPosScore):
101
102     predict_vertex_labels = np.argmax(event_data[:, -8:-1], axis=1)
103     secondary_event_data = []
104     tmp_secondary_scores = []
105     for event_datum, predict_vertex_label in zip(event_data,
106                                                   predict_vertex_labels):
107         tmp_secondary_score = event_datum[-6] + event_datum[-5] + event_datum
108                     [-4] + event_datum[-3]
109         if predict_vertex_label==0 or predict_vertex_label==1 or
110             predict_vertex_label==6: continue
111         if tmp_secondary_score < ThresholdPairSecondaryScore: continue
112         if event_datum[-1] > ThresholdPairPosScore: continue
113
114         secondary_event_data.append(event_datum)
115         tmp_secondary_scores.append(tmp_secondary_score)
116
117     tmp_secondary_scores = np.array(tmp_secondary_scores)
118     secondary_event_data = np.array(secondary_event_data)
119     index = np.argsort(-tmp_secondary_scores)
```

```
116
117     if debug==True:
118         for i, secondary_event_datum in enumerate(secondary_event_data[index]):
119             print("Secondary\u201cSeeds\u201d" + str(i) + "\u201cTrack\u201d1:" + str(
120                 secondary_event_datum[1]) + "\u201cTrack\u201d2:" + str(
121                 secondary_event_datum[2]) + "\u201cSV\u201dScore:\u201d" + str(
122                 secondary_event_datum[-6] + secondary_event_datum[-5] +
123                 secondary_event_datum[-4] + secondary_event_datum[-3]))
124
125     return secondary_event_data[index]
126
127
128
129
130 def PrimaryVertexFinder(debug, MaxPrimaryVertexLoop, ThresholdPrimaryScore,
131                         event_data,
132                         encoder_tracks, decoder_tracks, lstm_model):
133
134     primary_track_list = []
135     bigger_primary_scores = []
136     primary_pairs = []
137
138     for event_datum in event_data[np.argsort(-event_data[:, -7])][:MaxPrimaryVertexLoop]:
139         primary_pairs.append(event_datum[3:47])
140
141     primary_encoder_tracks = np.tile(encoder_tracks, (MaxPrimaryVertexLoop, 1,
142                                                       1)) # tracks.shape = (MaxPrimaryVertexLoop, MaxTrack, 23)
143     primary_decoder_tracks = np.tile(decoder_tracks, (MaxPrimaryVertexLoop, 1,
144                                                       1)) # tracks.shape = (MaxPrimaryVertexLoop, NTrack, 23)
145
146     if debug==True: print("Primary\u201cEncoder\u201dTrach\u201dShape\u201d" + str(
147         primary_encoder_tracks.shape))
148     if debug==True: print("Primary\u201cDecoder\u201dTrach\u201dShape\u201d" + str(
149         primary_decoder_tracks.shape))
150
151     primary_scores = lstm_model.predict([primary_pairs, primary_encoder_tracks,
152                                         primary_decoder_tracks])
153
154
155     for i in range(len(primary_scores[0])):
156         tmpbigger_primary_scores = 0
157         for j in range(len(primary_scores)):
158             score = primary_scores[j][i]
159             if tmpbigger_primary_scores < score: tmpbigger_primary_scores = score
160
161         bigger_primary_scores.append(tmpbigger_primary_scores)
```

```
145     if tmpbigger_primary_scores < ThresholdPrimaryScore: continue
146     if debug==True: print("Track" + str(i) + "PrimaryScore:" + str(
147         tmpbigger_primary_scores))
148     primary_track_list.append(i)
149
150
151
152 def SecondaryVertexFinder(debug, ThresholdSecondaryScore, bigger_primary_scores,
153                           primary_track_list, secondary_event_data,
154                           encoder_tracks, decoder_tracks, slstm_model):
155
156     track_list = np.arange(decoder_tracks.shape[0])
157     secondary_track_lists = []
158     for secondary_event_datum in secondary_event_data:
159         track1, track2 = secondary_event_datum[1], secondary_event_datum[2]
160         if (track1 in primary_track_list) or (track2 in primary_track_list):
161             continue
162         if (track1 not in list(track_list)) or (track2 not in list(track_list)):
163             continue
164
165         #if debug==True: print("Track List " + str(track_list))
166
167         remain_decoder_tracks = decoder_tracks[track_list]
168         secondary_pair = np.tile(secondary_event_datum[3:47], (1, 1)) # pair.
169         shape = (1, 44)
170         secondary_encoder_tracks = np.tile(encoder_tracks, (1, 1, 1)) # tracks.
171         shape = (1, MaxTrack, 23)
172         secondary_decoder_tracks = np.tile(remain_decoder_tracks, (1, 1, 1)) #
173         tracks.shape = (1, RemainNTrack, 23)
174         secondary_scores = slstm_model.predict([secondary_pair,
175                                               secondary_encoder_tracks, secondary_decoder_tracks])
176         secondary_scores = np.array(secondary_scores).reshape((-1, 1))
177
178         tmptrack_list = np.copy(track_list)
179         primary_track_list = np.array(primary_track_list, dtype=int)
180         tmpsecondary_track_list = []
181         for i, t in enumerate(track_list):
182             if secondary_scores[i] > ThresholdSecondaryScore:
183                 if t not in primary_track_list:
```

```
177         tmpsecondary_track_list.append(t)
178         tmptrack_list = tmptrack_list[~(tmptrack_list == t)]
179     elif (t in primary_track_list) and (secondary_scores[i] >
180         bigger_primary_scores[t]):
181         tmpsecondary_track_list.append(t)
182     if debug==True:
183         print("Scramble\u2022Track\u2022Number\u2022" + str(t) + "\u2022SV\u2022Score\u2022" +
184             str(secondary_score[i])
185             + "\u2022PV\u2022Score\u2022" + str(bigger_primary_scores[t]))
186     primary_track_list = primary_track_list[~(primary_track_list
187         == t)]
188     tmptrack_list = tmptrack_list[~(tmptrack_list == t)]
189     if len(tmpsecondary_track_list)!=0: secondary_track_lists.append(
190         tmpsecondary_track_list)
191     track_list = np.copy(tmptrack_list)
192
193
194     ccbbvtx = 0
195     bcvtx = 0
196     true_secondary_bb_track_lists = []
197     true_secondary_cc_track_lists = []
198     true_secondary_same_chain_track_lists = []
199
200     print(pycolor.YELLOW + "True\u2022Primary\u2022Vertex" + pycolor.END)
201     true_primary_track_list = [i for i, x in enumerate(true_label) if x == "p"]
202     print(pycolor.YELLOW + str(true_primary_track_list) + pycolor.END)
203
204     for vtx in list(set(chain_label)):
205         tcc = [i for i, (x, c) in enumerate(zip(true_label, chain_label)) if x
206             == "c" and c == vtx]
207         tbb = [i for i, (x, c) in enumerate(zip(true_label, chain_label)) if x
208             == "b" and c == vtx]
209
210         if vtx < 0:
211             ccbbvtx = ccbbvtx + 1
212         if len(tcc) != 0:
213             print(pycolor.CYAN + "True\u2022Secondary\u2022Vertex\u2022Alone\u2022" + str(ccbbvtx)
```

```
        ) + pycolor.END)
211     print(pycolor.CYAN + "cc\u207b:\u207b" + str(tcc) + pycolor.END)
212     true_secondary_cc_track_lists.append(tcc)
213
214     if len(tbb) != 0:
215         print(pycolor.CYAN + "TrueSecondaryVertexAlone" + str(ccbbvtx)
216               ) + pycolor.END)
216     print(pycolor.CYAN + "bb\u207b:\u207b" + str(tbb) + pycolor.END)
217     true_secondary_bb_track_lists.append(tbb)
218
219     elif vtx > 0:
220         bcvtx = ccbbvtx + 1
221         print(pycolor.CYAN + "TrueSecondaryVertexChain" + str(bcvtx) +
222               pycolor.END)
222         print(pycolor.CYAN + "cc\u207b:\u207b" + str(tcc) + pycolor.END)
223         print(pycolor.CYAN + "bb\u207b:\u207b" + str(tbb) + pycolor.END)
224         true_secondary_cc_track_lists.append(tcc)
225         true_secondary_bb_track_lists.append(tbb)
226         true_secondary_same_chain_track_lists.append([i for i, c in enumerate
227               (chain_label) if c == vtx])
227
228     print(pycolor.YELLOW + "TrueOtherTracks" + pycolor.END)
229     true_other_track_list = [i for i, x in enumerate(true_label) if x == "o"]
230     print(pycolor.YELLOW + str(true_other_track_list) + pycolor.END)
231
232     return true_primary_track_list, true_secondary_bb_track_lists,
233           true_secondary_cc_track_lists, true_secondary_same_chain_track_lists,
234           true_other_track_list
235
236
237 def PrintPredTrackLists(primary_track_list, secondary_track_lists):
238
239     print("PredictPrimaryVertex")
240     print(list(primary_track_list))
241
242     for i, secondary_track_list in enumerate(secondary_track_lists):
243         print("PredictSecondaryVertex" + str(i))
244         print(list(secondary_track_list))
```

```
245 def yinx(y, x):
246     for _y in y:
247         if _y not in x: return False
248     return True
249
250
251 def listremove(y, x):
252     for _y in y:
253         x.remove(_y)
254     return x
255
256
257 def EvalPrintResults(debug, secondary_track_lists, true_primary_tracks,
258                      true_secondary_bb_track_lists, true_secondary_cc_track_lists,
259                      true_secondary_same_chain_track_lists, true_other_tracks,
260                      MCPrimaryRecoSV, MCOthersRecoSV, MCBottomRecoSV,
261                      MCBottomRecoSVSameChain, MCBottomRecoSVSameParticle,
262                      MCCharmRecoSV, MCCharmRecoSVSameChain,
263                      MCCharmRecoSVSameParticle,
264                      NumPVEvent, NumOthersEvent, NumBBEvent, NumCCEvent,
265                      MCPrimaryRecoSVTrack, MCOthersRecoSVTrack,
266                      MCBottomRecoSVTrack, MCBottomRecoSVSameChainTrack,
267                      MCBottomRecoSVSameParticleTrack,
268                      MCCharmRecoSVTrack, MCCharmRecoSVSameChainTrack,
269                      MCCharmRecoSVSameParticleTrack,
270                      NumPVTrack, NumOthersTrack, NumBBTrack, NumCCTrack):
271
272     true_secondary_bb_tracks = [track for tracks in true_secondary_bb_track_lists
273                                 for track in tracks]
274     true_secondary_cc_tracks = [track for tracks in true_secondary_cc_track_lists
275                                 for track in tracks]
276     true_secondary_same_particle_track_lists = true_secondary_bb_tracks +
277                                               true_secondary_cc_tracks
278     secondary_tracks = [track for tracks in secondary_track_lists for track in
279                         tracks]
280
281
282     tmpMCPrimaryRecoSV = 0
283     tmpMCOthersRecoSV = 0
284     tmpMCBottomRecoSV = 0
```

```
273     tmpMCBottomRecoSVSameChain = 0
274     tmpMCBottomRecoSVSameParticle = 0
275     tmpMCCharmRecoSV = 0
276     tmpMCCharmRecoSVSameChain = 0
277     tmpMCCharmRecoSVSameParticle = 0
278
279     chains = deepcopy(secondary_tracks)
280     particles = deepcopy(secondary_tracks)
281     for secondary_track_list in secondary_track_lists:
282         if len(secondary_track_list) == 0: continue
283         chain_TrueorFalse = []
284         particle_TrueorFalse = []
285         for true_secondary_same_chain_track_list in
286             true_secondary_same_chain_track_lists:
287             chain_TrueorFalse.append(yinx(secondary_track_list,
288                 true_secondary_same_chain_track_list))
289         if not any(chain_TrueorFalse): chains = listremove(secondary_track_list,
290             chains)
291         for true_secondary_same_particle_track_list in
292             true_secondary_same_particle_track_lists:
293             particle_TrueorFalse.append(yinx(secondary_track_list,
294                 true_secondary_same_particle_track_list))
295         if not any(particle_TrueorFalse): particles = listremove(
296             secondary_track_list, particles)
297
298         if len(true_primary_tracks) != 0:
299             for true_primary_track in true_primary_tracks:
300                 if true_primary_track in secondary_tracks: tmpMCPrimaryRecoSV =
301                     tmpMCPrimaryRecoSV + 1
302                 if len(true_other_tracks) != 0:
303                     for true_other_track in true_other_tracks:
304                         if true_other_track in secondary_tracks: tmpMCOthersRecoSV =
305                             tmpMCOthersRecoSV + 1
306                         if len(true_secondary_bb_tracks) != 0:
307                             for true_secondary_bb_track in true_secondary_bb_tracks:
308                                 if true_secondary_bb_track in secondary_tracks: tmpMCBottomRecoSV =
309                                     tmpMCBottomRecoSV + 1
310                                 if true_secondary_bb_track in chains: tmpMCBottomRecoSVSameChain =
311                                     tmpMCBottomRecoSVSameChain + 1
312                                 if true_secondary_bb_track in particles:
```

```
tmpMCBottomRecoSVSameParticle = tmpMCBottomRecoSVSameParticle + 1
303 if len(true_secondary_cc_tracks) != 0:
304     for true_secondary_cc_track in true_secondary_cc_tracks:
305         if true_secondary_cc_track in secondary_tracks: tmpMCCharmRecoSV =
306             tmpMCCharmRecoSV + 1
307         if true_secondary_cc_track in chains: tmpMCCharmRecoSVSameChain =
308             tmpMCCharmRecoSVSameChain + 1
309         if true_secondary_cc_track in particles: tmpMCCharmRecoSVSameParticle
310             = tmpMCCharmRecoSVSameParticle + 1
311
312 print(pycolor.ACCEPT + "-----" +
313       pycolor.END)
314 if len(true_primary_tracks) != 0:
315     tmp_score = tmpMCPPrimaryRecoSV/len(true_primary_tracks)
316     print(pycolor.RED + "MC\u2022Primary\u2022/\u2022Reco\u2022SV\u2022:\u2022" + str(tmp_score) + pycolor
317       .END)
318     NumPVEvent = NumPVEvent + 1
319     MCPPrimaryRecoSV = MCPPrimaryRecoSV + tmp_score
320     MCPPrimaryRecoSVTrack = MCPPrimaryRecoSVTrack + tmpMCPPrimaryRecoSV
321     NumPVTrack = NumPVTrack + len(true_primary_tracks)
322 else: print(pycolor.RED + "MC\u2022Primary\u2022/\u2022Reco\u2022SV\u2022:\u2022[not\u2022exists]" + pycolor.
323       END)
324
325 if len(true_other_tracks) != 0:
326     tmp_score = tmpMCOthersRecoSV/len(true_other_tracks)
327     print(pycolor.RED + "MC\u2022Others\u2022/\u2022Reco\u2022SV\u2022:\u2022" + str(tmp_score) + pycolor.
328       END)
329     NumOthersEvent = NumOthersEvent + 1
330     MCOthersRecoSV = MCOthersRecoSV + tmp_score
331     MCOthersRecoSVTrack = MCOthersRecoSVTrack + tmpMCOthersRecoSV
332     NumOthersTrack = NumOthersTrack + len(true_other_tracks)
333 else: print(pycolor.RED + "MC\u2022Others\u2022/\u2022Reco\u2022SV\u2022:\u2022[not\u2022exists]" + pycolor.END
334   )
335
336 if len(true_secondary_bb_tracks) != 0:
337     tmp_score, tmp_chain, tmp_particle = tmpMCBottomRecoSV/len(
338       true_secondary_bb_tracks), tmpMCBottomRecoSVSameChain/len(
339       true_secondary_bb_tracks), tmpMCBottomRecoSVSameParticle/len(
340       true_secondary_bb_tracks)
```



```
355     return MCPrimaryRecoSV, MCOthersRecoSV, MCBottomRecoSV,
356         MCBottomRecoSVSameChain, MCBottomRecoSVSameParticle, MCCharmRecoSV,
357         MCCharmRecoSVSameChain, MCCharmRecoSVSameParticle, NumPVEvent,
358         NumOthersEvent, NumBBEvent, NumCCEvent, MCPrimaryRecoSVTrack,
359         MCOthersRecoSVTrack, MCBottomRecoSVTrack, MCBottomRecoSVSameChainTrack,
360         MCBottomRecoSVSameParticleTrack, MCCharmRecoSVTrack,
361         MCCharmRecoSVSameChainTrack, MCCharmRecoSVSameParticleTrack, NumPVTrack,
362         NumOthersTrack, NumBBTrack, NumCCTrack
363
364
365
366
367
368
369
370
371
372
373
374
```

```
355     return MCPrimaryRecoSV, MCOthersRecoSV, MCBottomRecoSV,
356         MCBottomRecoSVSameChain, MCBottomRecoSVSameParticle, MCCharmRecoSV,
357         MCCharmRecoSVSameChain, MCCharmRecoSVSameParticle, NumPVEvent,
358         NumOthersEvent, NumBBEvent, NumCCEvent, MCPrimaryRecoSVTrack,
359         MCOthersRecoSVTrack, MCBottomRecoSVTrack, MCBottomRecoSVSameChainTrack,
360         MCBottomRecoSVSameParticleTrack, MCCharmRecoSVTrack,
361         MCCharmRecoSVSameChainTrack, MCCharmRecoSVSameParticleTrack, NumPVTrack,
362         NumOthersTrack, NumBBTrack, NumCCTrack
363
364
365
366
367
368
369
370
371
372
373
374
```

```
358 def PrintFinish(MCPrimaryRecoSV, MCOthersRecoSV, MCBottomRecoSV,
359                 MCBottomRecoSVSameChain, MCBottomRecoSVSameParticle, MCCharmRecoSV,
360                 MCCharmRecoSVSameChain, MCCharmRecoSVSameParticle,
361                 NumPVEvent, NumCCEvent, NumBBEvent, NumOthersEvent):
362     if(NumPVEvent>0): print(pycolor.RED + "MC\u2022Primary\u2022/\u2022Reco\u2022SV\u2022:\u2022" + str(
363         MCPrimaryRecoSV/NumPVEvent) + pycolor.END)
364     else: print(pycolor.RED + "MC\u2022Primary\u2022/\u2022Reco\u2022SV\u2022:[not\u2022exist]" + pycolor.END
365         )
366
367     if(NumOthersEvent>0): print(pycolor.RED + "MC\u2022Others\u2022/\u2022Reco\u2022SV\u2022:\u2022" + str(
368         MCOthersRecoSV/NumOthersEvent) + pycolor.END)
369     else: print(pycolor.RED + "MC\u2022Others\u2022/\u2022Reco\u2022SV\u2022:[not\u2022exist]" + pycolor.END)
370
371     if(NumBBEvent>0):
372         print(pycolor.RED + "MC\u2022Bottom\u2022/\u2022Reco\u2022SV\u2022:\u2022" + str(MCBottomRecoSV/
373             NumBBEvent) + "\u2022Same\u2022Chain\u2022:\u2022" + str(MCBottomRecoSVSameChain/
374             NumBBEvent) + "\u2022Same\u2022Particle\u2022:\u2022" + str(MCBottomRecoSVSameParticle/
375             NumBBEvent) + pycolor.END)
376     else: print(pycolor.RED + "MC\u2022Bottom\u2022/\u2022Reco\u2022SV\u2022:[not\u2022exists]\u2022Same\u2022Chain\u2022:[
377         not\u2022exists]\u2022Same\u2022Particle\u2022:[not\u2022exists]" + pycolor.END)
378
379     if(NumCCEvent>0):
380         print(pycolor.RED + "MC\u2022Charm\u2022/\u2022Reco\u2022SV\u2022:\u2022" + str(MCCharmRecoSV/
381             NumCCEvent) + "\u2022Same\u2022Chain\u2022:\u2022" + str(MCCharmRecoSVSameChain/
382             NumCCEvent) + "\u2022Same\u2022Particle\u2022:\u2022" + str(MCCharmRecoSVSameParticle/
383             NumCCEvent) + pycolor.END)
384     else: print(pycolor.RED + "MC\u2022Charm\u2022/\u2022Reco\u2022SV\u2022:[not\u2022exists]\u2022Same\u2022Chain\u2022:[
385         not\u2022exists]\u2022Same\u2022Particle\u2022:[not\u2022exists]" + pycolor.END)
```

```

375
376 def PrintFinishTrackBase(MCPrimaryRecoSVTrack, MCOthersRecoSVTrack,
377                           MCBottomRecoSVTrack, MCBottomRecoSVSameChainTrack,
378                           MCBottomRecoSVSameParticleTrack,
379                           MCCharmRecoSVTrack, MCCharmRecoSVSameChainTrack,
380                           MCCharmRecoSVSameParticleTrack,
381                           NumPVTrack, NumCCTrack, NumBBTrack, NumOthersTrack):
382
383     if(NumPVTrack>0): print(pycolor.RED + "MC" + str(NumPVTrack) + " MCPrimary"
384                               + "/ RecoSV" + str(MCPrimaryRecoSVTrack/NumPVTrack) + pycolor.END)
385     else: print(pycolor.RED + "MCPrimary / RecoSV[not exist]" + pycolor.END)
386
387     if(NumOthersTrack>0): print(pycolor.RED + "MC" + str(NumOthersTrack) + " MC"
388                               + "Others / RecoSV" + str(MCOthersRecoSVTrack/NumOthersTrack) +
389                               pycolor.END)
390     else: print(pycolor.RED + "MCOthers / RecoSV[not exist]" + pycolor.END)
391
392     if(NumBBTrack>0):
393         print(pycolor.RED + "MC" + str(NumBBTrack) + " MCBottom / RecoSV"
394               + str(MCBottomRecoSVTrack/NumBBTrack) + " SameChain" + str(
395               MCBottomRecoSVSameChainTrack/NumBBTrack) + " SameParticle" + str(
396               MCBottomRecoSVSameParticleTrack/NumBBTrack) + pycolor.END)
397     else: print(pycolor.RED + "MCBottom / RecoSV[not exists] SameChain"
398                 + "not exists] SameParticle[not exists]" + pycolor.END)
399
400     if(NumCCTrack>0):
401         print(pycolor.RED + "MC" + str(NumCCTrack) + " MCCharm / RecoSV"
402               + str(MCCharmRecoSVTrack/NumCCTrack) + " SameChain" + str(
403               MCCharmRecoSVSameChainTrack/NumCCTrack) + " SameParticle" + str(
404               MCCharmRecoSVSameParticleTrack/NumCCTrack) + pycolor.END)
405     else: print(pycolor.RED + "MCCharm / RecoSV[not exists] SameChain"
406                 + "not exists] SameParticle[not exists]" + pycolor.END)

```

---

```

1 if __name__ == "__main__":
2
3 #MaxEvent = 1000
4 MaxSample = -1
5 MaxTrack = 60
6 MaxPrimaryVertexLoop = 3

```

```
7     ThresholdPairSecondaryScoreBBCC = 0.6
8     ThresholdPairSecondaryScore = 0.88
9     ThresholdPairPosScore = 30
10    ThresholdPrimaryScore = 0.55
11    ThresholdSecondaryScore = 0.70
12    debug = False
13
14    NumPVEvent = 0
15    NumOthersEvent = 0
16    NumBBEvent = 0
17    NumCCEvent = 0
18
19    MCPPrimaryRecoSV = 0
20    MC0thersRecoSV = 0
21
22    MCBottomRecoSV = 0
23    MCBottomRecoSVSameChain = 0
24    MCBottomRecoSVSameParticle = 0
25    MCCharmRecoSV = 0
26    MCCharmRecoSVSameChain = 0
27    MCCharmRecoSVSameParticle = 0
28
29    MCPPrimaryRecoSVTrack = 0
30    MC0thersRecoSVTrack = 0
31    MCBottomRecoSVTrack = 0
32    MCBottomRecoSVSameChainTrack = 0
33    MCBottomRecoSVSameParticleTrack = 0
34    MCCharmRecoSVTrack = 0
35    MCCharmRecoSVSameChainTrack = 0
36    MCCharmRecoSVSameParticleTrack = 0
37
38    NumPVTrack = 0
39    NumOthersTrack = 0
40    NumBBTrack = 0
41    NumCCTrack = 0
42
43    data_path = "data/numpy/Vertex_Finder_bb08_reshaped.npy"
44
45    pair_model_name = "Pair_Model_Standard"
46
```

```
47     lstm_model_name = "VLSTM_Model_Standard_PV"
48     slstm_model_name = "VLSTM_Model_Standard_SV"
49
50     print("Data>Loading...")
51     data = np.load(data_path)
52     variables = data[:MaxSample, 3:47]
53
54     MaxEvent = int(data[-1, 0])
55
56     print("Model>Loading...")
57     pair_model, lstm_model, slstm_model = tools.ModelsLoad(pair_model_name,
58                 lstm_model_name, slstm_model_name)
59
60     pred_vertex, pred_position = tools.PairInference(pair_model, variables)
61     data = np.concatenate([data[:MaxSample], pred_vertex], 1)
62     data = np.concatenate([data, pred_position], 1) # -8:NC, -7:PV, -6:SVCC, -5:
63                 SVBB, -4:TVCC, -3:SVBC, -2:Others, -1:Position
64
65     vertices_list = []
66     for ievent in range(MaxEvent):
67         print("====")
68         print("EVENT_NUMBER" + str(ievent))
69         print("====")
70         event_data = [datum for datum in data if datum[0]==ievent]
71         event_data = np.array(event_data)
72         NTrack = (1 + np.sqrt(1 + 8*event_data.shape[0]))/2
73
74         if debug==True: print("The Number of Tracks in this event is" + str(
75             NTrack))
76
77         # Making Tracks / True Labels
78         encoder_tracks, decoder_tracks, true_label, chain_label = tools.
79             GetEncoderDecoderTracksandTrue(debug, event_data, NTrack, MaxTrack)
80
81         # Secondary Seed Selection
82         print("Secondary Seed Selection...")
83         secondary_event_data = tools.SecondarySeedSelectionOne(debug, event_data,
84             ThresholdPairSecondaryScore, ThresholdPairPosScore)
85         #secondary_seed_data = SecondarySeedSelectionTwo(debug,
86             secondary_event_data, )
```

```
81
82     # Primary Vertex Finder
83     print("Primary\u2022Vertex\u2022Prediction\u2022...")
84     primary_track_list, bigger_primary_scores = tools.PrimaryVertexFinder(
85         debug, MaxPrimaryVertexLoop, ThresholdPrimaryScore, event_data,
86         encoder_tracks, decoder_tracks, lstm_model)
87
88     # Secondary Vertex Finder
89     print("Secondary\u2022Vertex\u2022Prediction\u2022...")
90     primary_track_list, secondary_track_lists = tools.SecondaryVertexFinder(
91         debug, ThresholdSecondaryScore, bigger_primary_scores,
92         primary_track_list, secondary_event_data, encoder_tracks,
93         decoder_tracks, slstm_model);
94
95
96     # Result
97     print("Finish\u2022!!")
98     true_primary_tracks, true_secondary_bb_track_lists,
99     true_secondary_cc_track_lists, true_secondary_same_chain_track_lists,
100    true_other_tracks = tools.CountPrintTrueTrackLists(debug, true_label
101        , chain_label)
102
103    tools.PrintPredTrackLists(primary_track_list, secondary_track_lists)
104
105
106    MCPrimaryRecoSV, MCOthersRecoSV, MCBottomRecoSV, MCBottomRecoSVSameChain,
107    MCBottomRecoSVSameParticle, MCCharmRecoSV, MCCharmRecoSVSameChain,
108    MCCharmRecoSVSameParticle, NumPVEvent, NumOthersEvent, NumBBEvent,
109    NumCCEvent, MCPrimaryRecoSVTrack, MCOthersRecoSVTrack,
110    MCBottomRecoSVTrack, MCBottomRecoSVSameChainTrack,
111    MCBottomRecoSVSameParticleTrack, MCCharmRecoSVTrack,
112    MCCharmRecoSVSameChainTrack, MCCharmRecoSVSameParticleTrack,
113    NumPVTrack, NumOthersTrack, NumBBTrack, NumCCTrack
114
115    = tools.EvalPrintResults(debug, secondary_track_lists,
116        true_primary_tracks, true_secondary_bb_track_lists,
117        true_secondary_cc_track_lists, true_secondary_same_chain_track_lists,
118        true_other_tracks, MCPrimaryRecoSV, MCOthersRecoSV, MCBottomRecoSV,
119        MCBottomRecoSVSameChain, MCBottomRecoSVSameParticle, MCCharmRecoSV,
120        MCCharmRecoSVSameChain, MCCharmRecoSVSameParticle, NumPVEvent,
121        NumOthersEvent, NumBBEvent, NumCCEvent, MCPrimaryRecoSVTrack,
122        MCOthersRecoSVTrack, MCBottomRecoSVTrack,
123        MCBottomRecoSVSameChainTrack, MCBottomRecoSVSameParticleTrack,
```

```
    MCCharmRecoSVTrack, MCCharmRecoSVSameChainTrack,  
    MCCharmRecoSVSameParticleTrack, NumPVTrack, NumOthersTrack,  
    NumBBTrack, NumCCTrack)  
98  
99  
100 tools.PrintFinish(MCPrimaryRecoSV, MCOthersRecoSV, MCBottomRecoSV,  
    MCBottomRecoSVSameChain, MCBottomRecoSVSameParticle, MCCharmRecoSV,  
    MCCharmRecoSVSameChain, MCCharmRecoSVSameParticle, NumPVEvent, NumCCEvent  
    , NumBBEvent, NumOthersEvent)  
101 tools.PrintFinishTrackBase(MCPrimaryRecoSVTrack, MCOthersRecoSVTrack,  
    MCBottomRecoSVTrack, MCBottomRecoSVSameChainTrack,  
    MCBottomRecoSVSameParticleTrack, MCCharmRecoSVTrack,  
    MCCharmRecoSVSameChainTrack, MCCharmRecoSVSameParticleTrack, NumPVTrack,  
    NumCCTrack, NumBBTrack, NumOthersTrack)
```

---

# 参考文献

- [1] P. Bambade *et al.*, The International Linear Collider: A Global Project, arXiv preprint arXiv:1903.01629 (2019).
- [2] T. I. Collaboration, International Large Detector: Interim Design Report, arXiv preprint arXiv:2003.01116 (2020).
- [3] T. Behnke *et al.*, The International Linear Collider Technical Design Report - Volume 1: Executive Summary, arXiv preprint arXiv:1306.6327 (2013).
- [4] Ilc photo gallery, <http://ilcgallery.com/>.
- [5] **KEK International Working Group**, K. Desch *et al.*, Recommendations on ILC Project Implementation, (2019).
- [6] H. Baer *et al.*, The International Linear Collider Technical Design Report - Volume 2: Physics, arXiv preprint arXiv:1306.6352 (2013).
- [7] ilcsoft, <https://github.com/iLCSsoft>.
- [8] lcfiplus/lcfiplus: Flavor tagging code for ilc detectors, <https://github.com/lcfiplus/LCFIPlus>.
- [9] T. Suehara and T. Tanabe, LCFIPlus: A framework for jet analysis in linear collider studies, Nuclear Instruments and Methods in Physics Research Section A: Accelerators, Spectrometers, Detectors and Associated Equipment **808**, 109 – 116 (2016).
- [10] F. Gaede, Marlin and LCCD—Software tools for the ILC, Nuclear Instruments and Methods in Physics Research Section A: Accelerators, Spectrometers, Detectors and Associated Equipment **559**, 177 (2006), Proceedings of the X International Workshop on Advanced Computing and Analysis Techniques in Physics Research.
- [11] S. Catani, Y. Dokshitzer, M. Olsson, G. Turnock, and B. Webber, New clustering algorithm for multijet cross sections in e+e – annihilation, Physics Letters B **269**, 432 (1991).
- [12] 斎. 康毅, ゼロから作る Deep Learning: Python で学ぶディープラーニングの理論と実装 ゼロから作る Deep Learning (オライリー・ジャパン, 2016).
- [13] 斎. 康毅, ゼロから作る Deep Learning 2: 自然言語処理編ゼロから作る Deep Learning (オライリー・ジャパン, 2018).
- [14] S. Raschka, クイープ, and 真. 福島, Python 機械学習プログラミング : 達人データサイ

- エンティストによる理論と実践 Impress top gear (インプレス, 2016).
- [15] J. McCarthy, M. Minsky, N. Rochester, and C. E. Shannon, A Proposal for the Dartmouth Summer Research Project on Artificial Intelligence, August 31, 1955., AI Magazine **27**, 12 (2006).
  - [16] V. Vapnik, Pattern recognition using generalized portrait method, Automation and remote control **24**, 774 (1963).
  - [17] B. E. Boser, I. M. Guyon, and V. N. Vapnik, A training algorithm for optimal margin classifiers, in *Proceedings of the fifth annual workshop on Computational learning theory*, pp. 144–152, 1992.
  - [18] F. Rosenblatt, The perceptron: A probabilistic model for information storage and organization in the brain, Psychological Review **65**, 386 (1958).
  - [19] G. CYBENKO, Approximation by superpositions of a sigmoidal function, Mathematics of Control, Signals and Systems **2**, 303 (1989).
  - [20] Tensorflow, <https://www.tensorflow.org/?hl=ja>.
  - [21] Home - keras documentation, <https://keras.io/ja/>.
  - [22] Pytorch, <https://pytorch.org/>.
  - [23] Caffe — deep learning framework, <http://caffe.berkeleyvision.org/>.
  - [24] V. Nair and G. E. Hinton, Rectified Linear Units Improve Restricted Boltzmann Machines., in *ICML*, edited by J. F. Schütz and T. Joachims, pp. 807–814, Omnipress, 2010.
  - [25] B. Widrow and M. E. Hoff, Adaptive Switching Circuits, 1960 IRE WESCON Convention Record , 96 (1960), Reprinted in *Neurocomputing* MIT Press, 1988 .
  - [26] T. Tieleman and G. Hinton, Lecture 6.5—rmsProp: Divide the gradient by a running average of its recent magnitude, COURSERA: Neural Networks for Machine Learning, 2012.
  - [27] D. P. Kingma and J. Ba, Adam: A Method for Stochastic Optimization, arXiv preprint arXiv:1412.6980 (2017).
  - [28] D. E. Rumelhart, G. E. Hinton, and R. J. Williams, Learning representations by back-propagating errors, Nature **323**, 533 (1986).
  - [29] G. E. Hinton and R. R. Salakhutdinov, Reducing the Dimensionality of Data with Neural Networks, Science **313**, 504 (2006).
  - [30] I. J. Goodfellow *et al.*, Generative Adversarial Networks, arXiv preprint arXiv:1406.2661 (2014).
  - [31] S. Hochreiter and J. Schmidhuber, Long Short-Term Memory, Neural Computation **9**, 1735 (1997), <https://doi.org/10.1162/neco.1997.9.8.1735>.
  - [32] K. Cho *et al.*, Learning Phrase Representations using RNN Encoder-Decoder for Statistical Machine Translation, arXiv preprint arXiv:1406.1078 (2014).

- 
- [33] D. Bahdanau, K. Cho, and Y. Bengio, Neural Machine Translation by Jointly Learning to Align and Translate, arXiv preprint arXiv:1409.0473 (2016).
  - [34] M.-T. Luong, H. Pham, and C. D. Manning, Effective Approaches to Attention-based Neural Machine Translation, arXiv preprint arXiv:1508.04025 (2015).
  - [35] A. Vaswani *et al.*, Attention Is All You Need, arXiv preprint arXiv:1706.03762 (2017).
  - [36] Goto-k/vertexfinderwithdl, <https://github.com/Goto-K/VertexFinderwithDL>.
  - [37] W. Kilian, T. Ohl, and J. Reuter, WHIZARD—simulating multi-particle processes at LHC and ILC, The European Physical Journal C **71** (2011).
  - [38] T. Kramer, Track parameters in LCIO, (2006).
  - [39] S. Ioffe and C. Szegedy, Batch Normalization: Accelerating Deep Network Training by Reducing Internal Covariate Shift, arXiv preprint arXiv:1502.03167 (2015).
  - [40] L. van der Maaten and G. Hinton, Visualizing Data using t-SNE, Journal of Machine Learning Research **9**, 2579 (2008).
  - [41] Goto-k/lcfiplus: Flavor tagging code for ilc detectors, <https://github.com/Goto-K/LCFIPlus>.