修士論文

深層学習タスクを対象としたマルチプロセッサ用リアルタイムスケジューリング

石井　響

主指導教員　　田中　清史

北陸先端科学技術大学院大学
先端科学技術専攻
（情報科学）

令和 7 年 3 月

Abstract

In recent years, machine learning such as deep learning has been spreading around various fields. However, it is difficult for embedded systems to execute machine learning. That is because most embedded systems do not have sufficient resources to execute machine learning with high computational load. Therefore, methods to have machine learning processed by edge-servers instead of embedded systems are being considered. In the edge-server, the order of execution must be determined according to the requests from each embedded system. The server must satisfy the predetermined timing constraints (deadlines) of the requests. This is real-time scheduling. In the inference in deep learning, increasing the number of layers improves accuracy but increases computation time. Increasing computation time means that scheduling multiple tasks while satisfying the predetermined timing constraints is difficult. On the other hand, if the computation time is shortened using fewer layers, the accuracy will decrease. Because of this trade-off problem, scheduling deep learning tasks is difficult. In 2020, a method was proposed for scheduling aperiodic deep learning tasks on a single-processor environment. However, a method for scheduling periodic tasks on a multiprocessor environment has not been proposed. Therefore, the objective of this study is to develop an algorithm for real-time scheduling of a set of periodic deep learning tasks on a multiprocessor environment.

Both the previous study and this study's method utilize the neural network model of deep learning with one input layer and multiple output layers (Multi-Exit neural network model). And both studies apply a task model in which the neural network is divided into execution units called stages. This task model is regarded as Imprecise Computation. Imprecise Computation divides the deep learning task into stages that must be executed (Mandatory stages) and stages that are executed if time is available (Optional stages). The proposed algorithm consists of two modules. The first module (Selection module) is a module that selects Optional stages to satisfy the multiprocessor scheduling policy. The second module (Scheduling module) performs scheduling in a multiprocessor environment. In the selection module, I propose an algorithm to select Optional stages that can obtain high accuracy while satisfying the scheduling policy by applying the knapsack problem algorithm. The knapsack problem is a combinatorial optimization problem where the combination of items with weight and value is determined so that the total value is as large as possible, and the total weight is less than or equal to the knapsack's capacity. In this study's method, I define the weight of an item as the processor utilization of the Optional stage, the value of an item as the accuracy gained by the Optional stage, and the knapsack's capacity as the maximum processor utilization (the number of processors) that can be assigned to task's execution. By the above definition, I apply the solution algorithms of the knapsack problem. Typical algorithms for the knapsack problem include approximate solution model (greedy method) and exact solution model (dynamic programming method). I designed two types of solutions for selecting Optional stages: an approximate solution model and an exact solution model. In the scheduling module, I use the existing P-Fair scheduling algorithm. P-Fair scheduling algorithm is the real-time scheduling algorithm for periodic tasks on a multiprocessor environment. This algorithm is a method that decomposes a task into multiple smaller tasks (subtasks) and schedules them in units of subtasks.

As a comparison target for the proposed method, I extend the method of the previous study for a single-processor environment and aperiodic tasks scheduling to the method

for a multiprocessor environment and periodic tasks scheduling. The previous study's algorithm consists of three parts. The first part is the EDF scheduling part of the Mandatory stages, the second part is the selection part of Optional stages, and the third part is the EDF scheduling part of the selected Optional stages. EDF scheduling algorithm is the real-time scheduling algorithm for periodic tasks on a single-processor environment. This algorithm schedules a task with the closest deadline preferentially when events such as arrivals of execution requests and deadlines occur. The comparison target divides the (original) deep learning task set by the number of processors, assigns the divided subset to each processor, and performs the selection part of Optional stages and the EDF scheduling part of Mandatory stages and Optional stages on each processor. In the selection part of the previous study and the comparison target, the combinatorial optimization problem is considered to determine the combination with the minimum execution time among the combinations of Optional stages that can achieve the maximum total accuracy. The combinatorial optimization problem is then solved using a dynamic programming algorithm and the combination of Optional stages is calculated.

Both the proposed and comparison target methods were designed in C programming language in Visual Studio 2022. In this study, the methods were evaluated as a simulation. In the simulation procedure, a set of virtual deep learning tasks was randomly generated and input to the proposed method and the comparison target. I examined the selection results of the Optional stages and measured the processing time of the Selection module. I finally obtained the average value (average accuracy) from the output accuracy of all tasks in a task set. Based on the above procedure, 100 virtual deep learning task sets were randomly generated, and I obtained 100 average accuracies and 100 processing times through the simulation. The average value (final average accuracy and final average processing time) for 100 patterns is the result of the simulation. I examined the relationship between the number of tasks and simulation results (final average accuracy and final average processing time).

First, from the relationship between the number of tasks and final average accuracies, the proposed methods with approximate solution model and exact solution model obtained higher accuracy than comparison target even as the number of tasks increased. Therefore, the results showed that the proposed method obtained a combination of Optional stages with higher accuracy than the comparison target while satisfying the scheduling policy. Therefore, it was found that the proposed method executes deep learning tasks in a way that ensures higher accuracy while scheduling them within their deadlines. Next, from the relationship between the number of tasks and final average processing time, the results showed that the proposed method (approximate solution model) executes in a shorter time than the comparison target.

From the above evaluation, I found that the proposed method (approximate solution model) is the best method for scheduling a set of periodic deep learning tasks on a multiprocessor environment. The future work is to improve the algorithm so that the Selection module of proposed method (exact solution model) can be processed faster. Another work is to develop an algorithm that enables dynamic Optional stage selection and scheduling when task parameters change or when a new task is input. I developed a new real-time scheduling algorithm for a set of periodic deep learning tasks on a multiprocessor environment through this study. I also presented the performance of the developed algorithm. In the future, the developed algorithm is expected to be applied to an edge-server for real-time inference by embedded systems.

目次

第 1 章 はじめに 1

1.1 研究背景 1

1.2 研究課題 2

1.3 研究目的 3

第 2 章 基礎理論 4

2.1 リアルタイムスケジューリング 4

2.2 リアルタイムスケジューリングアルゴリズム 4

2.2.1 非周期・周期タスク 5

2.2.2 プリエンプティブ・ノンプリエンプティブ 6

2.2.3 最適・ヒューリスティック 6

2.2.4 最悪実行時間 7

2.2.5 オフライン・オンラインアルゴリズム 7

2.2.6 シングルプロセッサ版アルゴリズム 7

- Earliest Deadline First スケジューリングアルゴリズム[5] 7

2.2.7 マルチプロセッサ版アルゴリズム 9

- Proportionate Fair スケジューリングアルゴリズム[6] 9

2.3 深層学習 15
2.3.1 ニューラルネットワークの概要 15
2.3.2 畳み込みニューラルネットワーク 16
- 入力層 16
- 隠れ層 17
- 出力層 19

2.4 深層学習タスク 20
2.4.1 Multi-Exit 型ニューラルネットワーク 20
2.5 Imprecise Computation 22
第 3 章 先行研究 24
3.1 全体のアルゴリズム 24
3.2 Optional stage 選択部分 25
- 表 1 行目の計算 26
- 表 2 行目以降の計算 27
- Optional stage の組合せの算出 30

第 4 章 提案手法 31
4.1 全体のアルゴリズム 31
4.2 Optional stage 選択部分 32

4.2.1 近似解法（貪欲法） 33

4.2.2 厳密解法（動的計画法） 36

- 表 1 行目の計算 38
- 表 2 行目以降の計算 38
- 1 番目のステージの計算 39
- 2 番目以降のステージの計算 42
- Optional stage の組合せの算出 47

第 5 章 比較手法 48

5.1 全体のアルゴリズム 48

5.2 タスクのサブセットの割り当て 49

5.3 Optional stage 選択部分の拡張 53

第 6 章 評価実験 57

6.1 タスクセット生成部 57

6.2 実験環境 63

6.3 機能確認 64

6.3.1 提案手法の確認 70

6.3.1.1 近似解法（貪欲法） 70

6.3.1.2 厳密解法（動的計画法） 73

6.3.2 比較手法の確認 74

6.4 性能評価 79

6.4.1 出力精度の評価方法 79

6.4.2 出力精度の評価結果 84

6.4.3 Optional stage 選択の実行時間の評価方法 87

6.4.4 Optional stage 選択の実行時間の評価結果 91

第 7 章 おわりに 97

図目次

1.1.1	エッジコンピューティングの例	2
2.2.1	タスク T_i についてのタイミング図	5
2.2.1.1	非周期タスク（上）と周期タスク（下）の例	6
2.2.6.1	EDF スケジューリング結果	9
2.2.7.1	タスク T（wt(T) = 7/10）から window を生成した例	12
2.2.7.2	サブタスクを式(2.2.7.9)に基づいてグループ分けした結果（点線）	13
2.3.1.1	ニューラルネットワーク	16
2.3.2.1	1000×1000 ピクセルの RGB 画像を入力層へ入力した例	16
2.3.2.2	畳み込みの演算	17
2.3.2.3	活性化関数	18
2.3.2.4	ゼロパディング	19
2.3.2.5	最大値プーリング	19
2.4.1.1	AlexNet に BranchyNet を応用したアーキテクチャ（Multi-Exit 型ニューラルネットワークモデル）	21
2.5.1	ニューラルネットワークモデルを Imprecise Computation に従い分割した例	23
2.5.2	深層学習タスクの概念図	23
3.1.1	先行研究[2]のアルゴリズム	24
3.2.1	動的計画法アルゴリズムで使用する表	25
3.2.2	1 番目のタスク（Task1）の例	26
3.2.3	Task1 の Optional stage のみで合計精度を獲得できる時の最小実行時間	27

3.2.4 2 番目のタスク（Task2）の例 29
3.2.5 各選択肢の値を表中へ図示（先行研究[2]・動的計画法） 29
4.1.1 提案手法の構成 31
4.2.1.1 深層学習タスクのステージの順序関係 34
4.2.1.2 順序関係を保持したまま Optional stage を選択する決定木 35
4.2.2.1 動的計画法アルゴリズムで使用する表 37
4.2.2.2 表の 1 行目の処理 38
4.2.2.3 現在のステージ（1 番目のステージ）のプロセッサ使用率に対してナップサックの容量が小さい時 39
4.2.2.4 ナップサックの容量が現在のステージ（1 番目のステージ）のプロセッサ使用率以上の時 40
4.2.2.5 各選択肢の値を表中へ図示（提案手法・動的計画法・1 番目のステージ） 41
4.2.2.6 現在のステージ（2 番目以降のステージ）のプロセッサ使用率に対してナップサックの容量が小さい時 42
4.2.2.7 ナップサックの容量が現在のステージ（2 番目以降のステージ）のプロセッサ使用率以上の時 43
4.2.2.8 Optional1 が非選択部分（黄色）と選択部分（緑色） 44
4.2.2.9 各選択肢の値を表中へ図示（提案手法・動的計画法・2 番目以降のステージ） 45
4.2.2.10 順序関係の保持のために現在のステージが選択されない状態 46

4.2.2.11 現在ステージ選択決定のための各ステージの選択／非選択フラグ ……46
4.2.2.12 動的計画法アルゴリズムにおける Optional stage の組合せの算出 ……47
5.1.1 比較手法のアルゴリズム ……49
5.2.1 サブセット割り当ての FFD アルゴリズムで空のビンが発生する現象 ……51
5.2.2 空のビンへタスクを移動させる処理 ……52
5.3.1 図 3.2.2 のタスク（Task1）の例（プロセッサ使用率はそれぞれ l%、m%、n%（紫色）） ……53
5.3.2 Task1 の Optional stage のみで合計精度を獲得できる時のプロセッサ使用率の合計 ……54
5.3.3 図 3.2.4 のタスク（Task2）の例（プロセッサ使用率はそれぞれ p%、q%（紫色）） ……55
5.3.4 図 3.2.5 の選択肢に合わせて計算したプロセッサ使用率 ……55
6.1.1 深層学習タスクが持つパラメータ ……58
6.1.2 深層学習タスク（構造体）とパラメータ（メンバ） ……62
6.2.1 実験環境 ……63
6.3.1 タスクセット生成部の結果（Task1～Task5） ……67
6.3.2 タスクセット生成部の結果（Task6～Task10） ……68
6.3.3 タスクセット生成部の結果（Task11～Task12） ……69
6.3.1.1.1 Optional stage 選択部分の結果（提案手法・貪欲法） ……70

6.3.1.1.2 スケジューリング部分の結果（提案手法・貪欲法）（赤色：Mandatory stage、青色：Optional stage）.....72
6.3.1.2.1 Optional stage 選択部分の結果（提案手法・動的計画法）.....73
6.3.1.2.2 スケジューリング部分の結果（提案手法・動的計画法）（赤色：Mandatory stage、青色：Optional stage）.....74
6.3.2.1 Optional stage 選択部分の結果（比較手法・プロセッサ 1）.....75
6.3.2.2 スケジューリング部分の結果（比較手法・プロセッサ 1）（赤色：Mandatory stage、青色：Optional stage）.....76
6.3.2.3 Optional stage 選択部分の結果（比較手法・プロセッサ 2）.....76
6.3.2.4 スケジューリング部分の結果（比較手法・プロセッサ 2）（赤色：Mandatory stage、青色：Optional stage）.....76
6.3.2.5 Optional stage 選択部分の結果（比較手法・プロセッサ 3）.....77
6.3.2.6 スケジューリング部分の結果（比較手法・プロセッサ 3）（赤色：Mandatory stage、青色：Optional stage）.....77
6.3.2.7 Optional stage 選択部分の結果（比較手法・プロセッサ 4）.....78

6.3.2.8 スケジューリング部分の結果（比較手法・プロセッサ 4)（赤色：Mandatory stage、青色：Optional stage） 78
6.4.1.1 100 パターンの乱数の種から最終平均出力精度を算出 80
6.4.1.2 各相対デッドラインのパターンで、3 手法についてタスク数に対する最終平均出力精度を評価 80
6.4.1.3 スケジューリング不可能時の試行のスキップ処理の例 82
6.4.2.1 タスク数変化時の 3 手法の最終平均出力精度（青色：比較手法、橙色：提案手法（近似解法モデル)、緑色；提案手法（厳密解法モデル))（相対デッドライン（短)) 85
6.4.2.2 タスク数変化時の 3 手法の最終平均出力精度（青色：比較手法、橙色：提案手法（近似解法モデル)、緑色；提案手法（厳密解法モデル))（相対デッドライン（中)) 85
6.4.2.3 タスク数変化時の 3 手法の最終平均出力精度（青色：比較手法、橙色：提案手法（近似解法モデル)、緑色；提案手法（厳密解法モデル))（相対デッドライン（長)) 86
6.4.3.1 比較手法での実行時間の計測環境 87
6.4.3.2 提案手法での実行時間の計測環境 88
6.4.3.3 比較手法の Optional stage 選択部分の計測 89
6.4.3.4 提案手法（近似解法モデル）の Optional stage 選択部分の計測 90

6.4.3.5 提案手法（厳密解法モデル）の Optional stage 選択部分の計測 90
6.4.4.1 タスク数変化時の 3 手法の最終平均実行時間（青色：比較手法、橙色：提案手法（近似解法モデル)、緑色；提案手法（厳密解法モデル））（相対デッドライン（短)） 92
6.4.4.2 タスク数変化時の 3 手法の最終平均実行時間（青色：比較手法、橙色：提案手法（近似解法モデル)、緑色；提案手法（厳密解法モデル））（相対デッドライン（中)） 92
6.4.4.3 タスク数変化時の 3 手法の最終平均実行時間（青色：比較手法、橙色：提案手法（近似解法モデル)、緑色；提案手法（厳密解法モデル））（相対デッドライン（長)） 93
6.4.4.4 タスク数の変化に対する動的計画法アルゴリズムの表の面積 94
6.4.4.5 タスク数変化時の 3 パターンの相対デッドラインにおける N*W（提案手法（厳密解法モデル)） 95
6.4.4.6 タスク数変化時の 3 パターンの相対デッドラインにおける最終平均実行時間[ns]（提案手法（厳密解法モデル)） 96

表目次

2.2.6.1	タスクセットの例	8
6.1.1	深層学習タスクが持つパラメータ	57
6.1.2	深層学習タスクのパラメータ計算	60
6.1.3	深層学習タスクのパラメータ計算に使用しているマクロ定数	61
6.2.1	評価実験の使用器材	63
6.3.1	機能確認実験のタスクセット生成部のマクロ定数の設定	65
6.3.2	機能確認実験のタスクセットのパラメータ	66
6.4.1.1	スケジューリング不可能の状態	81
6.4.1.2	性能評価実験で固定しているタスクセット生成部のマクロ定数	82
6.4.1.3	3 パターンの相対デッドラインの変数の設定	83
6.4.1.4	タスクセット生成部における 3 パターンの相対デッドライン	83
6.4.2.1	タスク数変化時の 3 手法の最終平均出力精度（相対デッドライン（短)）	84
6.4.2.2	タスク数変化時の 3 手法の最終平均出力精度（相対デッドライン（中)）	84
6.4.2.3	タスク数変化時の 3 手法の最終平均出力精度（相対デッドライン（長)）	84
6.4.4.1	タスク数変化時の 3 手法の最終平均実行時間[ns]（相対デッドライン（短)）	91
6.4.4.2	タスク数変化時の 3 手法の最終平均実行時間[ns]（相対デッドライン（中)）	91

6.4.4.3 タスク数変化時の 3 手法の最終平均実行時間[ns]（相対デッドライン（長)) ……91
6.4.4.4 タスク数変化時の 3 パターンの相対デッドラインにおける N*W（提案手法（厳密解法モデル)) ……95
6.4.4.5 タスク数変化時の 3 パターンの相対デッドラインにおける最終平均実行時間[ns]（提案手法（厳密解法モデル)) ……95

コード目次

6.1.1 タスクセット生成関数（taskset_generate 関数）の擬似コード ……58

6.1.2 指定した範囲内で乱数を取得する関数（get_random 関数）の擬似コード・・・ ……59

6.1.3 タスクセット生成部の擬似コード ……62

第1章 はじめに

1.1 研究背景

近年、組込みシステムにおける機械学習の需要が増加しており、自動運転や産業用ロボットなどでその導入が進められている。機械学習の中でも深層学習は、従来の機械学習と比較して高い精度を獲得できることから、特に注目を集めている技術である。しかし、組込みシステムは搭載できる資源が限られることから、高い計算資源（マルチプロセッサ環境や大容量メモリ）を必要とする深層学習を組込み機器上で実装することは困難とされている。

そこで、深層学習のプログラムを組込み機器上に実装するのではなく、高い計算資源を持つコンピュータ（サーバ）を用意し、組込み機器で取得したデータをネットワーク経由でサーバへ送信して計算させる処理方法が検討されている[1]。このようなサーバをエッジサーバと呼び、エッジサーバと複数の組込み機器との間でデータを通信して計算する方法をエッジコンピューティングと呼ぶ。類似したサービスにクラウドサーバがあるが、クラウドサーバは範囲が広大で接続される端末数も膨大である。一方でエッジサーバはクラウドサーバと比較して範囲が小規模で、接続される端末数も少ないことが特徴である。そのため、エッジサーバはショッピングモールや工場などの施設に設置して、その施設内の端末群と通信することが想定される。

エッジサーバは、各組込み機器からデータと実行期限を受け取り、サーバ内のニューラルネットワークにより推論を実行する（図 1.1.1）。エッジサーバ内では、各組込み機器からの実行要求を受け取ると、推論に要する計算時間や実行期限をもとに、どの要求から処理するかという実行順序を決定する必要がある。仕事（タスク）の実行順序を決定することをスケジューリングと呼び、決められた期限までにタスクの実行が完了できるように、タスク群の実行順序を決定することをリアルタイムスケジューリングと呼ぶ。組込みシステムは家電製品や産業機器など特定の機能を提供するためのコンピュータシステムであり、その用途から実行期限を守ることが重要である。実行期限を守るなどの時間的制約が課されているシステムのことをリアルタイムシステムと呼ぶ。実行期限を違反することで、重大な危害（機器の故障や人命の損失など）が発生しないシステム（ソフトリアルタイムシステム）や、重大な危害が発生するシス

テム（ハードリアルタイムシステム）などがある。特に、ハードリアルタイムシステムでは、実行期限を守ることが厳しく求められるため、ハードリアルタイムシステムと通信するエッジサーバでは、リアルタイムスケジューリングが極めて重要である。

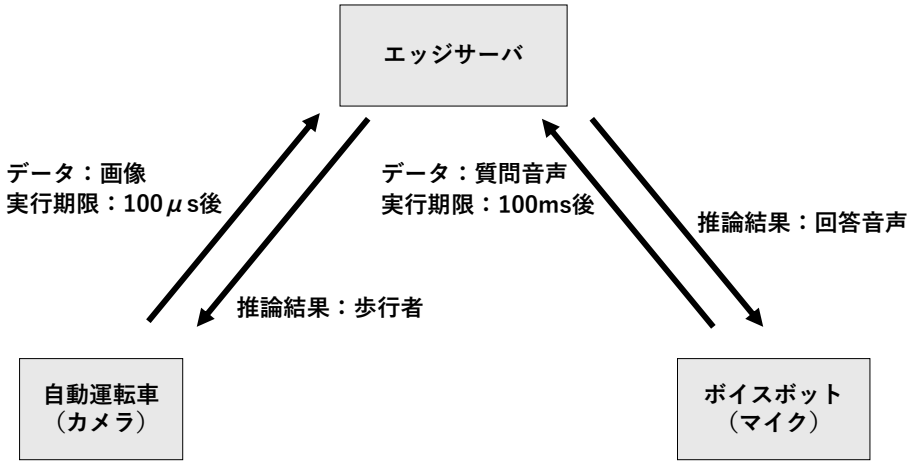

図 1.1.1：エッジコンピューティングの例

1.2 研究課題

ニューラルネットワークとは入力層、隠れ層、出力層で構成されるモデルであり、一般的には中間の隠れ層を多層化したディープニューラルネットワークとして使用されることが多い。ニューラルネットワークでは、層の数を増加すれば精度も向上するが、ニューラルネットワーク全体の計算時間も増加する。即ち、推論により高い精度を得るためには長い計算時間が必要であり、短い計算時間で実行すれば低い精度しか得られないというトレードオフの問題に直面する。

2020 年に、ニューラルネットワークによる推論のタスクをリアルタイムスケジューリングする手法が提案された[2]。しかし、この手法はシングルプロセッサ環境かつイベントドリブン方式の非周期タスク（実行要求が周期的に到来せず、実行要求が突発的に発生したらそれに応じてスケジューリングする方式）という制約があることから、柔軟性に乏しく、エッジサーバなどの高い計算資源の環境に拡張しにくいという課題がある。更に、マルチプロセッサ環境かつ周期・非周期タスクなどの多様な実行要求でスケジューリング可能なアルゴリズムは提案されていない。

1.3 研究目的

本研究では、マルチプロセッサ環境を対象とした、高い精度の獲得と時間的制約の両方を保証するような深層学習タスク群のリアルタイムスケジューリングアルゴリズムを開発することを目的とする。

この開発により、従来困難とされた精度と時間のトレードオフを解消する深層学習タスクのリアルタイムスケジューリングを、マルチプロセッサ環境へ初めて応用させる。更に、先行研究[2]では非周期タスクのみを対象としていたが、本研究では周期タスクも考慮しており、スケジューリングの柔軟性が向上する。

本研究により、マルチプロセッサ環境における深層学習タスク群のリアルタイムスケジューリングの性能や課題を新規に提示でき、今後のスケジューリングや深層学習タスクの研究の指標となる。将来的には、エッジサーバ上にアルゴリズムが応用され、組込みシステムのリアルタイムな推論が可能になることが期待される。

第2章 基礎理論

2.1 リアルタイムスケジューリング

本章では、本研究で関連する基礎理論について述べる。

コンピュータにおいて、プロセスやスレッドなどの特定の処理や作業の単位のことをタスクと呼ぶ。そして、タスク群をプロセッサへどのような順序で割り当て、実行するのかを決定することをタスクスケジューリングと呼ぶ。タスクスケジューリングは、オペレーティングシステム（OS）の機能の一つであり、Windows や Linux などの代表的な汎用 OS でも導入されている。

一方、組込みシステムなどのリアルタイム性が求められるシステムやリアルタイム OS では、「ある時刻までにタスクを完了しなければいけない」などの時間的制約を持っている。時間的制約を持つスケジューリングのことをリアルタイムスケジューリングと呼ぶ。

2.2 リアルタイムスケジューリングアルゴリズム

スケジューリングでは決められた規則の下で、タスク群の実行順序を決定する。この規則がスケジューリングアルゴリズムである。以下では、スケジューリングアルゴリズムを理解するための基本的なパラメータについて述べる。

以下の図 2.2.1 は、一つのタスク T_i について実行状態の変化を示したタイミング図である。タスクに実行要求が到来し、実行可能状態となった時刻のことをリリース（release）r（図 2.2.1 中(1)）、タスクを実行完了しなければいけない時刻のことを絶対デッドライン（absolute deadline）d（図 2.2.1 中(2)）、絶対デッドラインとリリースの差を相対デッドライン（relative deadline）D（図 2.2.1 中(3)）、プロセッサによってタスクが実行されている時間をプロセッサ時間（processor time）C（図 2.2.1 中(4)）、時刻の単位をティック（tick）（図 2.2.1 中(5)）、最小の時間間隔（ティック間のユニット）をスロット（slot）（図 2.2.1 中(6)）とする。プロセッサ時間は実行時間とも呼ばれる。スロット t の時間間隔は[t, t+1)となり、スロット t の開始時刻（ティック）が t である。尚、「絶対デッドライン」については以降、「デッドライン」と呼び、必要に応じて呼び方を使い分けることにする[3]。

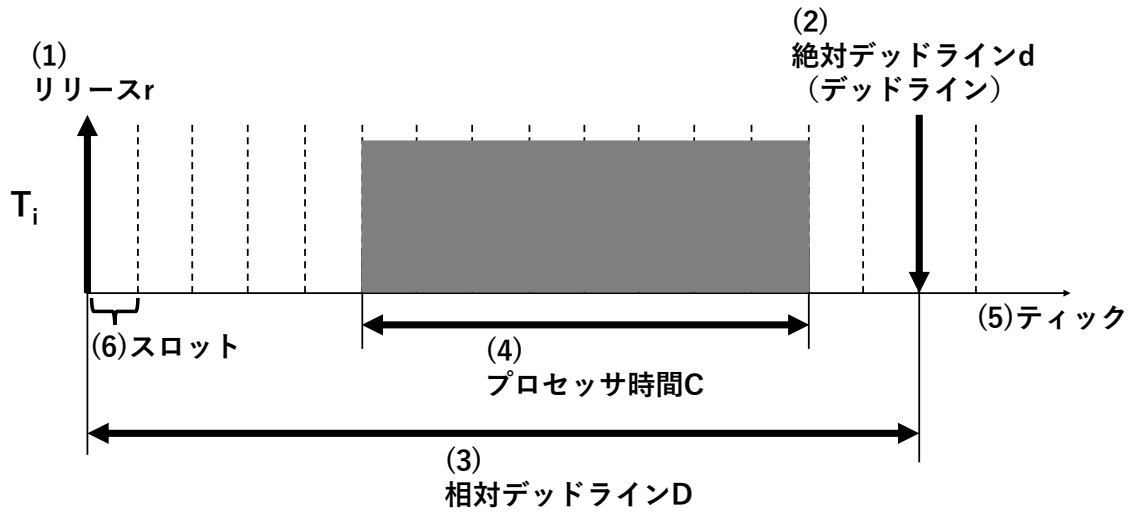

図 2.2.1：タスク T_i についてのタイミング図

2.2.1 非周期・周期タスク

スケジューリングで扱うタスクは、非周期タスクと周期タスクの二種類存在する。

非周期タスクとは、周期的に実行要求が到来しないタスクのことである（図 2.2.1.1)。外部のイベントの発生に伴って実行要求が到来するイベントドリブン方式などが挙げられる。一般的には実行要求の到来に合わせて動的にスケジューリングを行う。

周期タスクとは、周期的に実行要求が到来するタスクのことである（図 2.2.1.1)。一定時間連続して動作させるシステムの場合は、周期タスクとしてスケジューリングを行う。周期タスクはその性質上、周期 P を持つ。図 2.2.1.1 では周期と相対デッドラインが等しいと仮定しており、デッドラインと次のリリースが同時刻のため、両方向の矢印で表現している。そして、周期タスクでは、タスク T_i について、周期 P_i に対するプロセッサ時間 C_i をプロセッサ使用率 U_i と呼ぶ。周期タスクが N 個ある際のプロセッサ使用率の合計は式(2.2.1.1)で示される[3][4]。

$$\sum_{i=0}^{N-1} U_i = \sum_{i=0}^{N-1} \frac{C_i}{P_i} \qquad (2.2.1.1)$$

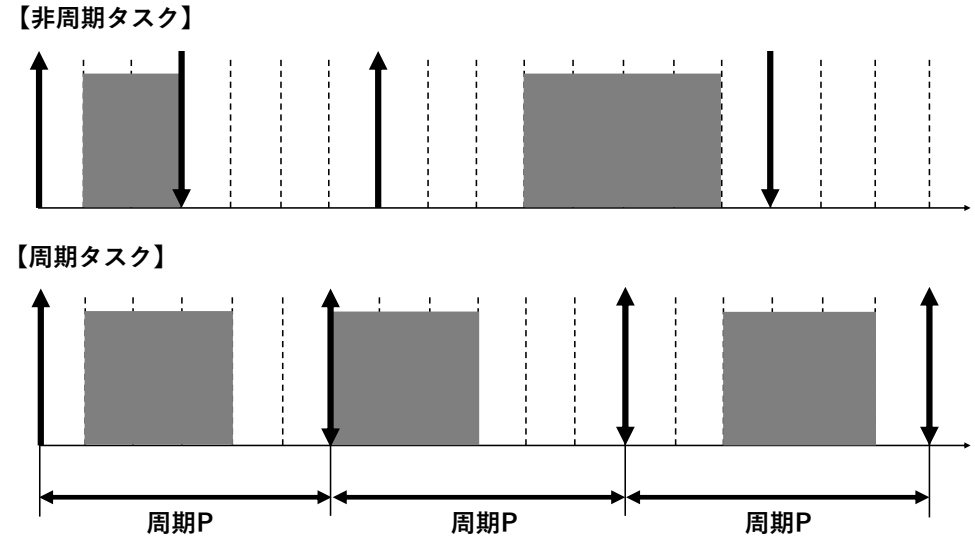

図 2.2.1.1：非周期タスク（上）と周期タスク（下）の例

2.2.2 プリエンプティブ・ノンプリエンプティブ

タスク A の実行中に、より優先度の高いタスク B の実行要求が到来した場合、タスク A の実行を中断し、タスク B に実行を遷移する方式をプリエンプション（preemption）と呼ぶ。また、プリエンプションが発生するようなタスクをプリエンプティブなタスクと呼ぶ。

そして、別のタスクの実行要求が到来しても、現在の実行を中断しないようなタスクをノンプリエンプティブなタスクと呼ぶ[3][4]。

2.2.3 最適・ヒューリスティック

スケジューリング手法の性能評価のためには、平均応答時間やデッドラインミスの数などのスケジューリング評価関数が用いられる。この評価関数の最適値（通常は最小値）が得られる時、そのスケジューリングは最適であるという。一方、最適値は得られないが、実用的な範囲にある時、そのスケジューリングはヒューリスティックであるという[3][4]。

2.2.4 最悪実行時間

タスクの実行時間は、入力データなどの条件によってその長さが変動する。タスクの実行時間が未知の場合、時間的制約を満たすようなスケジューリングは不可能となる。そこで、リアルタイムスケジューリングでは、一般的にタスクの最長の実行時間を想定してスケジューリングが行われる。この最長の実行時間を、最悪実行時間（Worst Case Execution Time）と呼ぶ[3][4]。

2.2.5 オフライン・オンラインアルゴリズム

システムの稼働前にタスクセットのスケジューリングを完了するアルゴリズムを、オフラインアルゴリズムと呼ぶ。オフラインアルゴリズムでは、スケジューリングするタスクセットの情報（プロセッサ時間やデッドライン、周期など）が既知である必要があり、それらの値が変動することを考慮していない。

一方、システムの稼働中にスケジューリングを行うアルゴリズムを、オンラインアルゴリズムと呼ぶ。オンラインアルゴリズムでは、タスクに関するイベント（リリースやデッドライン到来、実行終了など）が発生するごとにスケジューリングを行う。また、新規のタスクが投入された場合や終了された場合にもその都度スケジューリングを行う[3][4]。

2.2.6 シングルプロセッサ版アルゴリズム

シングルプロセッサにおける周期タスクを対象としたオンラインアルゴリズムについて、代表的な一例である Earliest Deadline First アルゴリズム（以下、EDF アルゴリズム）について説明する。EDF アルゴリズムは先行研究[2]で使用されているアルゴリズムである。

- **Earliest Deadline First スケジューリングアルゴリズム[5]**

EDF アルゴリズムは、デッドラインやリリースの到来、タスクの実行終了などのイベントが発生すると、その時点で最も近いデッドラインを持つタスクを優先的にスケジューリングするというアルゴリズムである。EDF アルゴリズムでは、N 個の周期タスクが存在し、式(2.2.6.1)を満たす時、N 個の周期タスクを EDF スケジューリング可能である。

$$\sum_{i=0}^{N-1} \frac{C_i}{P_i} \leq 1 \qquad (2.2.6.1)$$

EDF スケジューリングの例を示す。表 2.2.6.1 は T_1~T_4 のタスクセットと、それぞれの周期とプロセッサ時間を示している。尚、周期と相対デッドラインは等しく、初期リリースは全てティック 0、全てプリエンプティブなタスクで、全ての周期でプロセッサ時間は最悪実行時間と等しいと仮定する。表のタスクセットからプロセッサ使用率を式(2.2.6.1)に基づいて計算すると、0.95425…であることから、スケジューリング可能となる。即ち、全てのタスクの時間的制約は満たされる。

ティックが 0 の時、四つのタスクでリリースが到来したため、実行待ちタスクの中で最も近いデッドラインを調べると、T_3 が該当する。そこで T_3 を実行する。T_3 がティック 1 で終了したため、残りの実行待ちタスクの中で最も近いデッドラインを調べると、T_1 が該当する。そこで、T_1 を実行する。T_1 がティック 4 で終了したため、残りの実行待ちタスクの中で最も近いデッドラインを調べると、T_2 が該当する。そこで、T_2 を実行する。T_2 がティック 6 で終了し、同時に T_3 のリリースが到来したため、残りの実行待ちと新規のタスクの中で最も近いデッドラインを調べると、T_3 が該当する。そこで T_3 を実行する。T_3 がティック 7 で終了したため、実行待ちタスクの中で最も近いデッドラインを調べると、T_4 が該当する。そこで、T_4 を実行する。ティック 8 において T_1 のリリースが到来したため、実行待ちの T_1 と実行中の T_4 のうち最も近いデッドラインを調べると、T_4 が該当するため T_4 の実行を継続する。

上記のようにスケジューリングを実行していき、結果としてタイミング図に表したものを図 2.2.6.1 に示す。

タスク	周期	プロセッサ時間
T_1	8	3
T_2	11	2
T_3	6	1
T_4	13	3

表 2.2.6.1：タスクセットの例

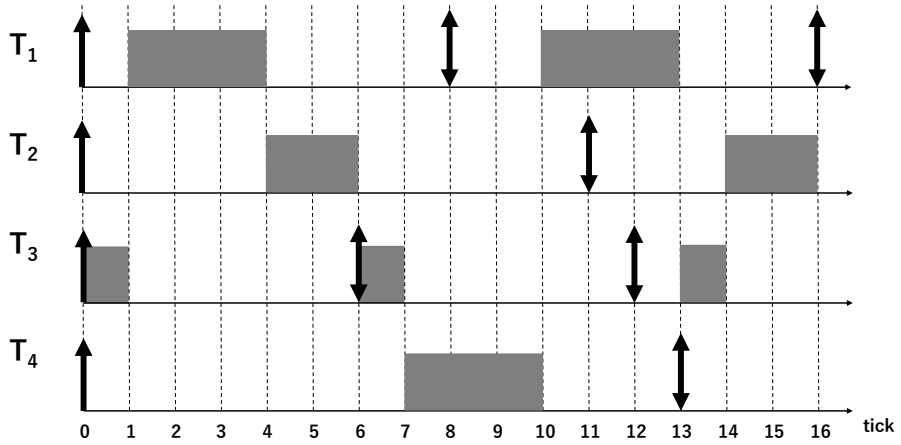

図 2.2.6.1：EDF スケジューリング結果

2.2.7 マルチプロセッサ版アルゴリズム

マルチプロセッサ環境における周期タスクを対象としたオンラインアルゴリズムについて、代表的な一例である Proportionate Fair アルゴリズム（以下、P-Fair アルゴリズム）について説明する。P-Fair アルゴリズムは本研究の提案手法で使用するアルゴリズムである。

- Proportionate Fair スケジューリングアルゴリズム[6]

P-Fair アルゴリズムは、タスクをサブタスクと呼ばれる小タスク群に分解して、逐次プロセッサを割り当てるようにスケジューリングするというアルゴリズムである。サブタスクは特定の実行区間を持ち、各サブタスクはその実行区間内でのみ実行可能である。

タスクの数を N、プロセッサ数を M、タスクセット（集合）を τ、ティックを t とする。P-Fair スケジューリングは周期タスクを対象としているため、周期タスク T の最悪実行時間を T.e、周期を T.p とする。プロセッサ使用率は wt(T) = T.e / T.p であり、タスクの重みとも呼び、$0 < wt(T) \leqq 1$ である。相対デッドラインと周期は等しいとする。

スロットへのタスクの割り当ては、Z をティックの集合とすると、タスクセット τ を用いて式(2.2.7.1)により形式的に定められる。

$$S: \tau \times Z \rightarrow \{0, 1\} \qquad (2.2.7.1)$$

タスク T がスロット t においてスケジューリングされた場合、S(T, t) = 1 となる。また、スロットにおいて同時にスケジューリングできるタスク数はプロセッサ数 M 以下となるため、式(2.2.7.2)を満たす必要がある。

$$\sum_{\forall T \in \tau} S(T,t) \leq M \qquad (2.2.7.2)$$

P-Fair アルゴリズムでは、N 個の周期タスクが存在し、式(2.2.7.3)を満たす時、N 個の周期タスクを P-Fair スケジューリング可能である。

$$\sum_{i=0}^{N-1} \frac{C_i}{P_i} \leq M \qquad (2.2.7.3)$$

P-Fair スケジューリングでは、タスク T は区間[0, t)において、wt(T)×t だけのプロセッサ時間が割り与えられるようにスケジューリングされる。しかし、これは理想的な割り当てであり、実際の割り当てでは、理想的な割り当てになるべく近似するようにプロセッサ時間が割り当てられる。区間[0, t)における理想的な割り当てと実際の割り当ての差を lag として式(2.2.7.4)に定義する。

$$lag(T,t) = wt(T) * t - \sum_{u=0}^{t-1} S(T,u) \qquad (2.2.7.4)$$

式(2.2.7.4)により計算した lag より、式(2.2.7.5)の条件を満たす時、スケジュール S は P-Fair であるという。

$$(\forall T, t :: -1 < lag(T,t) < 1) \qquad (2.2.7.5)$$

式(2.2.7.5)は、タスク T の理想的な割り当てと実際の割り当ての誤差が 1 ティック未満であることを意味する。即ち、各スロット t において、タスク T は $\lfloor wt(T) * t \rfloor$あるいは$\lceil wt(T) * t \rceil$だけ、プロセッサ時間が割り与えられる。

そして、P-Fair アルゴリズムでは、式(2.2.7.5)を満たすようにサブタスクに分解され、それぞれが決められた実行区間でスケジューリングされる。

タスク T の i 番目($i \geq 1$)のサブタスクを T_i とする。各サブタスクは window と呼ばれる実行区間を持ち、この区間以外ではサブタスクは実行できない。window は通常のタスクと同様にリリースやデッドラインを持ち、window における実行可能時刻を擬似リリース（pseudo-release）、実行期限を擬似デッドライン（pseudo-deadline）と呼ぶ。サブタスク Ti の擬似リリース $r(T_i)$を式(2.2.7.6)、擬似デッドライン $d(T_i)$を式(2.2.7.7)に示す。

$$r(T_i) = \left\lfloor \frac{i-1}{wt(T)} \right\rfloor \qquad (2.2.7.6)$$

$$d(T_i) = \left\lceil \frac{i}{wt(T)} \right\rceil \qquad (2.2.7.7)$$

P-Fair アルゴリズムでは、擬似デッドラインが早いサブタスクから優先的にスケジューリングする。また、window の長さを$|w(T_i)| = d(T_i) - r(T_i)$とする。

P-Fair アルゴリズムとして PF[6], PD[7], PD²[8]の三種類が提案されている。これらの違いは、擬似デッドラインが他のタスクと等しい場合にどのタスクを優先するかというものである。このうち PD² が効率的に実装可能であることから、本研究では PD² を実装する。

PD² では、擬似デッドラインが等しい時の判断基準として、(1)b ビット[9]と(2)グループデッドライン[10]の二種類を使用する。

一つ目の判断基準の b ビットとは、サブタスクの window が次のサブタスクの window と重複している時に 1 となり、重複していない時に 0 となる指標である。b ビットが 1 のサブタスクを優先的にスケジューリングする。これは、後続のサブタスクの window と重複しているスロットでスケジューリングされた場合、後続のサブタスクの実行可能なスロットが一つ減ることになり、後続のサブタスクの制約が厳しくなるためである。そこで、b ビットが 1 のサブタスクを優先的にスケジューリングすることによって、後続のサブタスクの実行可能時間を最大限伸ばすことが可能となる。サブタスク T_i の b ビットは式(2.2.7.8)で求められる。サブタスクの添え字は$i \geq 1$である。

$$b(T_i) = \left\lceil \frac{i}{wt(T_i)} \right\rceil - \left\lfloor \frac{i}{wt(T_i)} \right\rfloor \qquad (2.2.7.8)$$

二つ目の判断基準のグループデッドラインとは、サブタスクを連続で実行する際に必ず実行が終了する時刻である。グループデッドラインを考えるために、タスク T の全サブタスクから、サブタスクのグループ T_i, T_{i+1}, …, T_k, …, T_{j-1}, T_j を作ることを考える。グループを作る際の条件としてグループ内のサブタスク T_k $(i < k \leq j)$は以下の式(2.2.7.9)を満たす。

$$\left((|w(T_k)| = 2) \wedge (b(T_k) = 1)\right) \wedge \left(\left(\left|w\left(T_{j+1}\right)\right| = 3\right) \vee \left(b\left(T_j\right) = 0\right)\right) \qquad (2.2.7.9)$$

式(2.2.7.9)は、最初を除くサブタスクの window の長さが 2 かつ b ビットが 1（最初のサブタスクの window の長さの制限はない）、かつ、最後の次のサブタスクの window の長さが 3、あるいは最後のサブタスクの b ビットが 0 となるようなサブタスクのグループに分けることを意味する。

例として、図 2.2.7.1 の 8 つのサブタスクについて考える。このタスク T のプロセッサ時間（最悪実行時間）T.e は 7、周期は 10、したがってプロセッサ使用率 wt(T)は 7/10 である。サブタスクをそれぞれ T_1, T_2, …と添え字を$i \geq 1$とすると、式(2.2.7.6)と式(2.2.7.7)により擬似リリースと擬似デッドラインを計算すると、7 個の window が生成され、各 window ではティック 1 のサブタスクが実行されることになる。図 2.2.7.1 では上矢印（↑）を擬似リリース、下矢印（↓）を擬似デッドラインとしている。

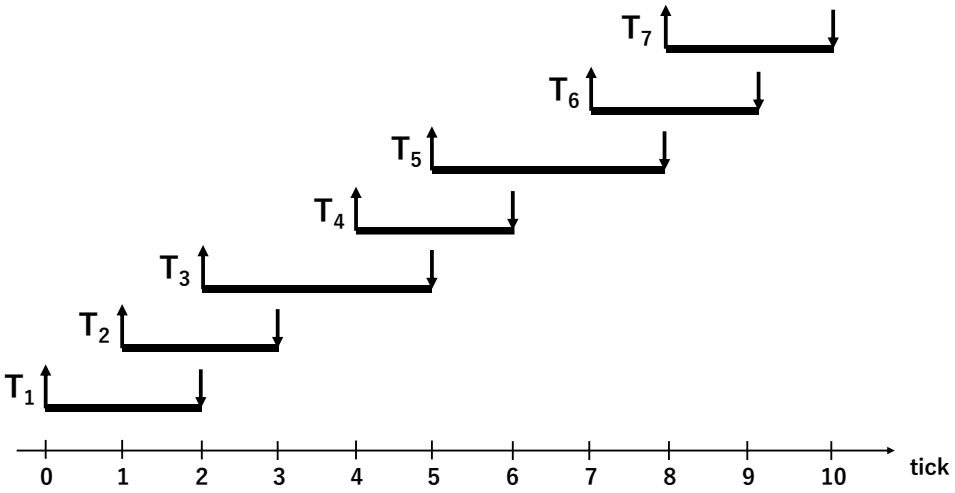

図 2.2.7.1：タスク T（wt(T) = 7/10）から window を生成した例

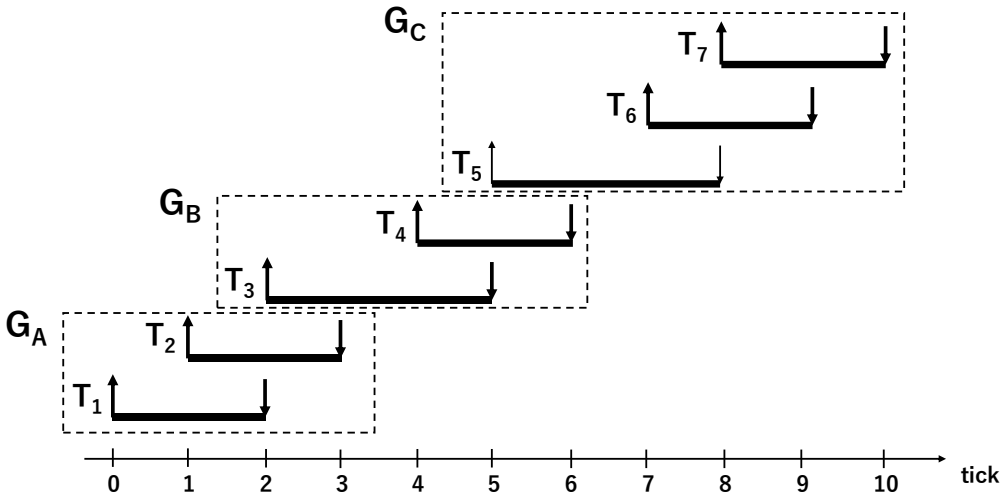

図 2.2.7.2：サブタスクを式(2.2.7.9)に基づいてグループ分けした結果（点線）

そして、式(2.2.7.9)に基づいて、サブタスクをグループ分けすると、図 2.2.7.2 のようになる。図 2.2.7.2 において、点線で囲まれた G_A, G_B, G_C の 3 つのグループが得られている。各グループに注目すると、window の長さが 2 で b ビットが 1 のサブタスクが含まれており、2 つ以上含まれている場合は連続で重複していることが分かる（G_A と G_C）。

G_C のグループを例に考えると、T_5 の最後のスロット（tick = 7）でスケジューリングされた場合、window の長さが 2 の T_6 は実行可能なスロットが残り一つとなり、これによって T_7 のスロットも残り一つとなる。このようなグループは最初のスケジューリングの結果で残りのスケジューリングも立て続けに決まってしまうため、T_5 で擬似デッドラインミスが発生した場合もその影響が後ろのサブタスクへ波及する。

グループデッドラインは立て続けにサブタスクが実行された際に、必ずその実行が終了する時刻である。サブタスク T_i のグループデッドラインを $D(T_i)$ とすると、式(2.2.7.10)により導出できる。

$$D(T_i) = \left\lceil \frac{\left\lceil \left\lceil \frac{i}{wt(T)} \right\rceil \times \left(1 - wt(T)\right) \right\rceil}{1 - wt(T)} \right\rceil \qquad (2.2.7.10)$$

式(2.2.7.10)により、図 2.2.7.2 のサブタスクのグループデッドラインを計算すると、$D(T_1) = D(T_2) = 4$, $D(T_3) = D(T_4) = 7$, $D(T_5) = D(T_6) = D(T_7) = 10$ となり、各グループ G_A, G_B, G_C に対応するデッドラインが得られている。G_A について、T_1 のスロット 1 でスケジューリングされると、T_2 はスロット 2 でスケジューリングされる。T_3 がスロット 3 でスケジューリングされても、T_4 の擬似リリースはまだ到来していないため、4 で終了することになる。G_B について、T_3 のスロット 4 でスケジューリングされると、T_4 はスロット 5 でスケジューリングされる。T_5 がスロット 6 でスケジューリングされても、T_6 の擬似リリースはまだ到来していないため、7 で終了することになる。G_C について、T_5 のスロット 7 でスケジューリングされると、T_6 はスロット 8 でスケジューリングされる。T_6 がスロット 8 でスケジューリングされることで、T_7 はスロット 9 でスケジューリングされるが、後続のサブタスクが存在しないため、10 で終了する。

他のタスクと擬似デッドラインが等しく、グループデッドラインで比較する際は、グループデッドラインが大きいサブタスクを優先的に実行する。グループデッドラインが大きいということは、それに対応するグループが大きいということであり、そのサブタスクで擬似デッドラインミスが発生した際は、後続のサブタスクや次の周期タスクへ影響が大きく波及する。それを避けるために、グループデッドラインが大きいサブタスクを優先的に実行する。

P-Fair アルゴリズムでは、初めに擬似デッドラインが早いかを評価し、擬似デッドラインが等しい場合は次に b ビットについて評価し、b ビットが等しい場合は最後にグループデッドラインについて評価する。

2.3 深層学習

深層学習とは、機械学習の教師あり学習の手法の一つであり、人間の脳の神経回路をモデルにして考案された技術である。深層学習は、サポートベクターマシンや決定木などの教師あり学習とは異なり、特徴量の抽出を自動で行えるという点や、従来の機械学習と比較して高い精度で予測可能という点で、近年注目を集めている。本節では、深層学習の基礎と本研究に関連する理論について述べる。

2.3.1 ニューラルネットワークの概要

深層学習は、図 2.3.1.1 のニューラルネットワークと呼ばれる、神経回路を模したネットワークが使われる。ニューラルネットワークは主に三つの層に分けられ、それぞれ、①入力層、②隠れ層、③出力層と呼ばれる。入力層は、データの入力部であり、画像や文字などのデータが入力される。隠れ層は、入力層から受け取った値をもとに計算する役割を持つ。出力層は、最終的な認識・分類結果を出力する役割を持つ。各層には、ユニット（ニューロン）と呼ばれる計算部があり、特に隠れ層のユニットでは、受け取った複数の入力値に重みを付けて、それを合計値（重み付き和）として計算する。そしてその合計値は各ユニットが持つ閾値を超えた時、次のユニットへ情報を渡している。ニューラルネットワークには、隠れ層を多層化したディープニューラルネットワーク（DNN)、畳み込み層を用いた畳み込みニューラルネットワーク（CNN)、ユニットの出力値を計算に再使用するリカレントニューラルネットワーク（RNN）などがある。

ニューラルネットワークモデルの構築には、一般的な機械学習と同様に訓練データやテストデータなどを用いて学習させる必要がある。学習では正解ラベルと出力値の誤差を損失関数として表現し、この損失関数の値が小さくなるように、隠れ層の重みを調整していく。重みを調整する際は、重みの変化に対する損失関数の大きさの勾配を考え、この勾配が 0 に近づくように調整する（勾配降下法)。この学習を繰り返すことにより最適なニューラルネットワークモデルが完成する。そして、完成されたニューラルネットワークモデルを用いて未知のデータを推論する[11]。

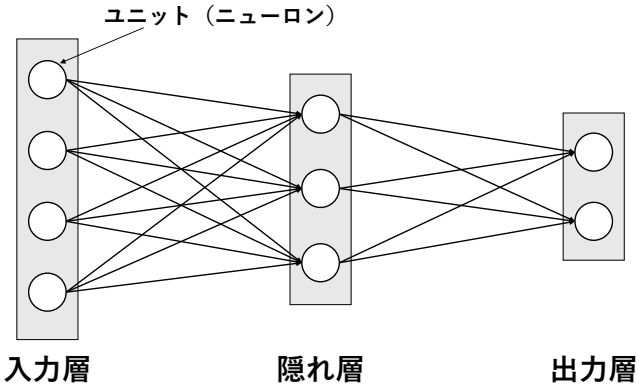

図 2.3.1.1：ニューラルネットワーク

2.3.2 畳み込みニューラルネットワーク

畳み込みニューラルネットワークは、隠れ層に畳み込み層を用いたニューラルネットワークであり、主に画像分類に使用される。ニューラルネットワークには、入力層、隠れ層、出力層の三種類の層があることは前述した[11]。

- 入力層

入力層は画像などのデータが入力される部分である。例として、1000×1000 ピクセルの RGB 画像が入力された場合、入力層のユニットは各ピクセルに対応し、1000×1000×3 = 3000000 個のユニットが存在することになる（図 2.3.2.1）[11]。

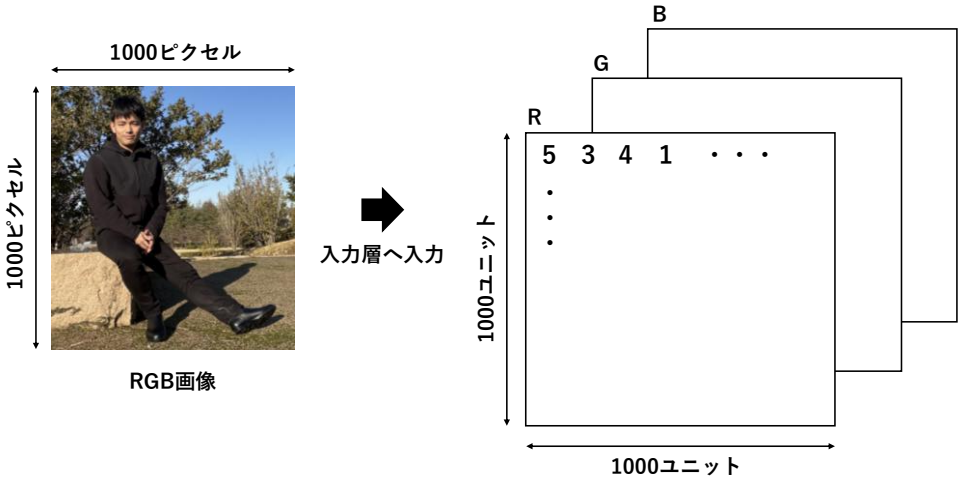

図 2.3.2.1：1000×1000 ピクセルの RGB 画像を入力層へ入力した例

● 隠れ層

隠れ層は入力層や前のユニットから受け取った値をもとに計算し、出力層や次のユニットに渡す役割を持つ。計算とは、重み付け和と閾値の判定であるが、畳み込みニューラルネットワークでは、重み付けは畳み込み層による畳み込みの演算、閾値の判定は活性化関数を指す。畳み込み演算は、入力値に対してフィルタ（カーネル）と呼ばれる領域を重ね、入力値とカーネルの各要素をかけて、最終的に総和を取る。フィルタは入力値の中を移動しながら畳み込み演算を行う。この移動距離をストライドと呼び、要素数だけフィルタを移動させる。入力値とフィルタの畳み込み演算の結果を特徴マップと呼び、入力値からフィルタによって抽出された特徴が数値的に表現されている（図 2.3.2.2）。

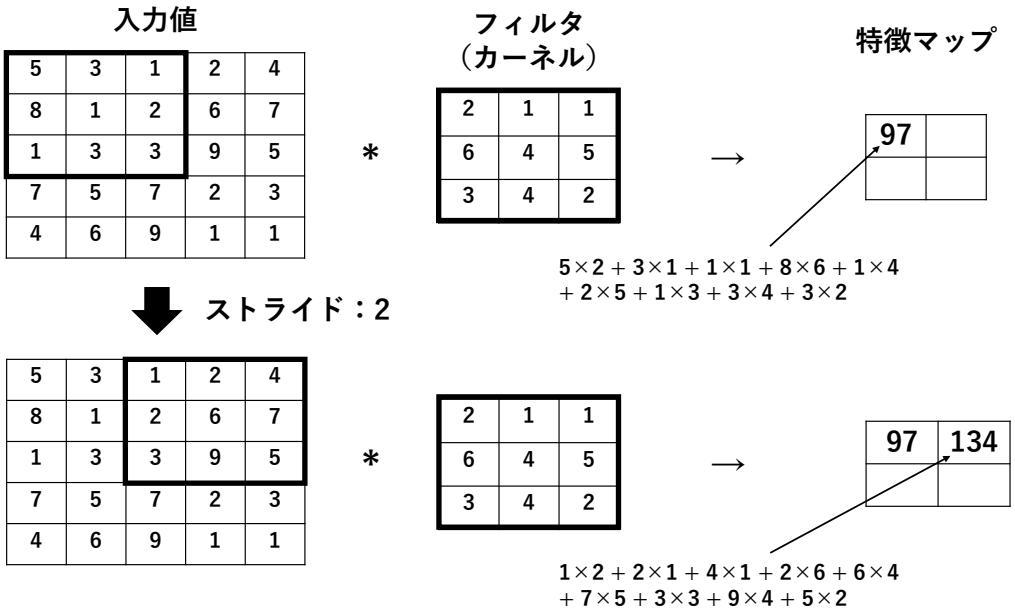

図 2.3.2.2：畳み込みの演算

閾値の判定に活性化関数を使用すると述べたが、重み付け和をもとにどの程度の情報を次のユニットへ渡すかを決定するものである。活性化関数には、ステップ関数、シグモイド関数、ReLU 関数などがある（図 2.3.2.3）。ステップ関数は式(2.3.2.1)、シグモイド関数は式(2.3.2.2)、ReLU 関数は式(2.3.2.3)で定義される。尚、シグモイド関数は a = 1 として図 2.3.2.3 に表示している。

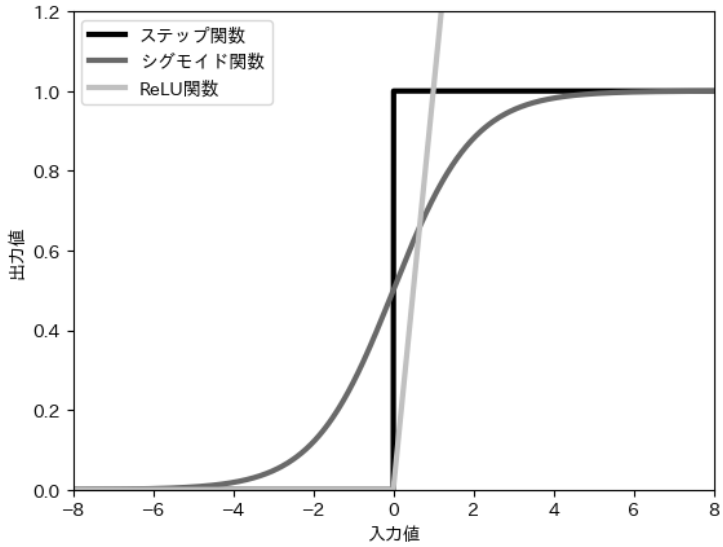

図 2.3.2.3：活性化関数

$$f(x) = \begin{cases} 0, & x < 0 \\ 1, & x \geq 0 \end{cases} \tag{2.3.2.1}$$

$$f(x) = \frac{1}{1 + e^{-ax}} \tag{2.3.2.2}$$

$$f(x) = \begin{cases} 0, & x < 0 \\ x, & x \geq 0 \end{cases} \tag{2.3.2.3}$$

畳み込み層では、フィルタによる畳み込み演算と活性化関数の計算が行われるが、その他にゼロパディングやプーリングの処理も行われる。

ゼロパディングとは、入力値に 0 のユニットを埋め込む処理である。図 2.3.2.2 の畳み込み演算では、入力の縁の値は 1 回のみ畳み込み演算が行われており、演算が繰り返されると入力の縁の情報が欠落してしまう。そこで、図 2.3.2.4 のように入力値の周辺に 0 の行・列を埋め込み、必要な情報を欠落しないようにする処理がゼロパディングである。

プーリングとは、情報を保持しつつデータのサイズを小さくする処理である。プーリングは、入力値にフィルタのような小領域を移動させ、その小領域の最大値を返す最大値プーリングや平均値を返す平均値プーリングなどがある。図 2.3.2.5 に最大値プーリングの例を示す。

畳み込みニューラルネットワークでは、畳み込み層とゼロパディング、プーリングの処理を繰り返すことで、入力の特徴量を伝達していく[11]。

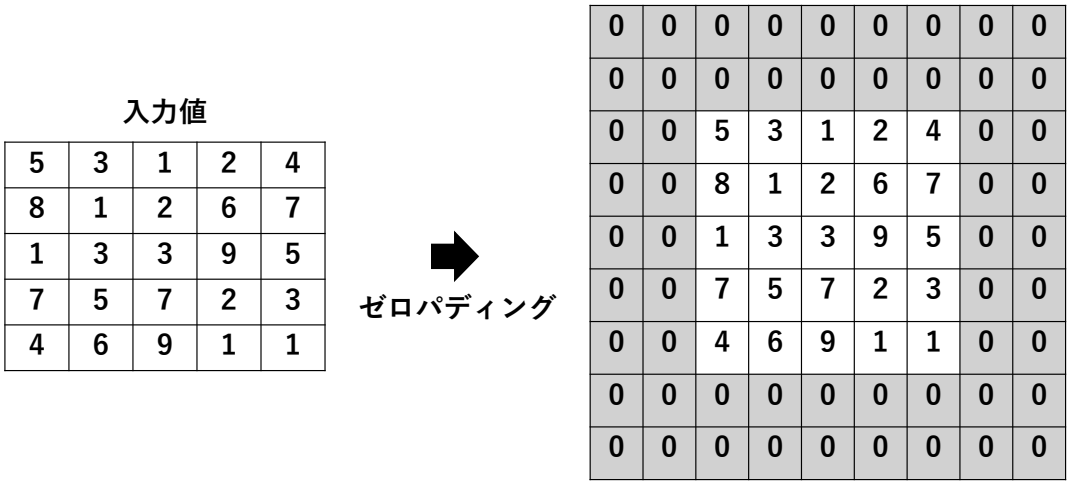

図 2.3.2.4：ゼロパディング

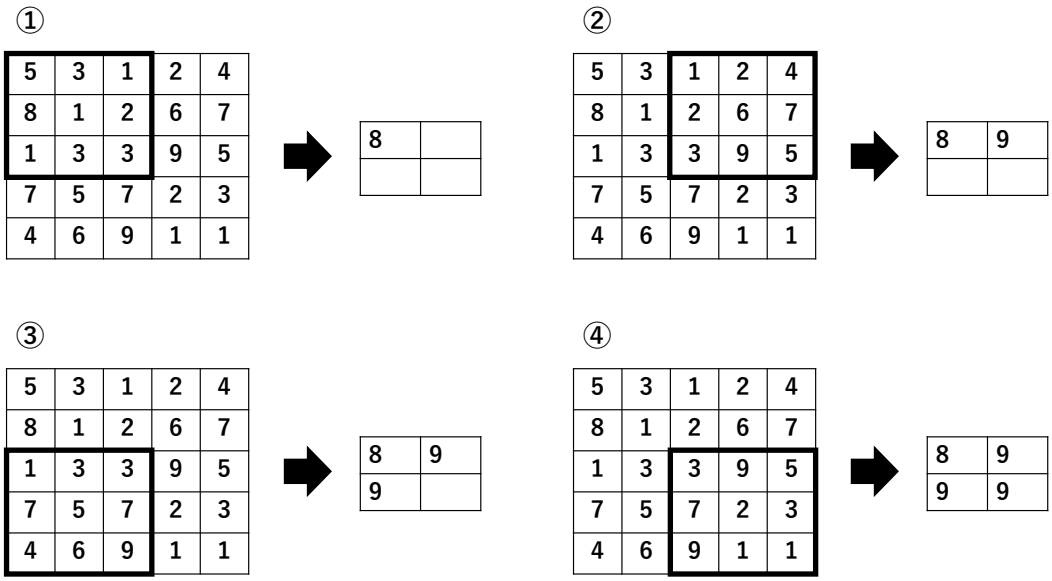

図 2.3.2.5：最大値プーリング

- 出力層

出力層は、隠れ層の特徴マップを、分類の各クラスに対応するユニットに全て接続する全結合層、各ユニットの数値を確率分布に変換する活性化関数である、ソフトマックス関数などで構成される。出力層では、入力データがどのクラスに分類されるかを出力する[11]。

2.4 深層学習タスク

深層学習は、学習と推論の二つの段階で構成される。学習は、訓練データとテストデータによってニューラルネットワークモデルを構築する作業である。推論は、構築されたニューラルネットワークモデルを用いて未知のデータを予測することである。本研究で取り扱う深層学習タスクとは、推論段階のタスクであり、構築されたニューラルネットワークモデルで未知のデータを推論する処理のことをタスクとしている。そのため、タスクの実行時間やデッドラインなどは、学習段階で得られた値をもとに設定すると仮定する。

構築されたニューラルネットワークモデルを用いて推論をする深層学習タスクは、出力層における出力精度とニューラルネットワーク全体の実行時間の間にトレードオフの問題を抱えている。即ち、高い出力精度を確保するためには、ネットワーク内の層数を増加する必要があり、全体の実行時間は長くなる。一方で、全体の実行時間を短くすると、高い出力精度を確保することはできなくなる。深層学習タスクのリアルタイムスケジューリングでは、タスクを最短時間で実行すると、時間的制約は守られるが十分な精度は確保できない。一方で、十分な精度を確保しようとすれば、デッドラインミスが発生し、時間的制約を守れないという問題がある。

2.4.1 Multi-Exit 型ニューラルネットワーク

一般的なニューラルネットワークのアーキテクチャ（AlexNet, VGGNet など）は一つの入力層に対して一つの出力層を持つような構成である。このような構成では、精度の向上のために実行時間の増加が伴うことから、前述のトレードオフの問題に直面する。そこで、2017 年に開発されたニューラルネットワークのアーキテクチャとして BranchyNet[12]がある。BranchyNet では、ネットワークの中間に複数の分岐を設けており、それぞれの分岐は出力層に接続されている。このような構造を設けることによって、要求する精度を早い段階の出力層で確保できる場合は、その段階で実行を終了することが可能となる。図 2.4.1.1 に AlexNet に BranchyNet を応用したアーキテクチャを示す。このような複数の出力層を設けたニューラルネットワークモデルを、Multi-Exit 型ニューラルネットワークモデルと呼ぶ。

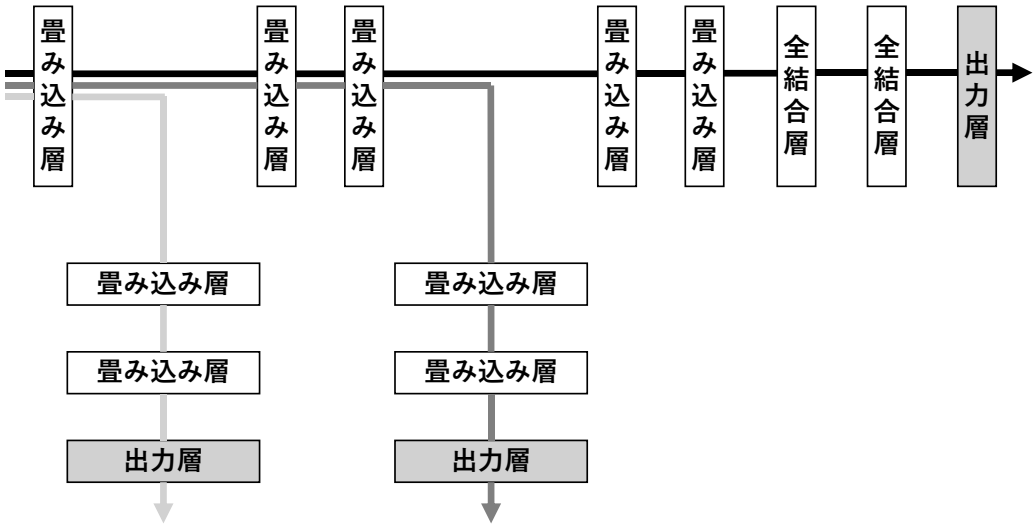

図 2.4.1.1：AlexNet に BranchyNet を応用したアーキテクチャ
（Multi-Exit 型ニューラルネットワークモデル）

深層学習タスクに Multi-Exit 型ニューラルネットワークモデルを応用することによって、時間的制約が厳しい場合は前段の出力層で終了し、その時点での精度を確保できる。一方で、時間的制約に余裕がある場合は後段の出力層で終了し、より高い精度を確保できる。

2.5 Imprecise Computation

Imprecise Computation[13]とは、一つのタスクをステージと呼ばれる実行領域に分割するタスクモデルである。ステージは二種類存在し、必ず実行しなければいけないステージ（Mandatory stage）と、時間的に余裕があれば実行するステージ（Optional stage）である。

深層学習タスクに Multi-Exit 型ニューラルネットワークモデルを応用することにより、時間的制約と精度のトレードオフを解消可能であると前述した。しかし、実行終了する出力層を変更することで実行領域を調整するためには、タスクの中で各出力層に対応する実行領域を作る必要がある。このような深層学習タスクには Imprecise Computation が有効であり、一つの出力層に対応する実行領域をステージとして取り扱える。必要最低限の要求精度を得るために必ず実行しなければいけない領域は Mandatory stage として、時間的制約を守りつつ追加で実行可能な領域を Optional stage として分割できる。

例として図 2.5.1 に構築したニューラルネットワークモデルを Imprecise Computation に従い分割する様子を示す。学習段階で構築されたニューラルネットワークモデルは 4 個の出力層を持っており、それぞれの出力層からは 75%, 85%, 90% 95%の正解率がテストデータにより得られることが分かっているとする。組込み機器が常に 80%以上の精度を要求している場合、4 個のステージのうち、前段の 2 個のステージが Mandatory stages、後段の 2 個のステージが Optional stages となる。

図 2.5.2 に深層学習タスクの概念図を示す。Multi-Exit 型ニューラルネットワークと 1 つの出力層に対応したステージと呼ばれる実行領域を導入した。従って各ステージは出力精度を持つ。更に、一つ前のステージから現在のステージまでの出力精度の増加分を獲得精度と呼ぶことにする。例えば、1 つ目のステージの出力精度が 70%、2 つ目のステージの出力精度が 85%とすると、2 つ目のステージの獲得精度は 15%である。

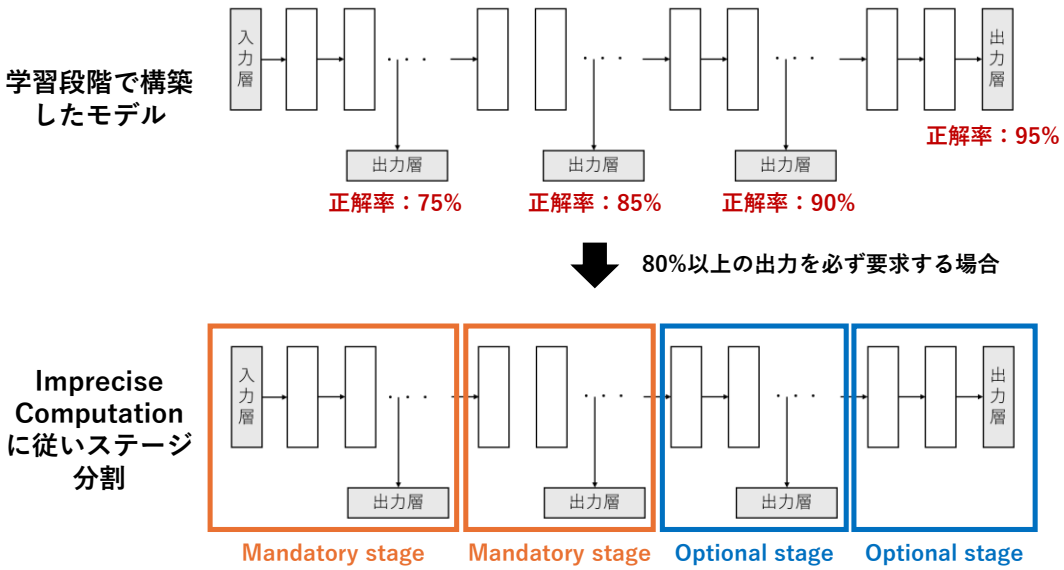

図 2.5.1：ニューラルネットワークモデルを
Imprecise Computation に従い分割した例

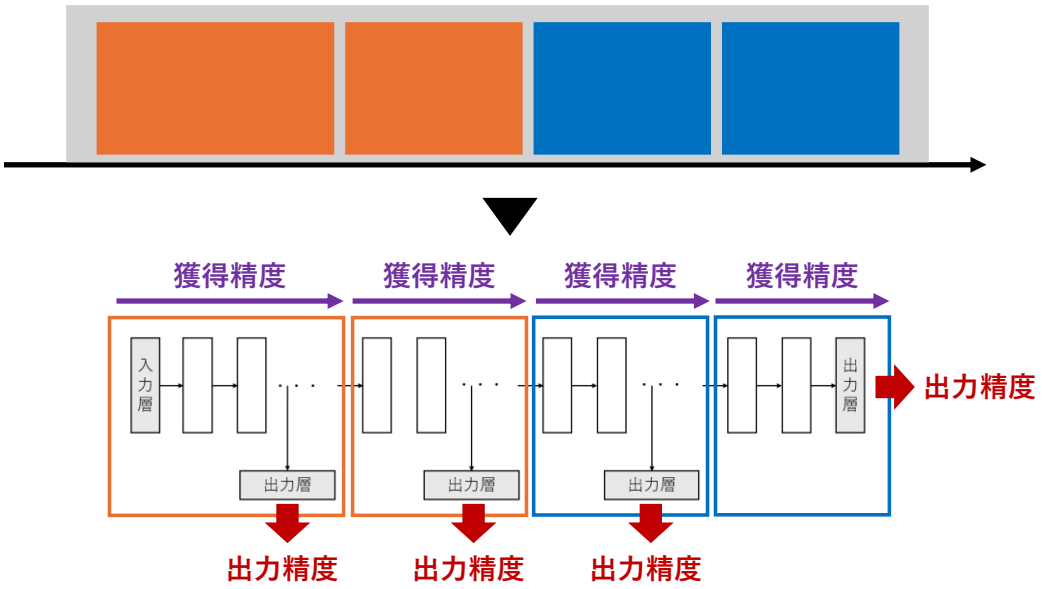

図 2.5.2：深層学習タスクの概念図

第3章 先行研究

3.1 全体のアルゴリズム

本章では、2020 年に提案された、シングルプロセッサ環境かつ非周期タスク版の深層学習タスク群のリアルタイムスケジューリングアルゴリズム[2]について述べる。アルゴリズムでは、初めに深層学習タスクセットが与えられ、デッドラインが近い順にソートする。深層学習タスクは Multi-Exit 型ニューラルネットワークに Imprecise Computation を適用させており、タスクの中をステージ群で分割している。そしてソートした順番でタスクの Mandatory stage からスケジューリングして実行していく。次に、Mandatory stage の実行が終了した後、時間的制約を満たしつつ可能な限り高い精度を得られるように、実行する Optional stage を選択する。最後に、選択された Optional stage をソートした順番でスケジューリングして実行する（図 3.1.1）。

本アルゴリズムで対象としている非周期タスクとはイベントドリブン方式を指す。全てのタスクのリリースは同時刻に到来し、各タスクは実行要求を一回のみ持つ。このようにして先行研究[2]では非周期タスクセットを単純化・理想化している。

図 3.1.1：先行研究[2]のアルゴリズム

3.2 Optional stage 選択部分

先行研究[2]では、デッドラインミスが発生しないように時間的制約を守りつつ、可能な限り高い出力精度を得られるような Optional stage の選択方法が重要となるため、動的計画法のアルゴリズムを使用して Optional stage の最適な組合せを選択している。本節では動的計画法のアルゴリズムについて述べる。

一般的に動的計画法とは、ある問題を複数の部分問題へ分解し、部分問題を利用して解くことによって元の問題の解を明らかにするという手法である。先行研究[2]では、ある合計精度以上を得られる Optional stage の組合せの中で、最も実行時間が短くなるような組合せは何かという問題を解くために動的計画法を用いている。

動的計画法アルゴリズムでは、図 3.2.1 のような表を作成して考える。表の横軸は合計精度、縦軸はタスク数、表の各要素は最小実行時間が入るものとする。動的計画法の表の計算方法は 1 行目と 2 行目以降で異なる。

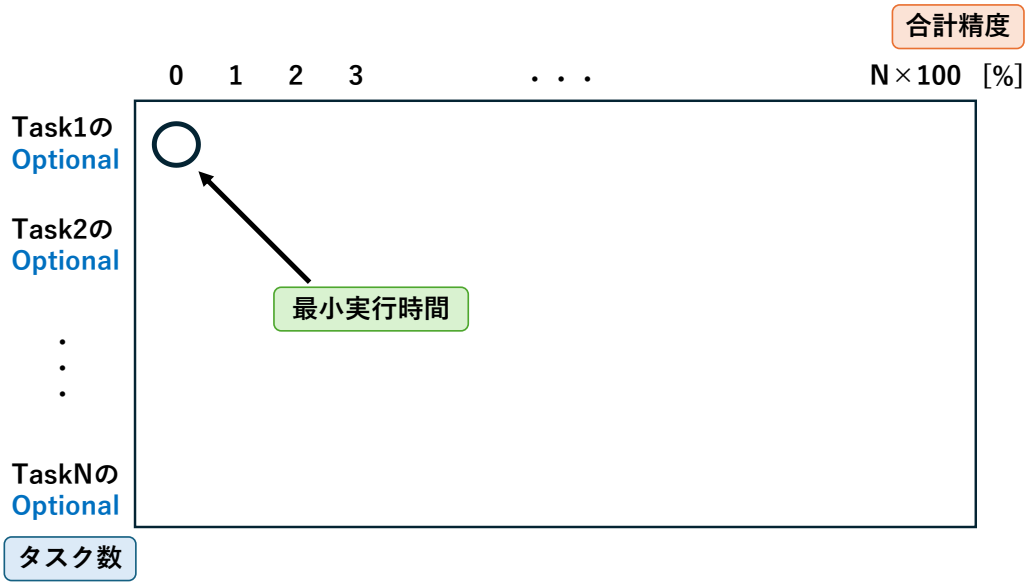

図 3.2.1：動的計画法アルゴリズムで使用する表

- 表 1 行目の計算

動的計画法アルゴリズムの表の 1 行目では、1 番目のタスクの Optional stage で合計精度を獲得できる時の最小実行時間を記入していく。

図 3.2.2 に例として、Optional stage を 3 つ持つ 1 番目のタスク（Task1）を示す。Optional stage はそれぞれ Optional1、Optional2、Optional3 とする。Optional1 の出力精度は a%で実行時間は 1 とする。Optional2 の出力精度は b%で実行時間は 3 とする。Optional3 の出力精度は c%で実行時間は 2 とする。尚、出力精度は$a < b < c$とする。

図 3.2.3 の表より、合計精度が a%以下の時は Optional1 のみを実行することで獲得可能のため、1 行目かつ合計精度が a%以下の要素には Optional1 の実行時間（1）が記入される。次に、合計精度が b%以下の時は Optional1 と Optional2 を実行することで獲得可能のため、1 行目かつ合計精度が a%より上で b%以下の要素には Optional1 と Optional2 の合計実行時間（4）が記入される。最後に、合計精度が c%以下の時は Optional1 と Optional2 と Optional3 を実行することで獲得可能のため、1 行目かつ合計精度が b%より上で c%以下の要素には Optional1 と Optional2 と Optional3 の合計実行時間（6）が記入される。そして、c%より大きい合計精度は Task1 の Optional stage のみでは獲得できないため、獲得不可能を示す×が記入される。

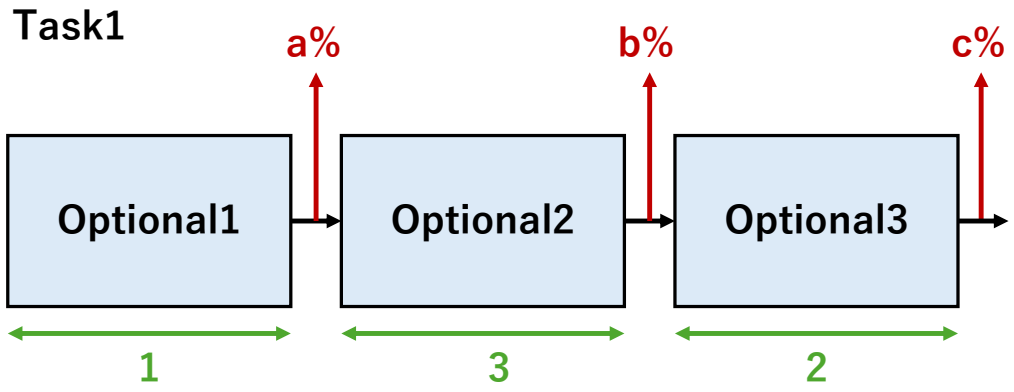

図 3.2.2：1 番目のタスク（Task1）の例

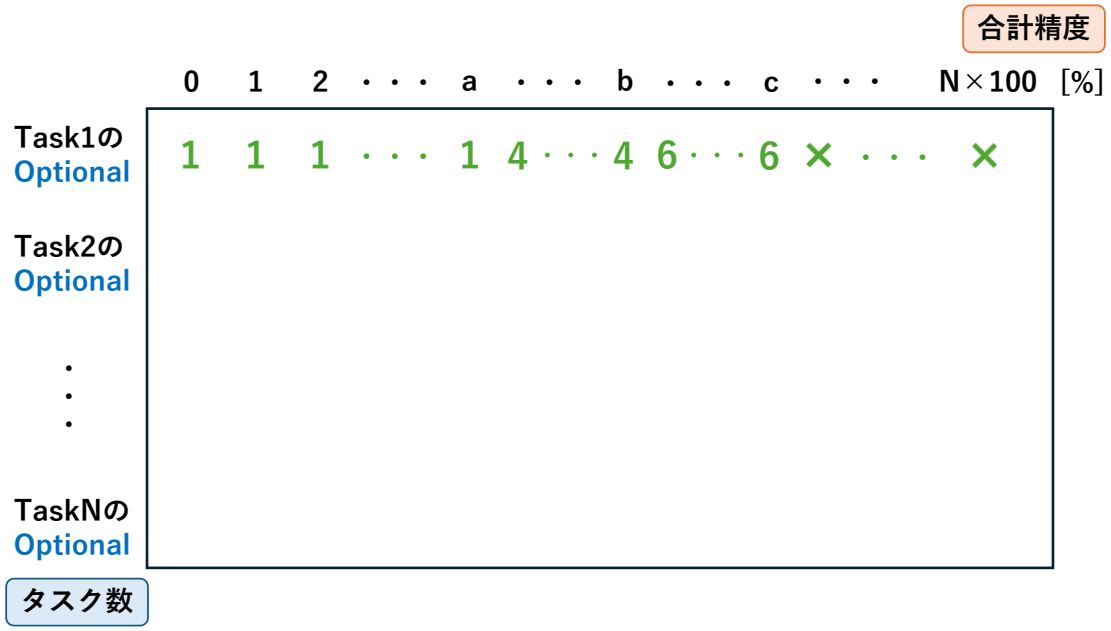

図 3.2.3：Task1 の Optional stage のみで合計精度を獲得できる時の最小実行時間

● 表 2 行目以降の計算

動的計画法アルゴリズムの表の 2 行目以降では、i 番目のタスク時点で合計精度を獲得できる時は 3 つの選択肢の中で最小実行時間を記入する。選択肢とは以下の通りである。

①	Taski の Optional stage のみで合計精度を獲得できる時の実行時間
②	Taski-1 までの Optional stage で合計精度を獲得できる時の実行時間
③	Taski-1 までの Optional stage に Taski の Optional stage を加えることで合計精度を獲得できる時の実行時間

図 3.2.4 に Optional stage を 2 つ持つ 2 番目のタスク（Task2）を示す。Optional stage はそれぞれ Optional4、Optional5 とする。Optional4 の出力精度は d%で実行時間は 1 とする。Optional5 の出力精度は a+d%で実行時間は 3 とする。尚、出力精度は$c < d < a + d$とする。

今、Task2 時点で合計精度 a+d%を獲得できる時の最小実行時間を記入することを考える。初めに、①Task2 の Optional stage のみで合計精度 a+d%を獲得できる時の実行時間を求める。Task2 の Optional stage のみでは、Optional4 と

Optional5 を実行することで a+d%を獲得することができるため、①の合計時間は 4 である。

次に、②Task1 までの Optional stage で合計精度 a+d%を獲得できる時の実行時間を求める。Task1 までの結果は既に計算済みであり、1 行目かつ合計精度が a+d%の要素（現在の真上の要素）を参照すればよい。該当する要素は×が記入されており、Task1 の Optional stage のみで合計精度 a+d%を獲得することはできないことを表している。

最後に、③Task1 までの Optional stage に Task2 の Optional stage を加えることで合計精度 a+d%を獲得できる時の実行時間を求める。今、Task2 は 2 つの Optional stage を持つことから、(i)Task1 までの結果に、Task2 の Optional4 を加えた時、(ii)Task1 までの結果に、Task2 の Optional4 と Optional5 を加えた時の二択が考えられる。Task2 の Optional4 を実行すれば d%の出力精度が得られることから、Task1 までで合計精度 a%を獲得できる時に、Task2 の Optional4 を加えることで a+d%を獲得できる。即ち、1 行目かつ合計精度が a%の要素に Task2 の Optional4 の実行時間を加算した結果である 2 が③の(i)の合計時間となる。また、Task2 の Optional4 と Optional5 を実行すれば a+d%の出力精度が得られることから、Task1 までで合計精度 0%を獲得できる時に、Task2 の Optional4 と Optional5 を加えることで a+d%を獲得できる。即ち、1 行目かつ合計精度が 0%の要素に Task2 の Optional4 と Optional5 の実行時間を加算した結果である 5 が③の(ii)の合計時間となる。

以上をまとめると以下の通りになる。各選択肢の値を表中へ図示したものを図 3.2.5 に示す。

①		Task2 の Optional stage のみで a+d%を獲得できる時の実行時間	4
②		Task1 までの Optional stage で a+d%を獲得できる時の実行時間	×
③	(i)	Task1 までの Optional stage に Task2 の Optional4 を加えることで a+d%を獲得できる時の実行時間	2
	(ii)	Task1 までの Optional stage に Task2 の Optional4 と Optional5 を加えることで a+d%を獲得できる時の実行時間	5

最小実行時間は 2 であり、2 行目の合計精度 a+d%の要素は 2 が記入される。

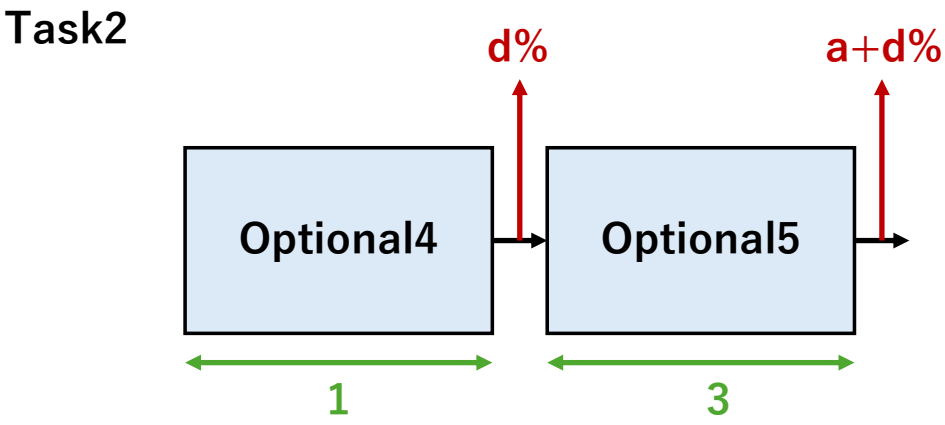

図 3.2.4：2 番目のタスク（Task2）の例

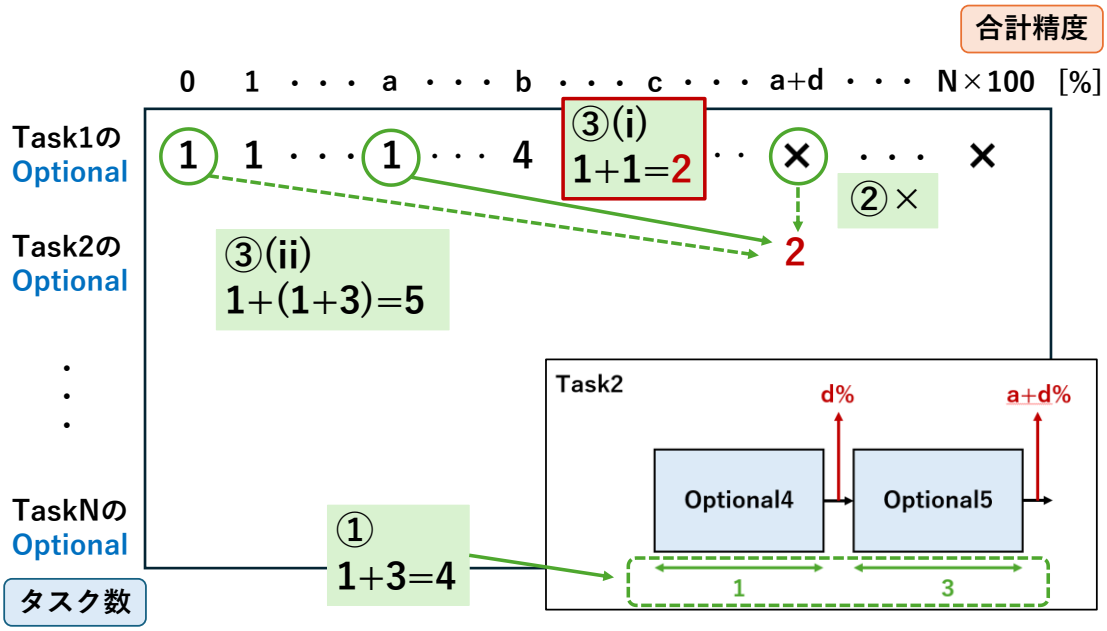

図 3.2.5：各選択肢の値を表中へ図示（先行研究[2]・動的計画法）

- Optional stage の組合せの算出

上記のように全タスクについて合計の最小実行時間を表へ記入していく。尚、合計時間を記入していく際は、各タスクでデッドラインミスが発生しないように確認をしている。即ち、Mandatory stage の実行終了時刻から各タスクのデッドライン以内に収まるように表の（各タスク時点における）合計時間は記入されるものとする。

表の全要素の記入が終了すれば、最終行（N 番目のタスクの行）を走査し、最大の合計精度が得られる時の要素を参照する。この要素には Task1～TaskN までの Optional stage の中で、最大の合計精度を獲得できる時の最小実行時間が記入されている。今回計算し記入したものは最小実行時間のみの表であるが、並行してステージの組合せも記入するように拡張すれば、最小実行時間の時の Optional stage の組合せを算出することができる。

以上から先行研究[2]では、全タスクでデッドラインを守るように時間的制約を満たしつつ、高い合計精度を獲得できるような Optional stage の組合せを選択することができる。

第4章 提案手法

4.1 全体のアルゴリズム

先行研究[2]のアルゴリズムは、シングルプロセッサ環境かつ非周期タスクのみ対象という制約があることから、柔軟性に乏しく、エッジサーバのような多様な要求を対象とする計算環境に拡張しにくいという課題があった。そこで新規にマルチプロセッサ環境かつ周期タスクを対象とした、深層学習タスク群のリアルタイムスケジューリングアルゴリズムを提案する。

提案手法は図 4.1.1 のように二種類のモジュールを持つ構成となっている。時間的制約を満たしつつ高い出力精度を確保するために Optional stage の実行区間を選択する Optional stage 選択部分、既存の P-Fair アルゴリズムを適用することでマルチプロセッサ環境上でリアルタイムスケジューリングを行うスケジューリング部分である。

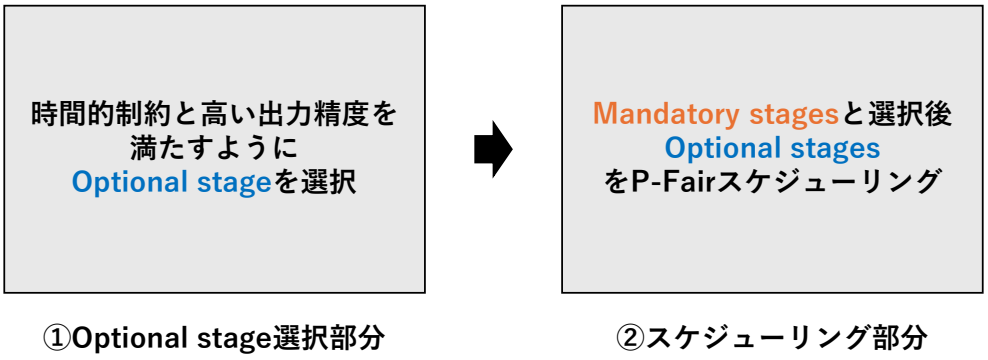

図 4.1.1：提案手法の構成

4.2 Optional stage 選択部分

2.2.7 節の式(2.2.7.3)より、P-Fair スケジューリング可能の条件は以下の通りである。

プロセッサ数が M、周期タスクが N 個存在する時、

$$\sum_{i=0}^{N-1} \frac{C_i}{P_i} \leq M$$

を満たすとき、N 個の周期タスクを P-Fair スケジューリング可能である。

一方で、組合せ最適化問題のナップサック問題とは、以下の通りである。

容量 W のナップサックとアイテムが N 個存在する時、i 番目のアイテムの重さが w_i、価値が v_i であるとする。i 番目のアイテムをナップサックに入れるかを示す状態を $x_i \in \{0, 1\}$とし、0 を入れない状態、1 を入れる状態とする。

$$\sum_{i=0}^{N-1} w_i x_i \leq W$$

を満たしつつ、

$$\sum_{i=0}^{N-1} v_i x_i$$

が最大化するようなアイテムの組合せは何か。

本研究では、ナップサック問題を応用することにより、Optional stage の組合せ最適化問題へ帰着させた。帰着させた問題は以下の通りである。

プロセッサ数が M、プロセッサ使用率と獲得精度（価値）を持つ Mandatory stage と Optional stage がそれぞれ存在する時、

$$\sum (\textit{Optional stage のプロセッサ使用率}) \leq M - \sum (\textit{Mandatory stage のプロセッサ使用率})$$

を満たし、総価値が最大となる Optional stage の組合せは何か。

Optional stage の組合せ最適化問題では、ナップサック問題でのアイテムの重さをステージのプロセッサ使用率で定義し、アイテムの価値をステージの獲得精度で定義している。

このような問題として定義することにより、2 つの利点が存在する。1 つ目がプロセッサ使用率の合計がプロセッサ数 M 以下を満たすようにするため、P-Fair アルゴリズムが適用でき、時間的制約を守りつつマルチプロセッサ環境上でスケジューリングすることができるという点である。2 つ目が、価値を獲得精度で定義していることから、最終的に選択された Optional stage の組合せは、最も合計精度が高い組合せとして得ることができるという点である。

深層学習タスクのリアルタイムスケジューリングの問題点として、時間的制約を満たしつつ高い精度を確保することが難しいというトレードオフの問題があると述べた。しかし、ナップサック問題を応用し、Optional stage の組合せ最適化問題として提案することによって、この問題を解決することができる。

4.2.1 近似解法（貪欲法）

ナップサック問題の代表的な解法の一つに、貪欲法と呼ばれる近似解法がある。本研究では Optional stage 選択部分の組合せ最適化問題に、ナップサック問題の貪欲法を応用した。本節では貪欲法のアルゴリズムについて述べる。

貪欲法のアルゴリズムは以下の通りである。

貪欲法アルゴリズム

①	全 Optional stage について、”獲得精度÷プロセッサ使用率”を算出する。
②	Optional stage を”獲得精度／プロセッサ使用率”の高い順にソートする。
③	プロセッサ使用率の総和が”(プロセッサ数) － (全 Mandatory stage のプロセッサ使用率)”を超えない範囲で、尚且つステージの順序関係を保つようにソートした順に Optional stage を選択する。

ナップサック問題の貪欲法では、各アイテムの”価値÷重さ”を計算し、高い順に選択していく。そのために、提案手法の貪欲法アルゴリズムでは、各 Optional stage の”獲得精度÷プロセッサ使用率”を計算している。Optional stage の獲得精度とは、一つ前のステージの出力精度から現在のステージの出力精度までど

のくらい精度が増加するかを表すものである。また、Optional stage のプロセッサ使用率とは、Optional stage が所属するタスクの周期に対する Optional stage の実行時間を示す。

③について、スケジューリング部分で扱う P-Fair アルゴリズムでは、プロセッサ使用率の総和がプロセッサ数以下の時に最適にスケジューリング可能であることから、Optional stage のプロセッサ使用率の総和が”(プロセッサ数) －(全 Mandatory stage のプロセッサ使用率)”を超えないように選択する必要がある。また、ナップサック問題のアイテムとは異なり、Optional stage には前後の順序関係が存在する（図 4.2.1.1）。そのため、順序関係が破綻しないように Optional stage を選択することが本アルゴリズムの特徴である。

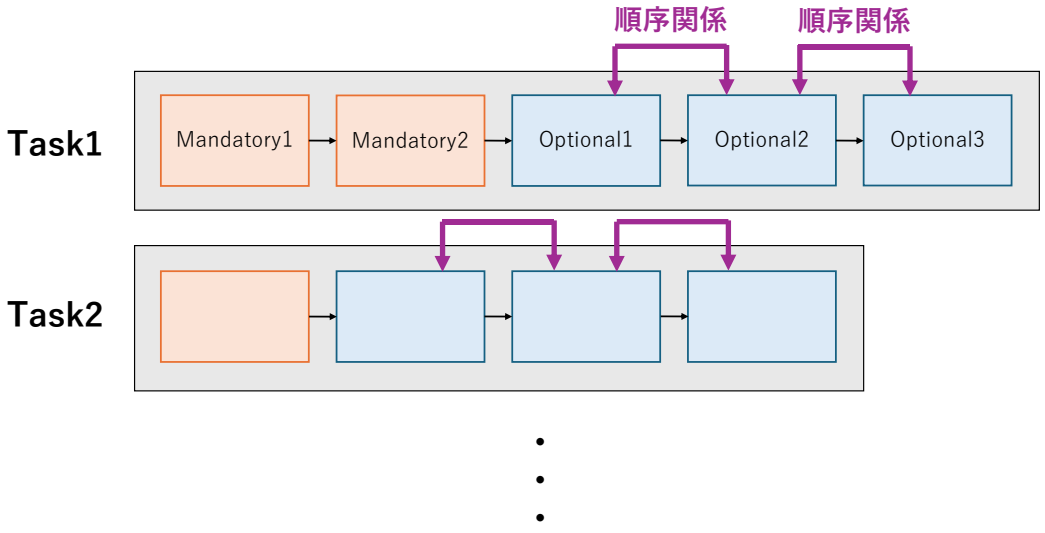

図 4.2.1.1：深層学習タスクのステージの順序関係

前後の順序関係を保持したまま、Optional stage を選択するアルゴリズムについて提案した。ステージは、”選択”と”保留”の 2 つの状態を持ち、選択とは、該当するステージが実行のために選ばれた状態であり、保留とは、以前のステージが選択されていないために一時的に選択を保留している状態である。アルゴリズムでは初めに、” 獲得精度÷プロセッサ使用率”の高い順に Optional stage を調べていく。現在の Optional stage がタスクの 1 番目のステージ(Mandatory stage の直後のステージ)、あるいは 2 番目以降のステージかを調べる。現在のステージが 1 番目のステージの場合はそのステージを選択し、そのステージ以降が保留中かどうかを調べる。保留されているステージがある場合はそれらの

ステージは選択される。現在のステージが 2 番目以降のステージの場合は 1 番目～現在の間のステージが全て選択されているかどうかを調べる。全て選択されている場合は現在のステージも選択できる。一方、全て選択されていない場合は 1 番目のステージが選択されているかどうかを調べる。1 番目のステージが選択済みであり、1 番目～現在の間のステージが保留中の場合、保留中のステージは全て選択され現在のステージも選択される。1 番目のステージが選択されていない場合、または 1 番目のステージが選択されているが 1 番目～現在の間のステージが保留されていない場合は、現在のステージを選択することはできない（ステージの選択が断続的となってしまうため）。そのため、現在のステージは保留となる。

以上の前後の順序関係を保持したまま Optional stage を選択するアルゴリズム（決定木）は図 4.2.1.2 に示される。

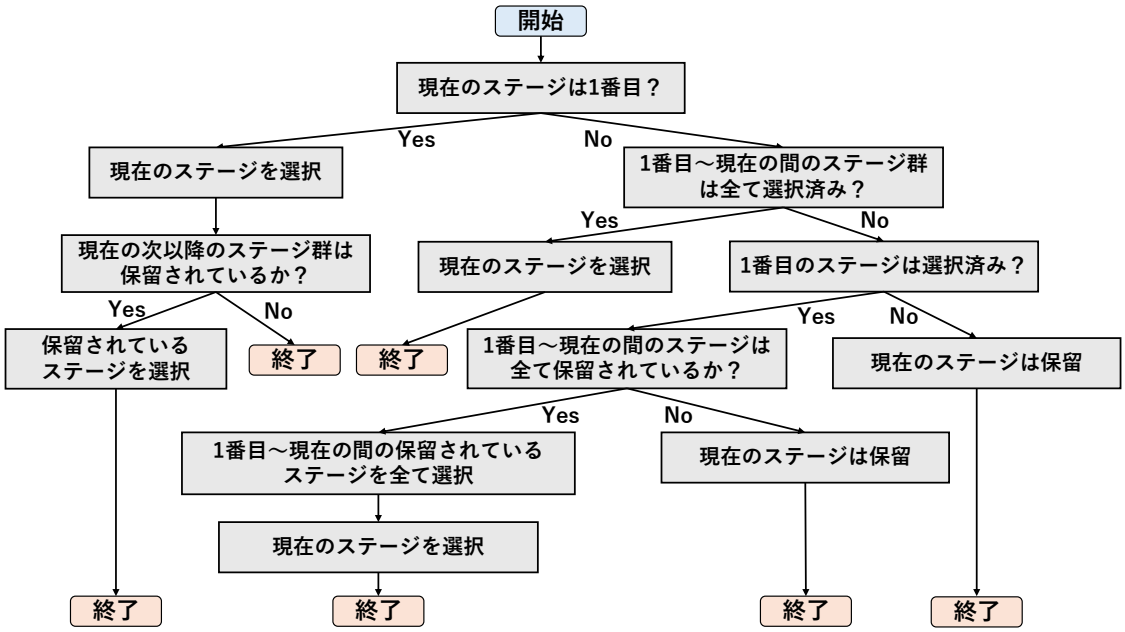

図 4.2.1.2：順序関係を保持したまま Optional stage を選択する決定木

以上の貪欲法アルゴリズムによって、Optional stage の組合せが算出される。得られた組合せについて、プロセッサ使用率の合計は”(プロセッサ数) －（全 Mandatory stage のプロセッサ使用率)”となっており、既存の Mandatory stage と Optional stage 選択部分で得られた Optional stage 群を合わせて P-Fair スケジューリングすることが可能である。

4.2.2 厳密解法（動的計画法）

ナップサック問題の代表的な解法には、前述の貪欲法とは別に、動的計画法と呼ばれる厳密解法がある。本研究では Optional stage 選択部分の組合せ最適化問題に、ナップサック問題の動的計画法を応用した。本節では動的計画法のアルゴリズムについて述べる。

動的計画法のアルゴリズムは以下の通りである。

動的計画法アルゴリズム

①	縦軸が Optional stage 数、横軸がプロセッサ使用率 j（ナップサックの容量)、表の要素を獲得精度とする表を用意する。尚、横軸の最大値 W は”(プロセッサ数) －（全 Mandatory stage のプロセッサ使用率)”とする。
②	表の 1 行目はステージなしとするため、表の要素は 0 で初期化する。
③	表の 2 行目以降について、現在のステージ時点についてナップサックの容量を変化させた時の、最大となる価値を記入していく。
③-①	ナップサックの容量 j が現在のステージのプロセッサ使用率 u より小さい時、1 つ前のステージまでで容量 j のナップサックに入れる時の最大価値（上の要素）を入力する。
③-②	ナップサックの容量 j が現在のステージのプロセッサ使用率 u 以上の時、 (i)1 つ前のステージまでで容量 j のナップサックに入れる時の最大価値 (ii)1 つ前のステージまでで容量 j-u のナップサックに入れる時の最大価値に現在のステージの価値を加えた和 以上の(i)と(ii)のうち大きい値を記入する。尚、(ii)により現在のステージを加えることで前後の順序関係が破綻する場合は、(i)を選ぶ。
④	③の計算を繰り返し、表の最終行（最後のステージの行）を入力し終えると、最終行の横軸の最大値 W の時の要素を参照する。この要素には、容量 W のナップサックへ全ての Optional stage から入れる組合せを考える時、最大となる総価値が記入されている。

動的計画法アルゴリズムでは、図 4.2.2.1 のような表を作成して考える。表の縦軸は Optional stage を表し、縦軸の最大値は全 Optional stage 数の N、横軸はナップサックの容量を表し、各値はプロセッサ使用率×100 としており、横軸の最大値 W は(プロセッサ数) －（全 Mandatory stage のプロセッサ使用率)としている。表中の要素には総価値の最大値が入り、価値はステージの獲得精度としている。

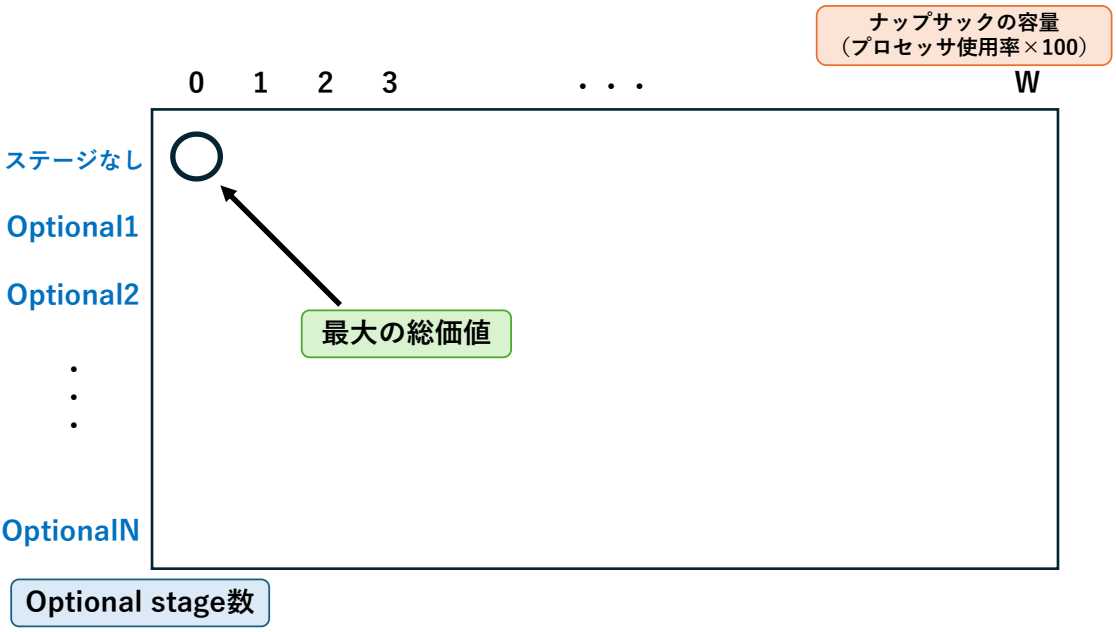

図 4.2.2.1：動的計画法アルゴリズムで使用する表

動的計画法アルゴリズムで使用する表の計算手順について、順に説明する。

● 表 1 行目の計算

表の 1 行目は、ステージが無い時の最大の総価値を記入していく。ステージが無い状態のため 1 行目の要素は全て 0 で初期化される。

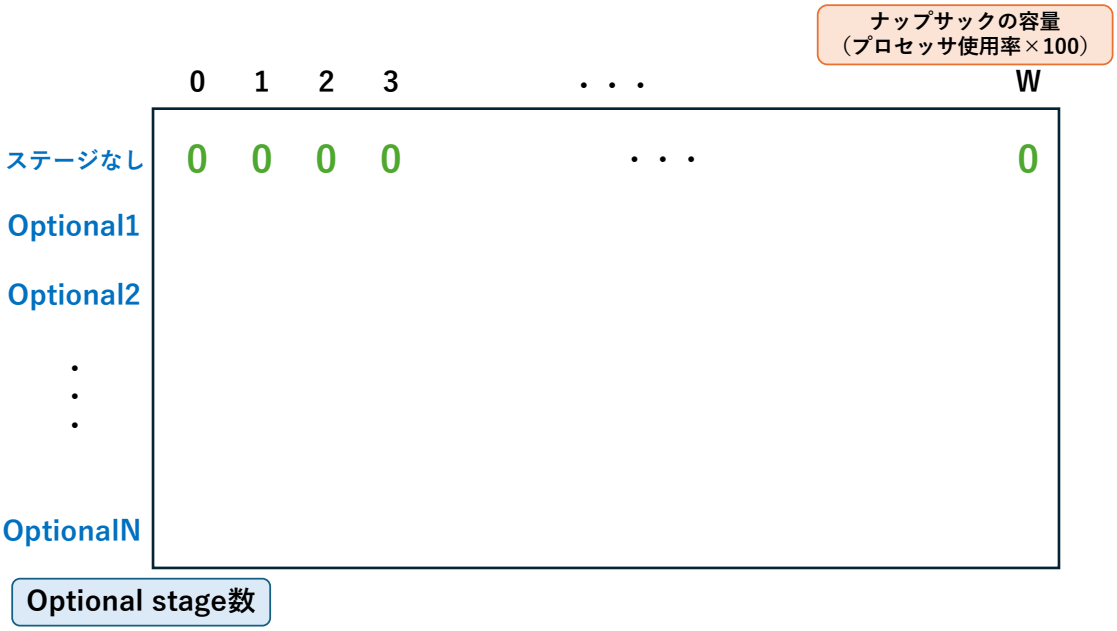

図 4.2.2.2：表の 1 行目の処理

● 表 2 行目以降の計算

表の 2 行目以降では、縦軸のステージ i までを候補として、横軸の容量 j のナップサックへ入れるステージの組合せを考えている。最終的に総価値が最大となる組合せが選ばれ、その時の総価値が表の要素へ記入される。そして、通常のナップサック問題では各アイテムは完全に独立しているためアイテム間で依存関係は存在しないが、今回取り扱っているステージでは前後の順序関係が存在する。そのため、貪欲法アルゴリズムと同様、動的計画法アルゴリズムでも順序関係が破綻しないように改良を加えている。動的計画法アルゴリズムでは現在のステージがタスクの 1 番目のステージ（Mandatory stage 直後のステージ）、あるいは 2 番目以降のステージで分岐を設けている。

● 1 番目のステージの計算

表の計算では、ステージに対応する行について横軸のナップサックの容量を変化させた時の組合せを調べていく。初めに、現在のステージのプロセッサ使用率に対してナップサックの容量が小さい場合である。この場合、現在のステージはナップサックへ入れることができないため、1 つ前のステージまででナップサックへ入れた時の最大の総価値（真上の要素）を現在のステージの要素へ記入する。例として、1 番目のステージである Optional1 を考えているとする。このステージのプロセッサ使用率は U_1、価値（獲得精度）は V_1 とする。Optional1 のプロセッサ使用率に対してナップサックの容量が小さい場合は、図 4.2.2.3 の横軸を紫色で塗りつぶした部分となる。これに対応する 2 行目の要素には真上の 1 行目の要素が記入される。

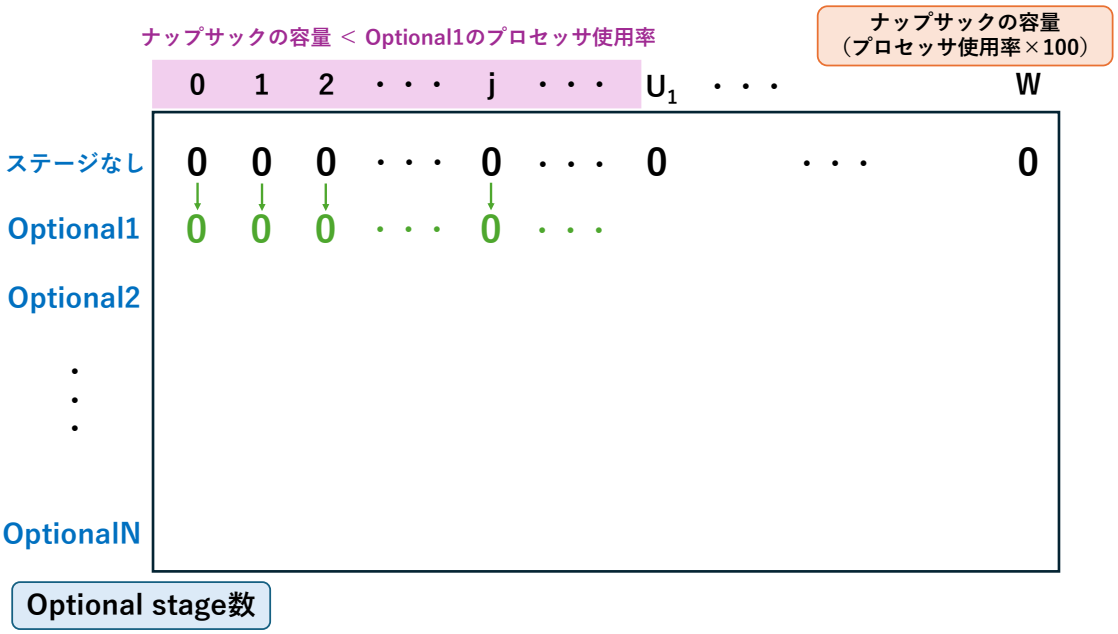

図 4.2.2.3：現在のステージ（1 番目のステージ）のプロセッサ使用率に対してナップサックの容量が小さい時

次に、ナップサックの容量が現在のステージのプロセッサ使用率以上の場合である。この場合、現在のステージはナップサックへ入れることができるため、ナップサックへの入れ方として考えられるのは以下の 2 つの選択肢である。

①　1 つ前のステージまでの組合せで、容量 j のナップサックへ入れる方法 ②　1 つ前のステージまでの組合せに、現在のステージを加えて容量 j のナップサックへ入れる方法

以上の 2 つの選択肢のうち、総価値が高い組合せが採用され、その時の総価値を要素へ記入する。先の例として、1 番目のステージである Optional1 を考えており、このステージのプロセッサ使用率は U_1、価値（獲得精度）は V_1 とする。ナップサックの容量が Optional1 のプロセッサ使用率以上の場合は、図 4.2.2.4 の横軸を紫で塗りつぶした部分となる。

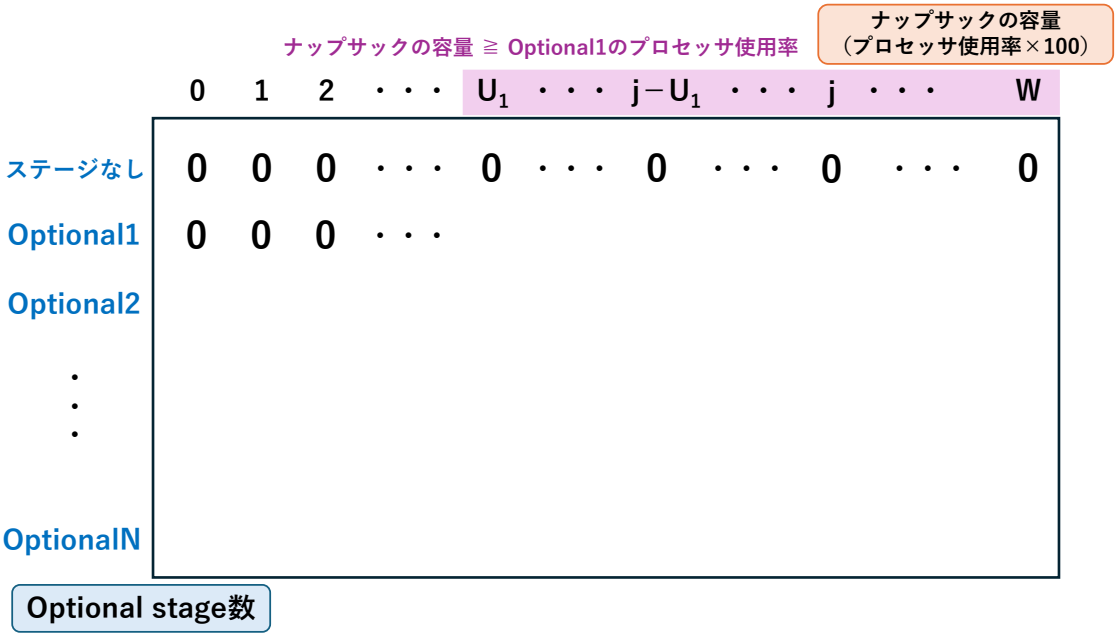

図 4.2.2.4：ナップサックの容量が現在のステージ（1 番目のステージ）のプロセッサ使用率以上の時

2 行目の Optional1 でナップサックの容量が j の時の要素を記入することを考える。この要素に記入される総価値の選択肢は 2 つあることは述べた。まず、①1 つ前のステージまでの組合せで、容量 j のナップサックへ入れる方法については、真上の要素を参照すればよい。ステージが無い状態で容量 j のナップサックへ入れる際の総価値は 0 であるため、①の値は 0 となる。次に、②1 つ前のステージまでの組合せに、現在のステージを加えて容量 j のナップサックへ入れる方法については、現在の容量である j から現在の Optional1 のプロセッサ使用率を引いた分の $j-U_1$ を容量とする要素を参照すればよい。即ち、容量 $j-U_1$ のナ

ップサックに入れる組合せに Optional1 を加えることで容量 j のナップサックへ入れることができる。表によると、ステージが無い状態で容量 $j-U_1$ のナップサックへ入れる際の総価値は 0 であり、それに Optional1 の価値を加えればよいことになる。従って、②の値は V_1 となる。最終的に、総価値が最大のものを表へ記入するため、2 行目の Optional1 でナップサックの容量が j の時の要素は V_1 となる。以上の値の選択肢を表へ図示したものが図 4.2.2.5 である。

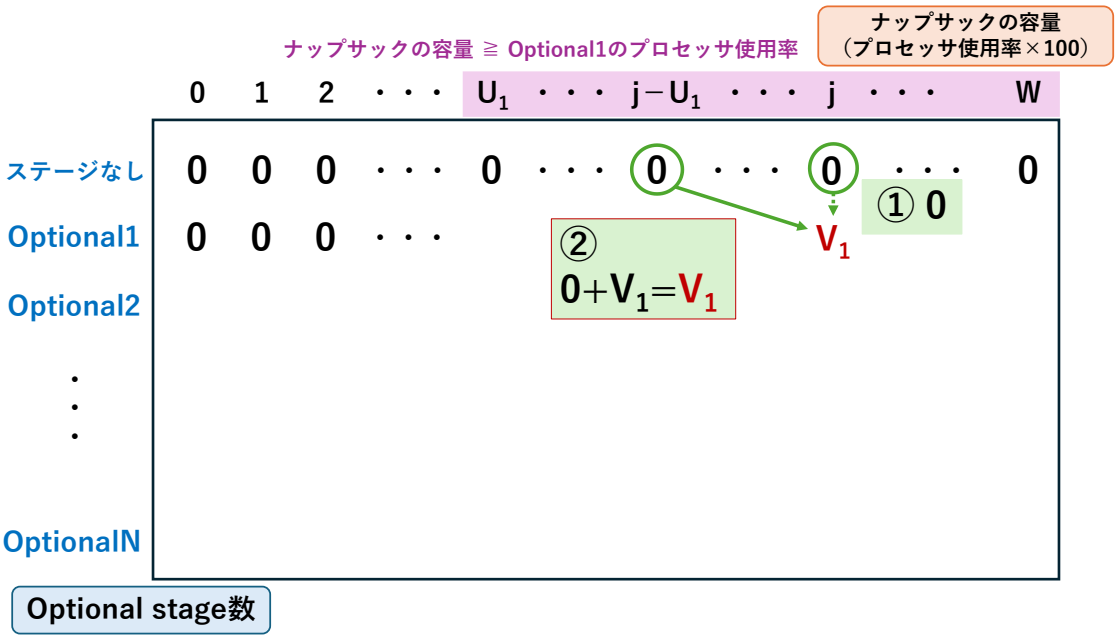

図 4.2.2.5：各選択肢の値を表中へ図示
（提案手法・動的計画法・1 番目のステージ）

● 2 番目以降のステージの計算

2 番目以降のステージについても、1 番目のステージと同様の計算を行う。しかし、2 番目以降のステージは前後の順序関係が破綻しないように選択することが必要である。初めに、現在のステージのプロセッサ使用率に対してナップサックの容量が小さい場合について考える。この場合、現在のステージはナップサックへ入れることができないため、1 つ前のステージまででナップサックへ入れた時の最大の総価値（真上の要素）を現在のステージの要素へ記入する。例として、2 番目以降のステージである Optional2 を考えているとする。このステージのプロセッサ使用率は U_2（$U_1 < U_2$）、価値（獲得精度）は V_2（$V_1 < V_2$）とする。Optional2 のプロセッサ使用率に対してナップサックの容量が小さい場合は、図 4.2.2.6 の横軸を紫色で塗りつぶした部分となる。これに対応する 3 行目の要素には真上の 2 行目の要素が記入される。

図 4.2.2.6：現在のステージ（2 番目以降のステージ）のプロセッサ使用率に対してナップサックの容量が小さい時

次に、ナップサックの容量が現在のステージのプロセッサ使用率以上の場合を考える。この場合、現在のステージはナップサックへ入れることができるため、ナップサックへの入れ方として考えられるのは前述の 2 つの選択肢である。

①　1 つ前のステージまでの組合せで、容量 j のナップサックへ入れる方法 ②　1 つ前のステージまでの組合せに、現在のステージを加えて容量 j のナップサックへ入れる方法

以上の 2 つの選択肢のうち、総価値が高い組合せが採用され、その時の総価値を要素へ記入する。先の例として、2 番目以降のステージである Optional2 を考えており、このステージのプロセッサ使用率は U_2（$U_1 < U_2$）、価値（獲得精度）は V_2（$V_1 < V_2$）とする。ナップサックの容量が Optional2 のプロセッサ使用率以上の場合は、図 4.2.2.7 の横軸を紫で塗りつぶした部分となる。

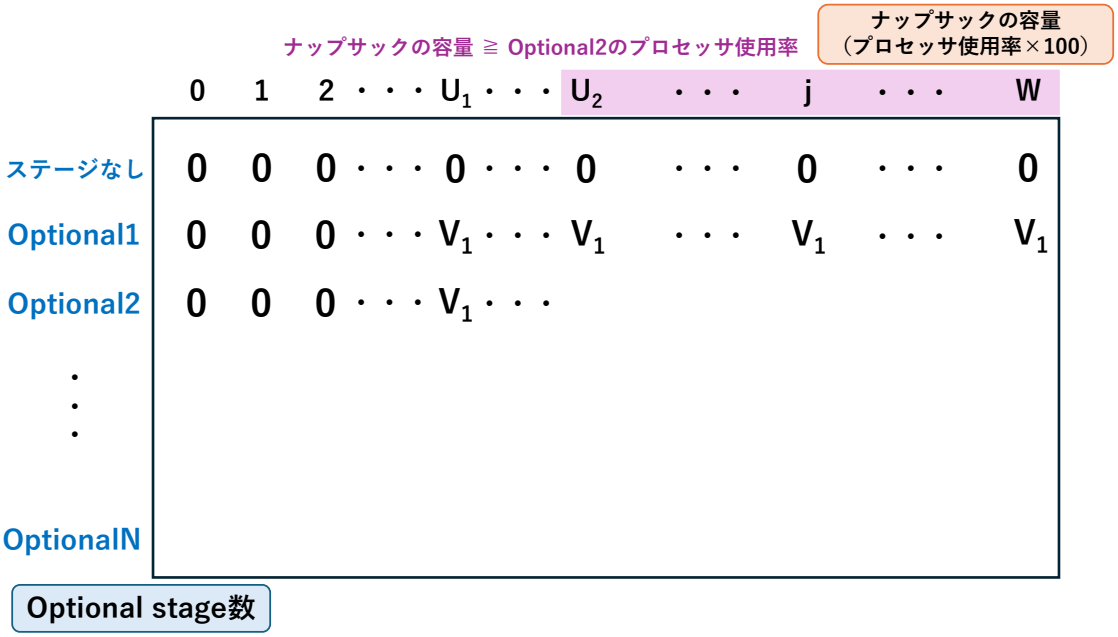

図 4.2.2.7：ナップサックの容量が現在のステージ（2 番目以降のステージ）のプロセッサ使用率以上の時

3 行目の Optional2 でナップサックの容量が j の時の要素を記入することを考える。この要素に記入される総価値の選択肢は 2 つあることは述べた。まず、①1 つ前のステージまでの組合せで、容量 j のナップサックへ入れる方法については、真上の要素を参照すればよい。次に、②1 つ前のステージまでの組合せに、現在のステージを加えて容量 j のナップサックへ入れる方法については、現在の容量である j から現在の Optional2 のプロセッサ使用率を引いた分の $j-U_2$ を容量とする要素を参照すればよい。

ここで、容量 $j-U_2$ の要素について注意することがある。Optional1 時点でナ

ップサックの容量が 0～U_1-1 の時は、要素は 0 であり Optional1 が選択されていないことが分かる（図 4.2.2.8 の黄色部分）。一方、Optional1 時点でナップサックの容量が U_1 以降の時は、要素が V_1 であり Optional1 が選択されていることが分かる（図 4.2.2.8 の緑色部分）。1 つ前のステージまでの組合せに、現在のステージを加えて容量 j のナップサックへ入れる方法で、図 4.2.2.8 の黄色部分に Optional2 を加えてしまうと、Optional1 は選択されずに Optional2 が選択されてしまう。Optional1 はタスクの 1 番目のステージであり、Optional2 はタスクの 2 番目以降のステージのため、前後の順序関係が破綻する。

図 4.2.2.8：Optional1 が非選択部分（黄色）と選択部分（緑色）

①1 つ前のステージまでの組合せで、容量 j のナップサックへ入れる方法と、②1 つ前のステージまでの組合せに、現在のステージを加えて容量 j のナップサックへ入れる方法のうち、大きい値を選択する。ここで、前後のステージの順序関係を保持するために、②の値が大きい際は、1 つ前のステージが選択済みかを調べる。1 つ前のステージが選択済みの場合、1 つ前のステージまでの総価値に現在のステージの価値を加えた値を記入する。1 つ前のステージが選択されていない場合、①の値を記入するようにする。以上のアルゴリズムを提案した。

先の例では 2 番目以降のステージである Optional2 を考えており、このステージのプロセッサ使用率は U_2（$U_1 < U_2$）、価値（獲得精度）は V_2（$V_1 < V_2$）とする。図 4.2.2.8 で、現在の容量 j から Optional2 のプロセッサ使用率 U_2 を

引いた時の容量 j－U_2 が緑色の部分にあったとする。要素へ記入する値としては、①Optional1 で容量 j の時の値、②Optional1 で容量 j－U_2 に Optional2 の価値を加えた時の値の 2 つの選択肢がある。①は V_1 であり、②は V_1+V_2 であることから、②の V_1+V_2 が記入されることになる（図 4.2.2.9）。

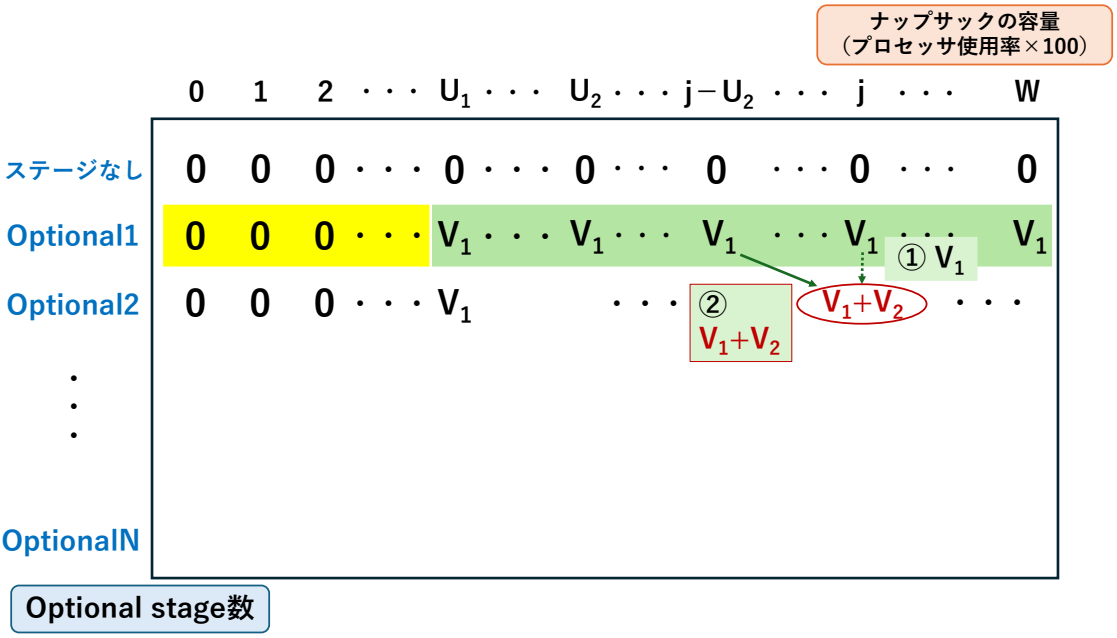

図 4.2.2.9：各選択肢の値を表中へ図示
（提案手法・動的計画法・2 番目以降のステージ）

次に、図 4.2.2.8 で、現在の容量 j から Optional2 のプロセッサ使用率 U_2 を引いた時の容量 j－U_2 が黄色の部分にあったとする。要素へ記入する値としては、①Optional1 で容量 j の時の値、②Optional1 で容量 j－U_2 に Optional2 の価値を加えた時の値の 2 つの選択肢がある。①は V_1、②は V_2 であり、②の値が大きい。しかし②とは、Optional1 は選択されずに Optional2 のみが選択されている状態である。Optional1 はタスクの 1 番目のステージで Optional2 はタスクの 2 番目以降のステージのため、②を選択してしまうと順序関係が破綻する。そこで順序関係を保持するために、①の V_1 の値が要素に記入されることになる（図 4.2.2.10）。

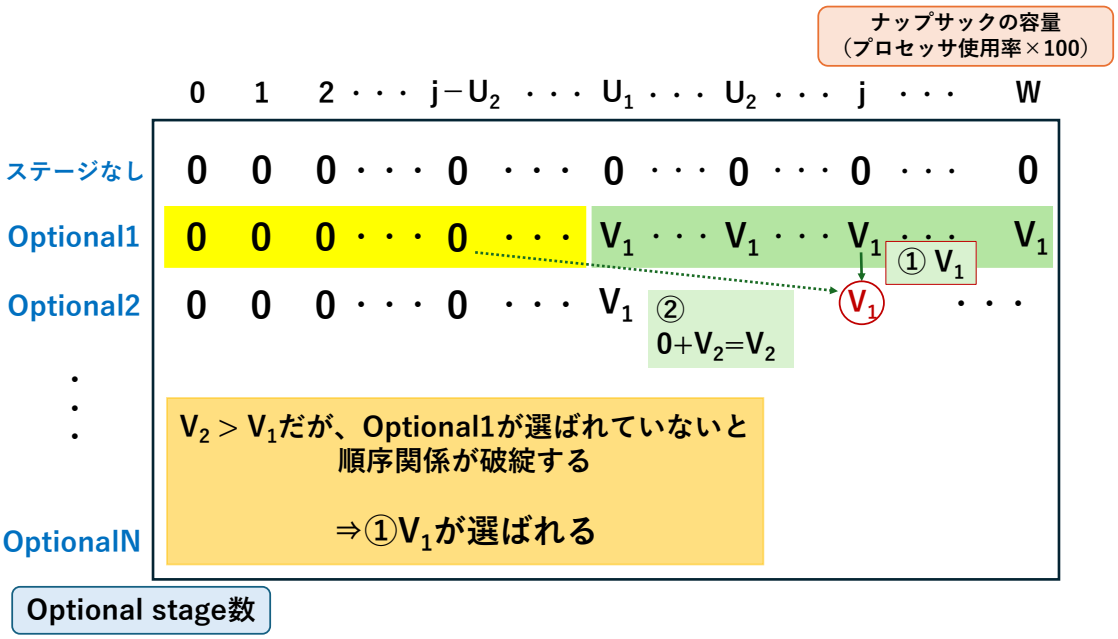

図 4.2.2.10：順序関係の保持のために現在のステージが選択されない状態

動的計画法アルゴリズムでは、表には総価値が記入されるため、表の計算が進行していくと 1 つ前のステージが選択されたかどうかが判断しにくい。そこで、設計したプログラムでは、表の各要素で各ステージについて選択済みか非選択かを{1, 0}で示すフラグを導入している（図 4.2.2.11）。1 つ前のステージを参照する際は、1 つ前のステージのフラグを調べ、それが 1 の場合は選択済みであることから、現在のステージを加えることができ、0 の場合は非選択であることから、現在のステージは加えないと判断することができる。

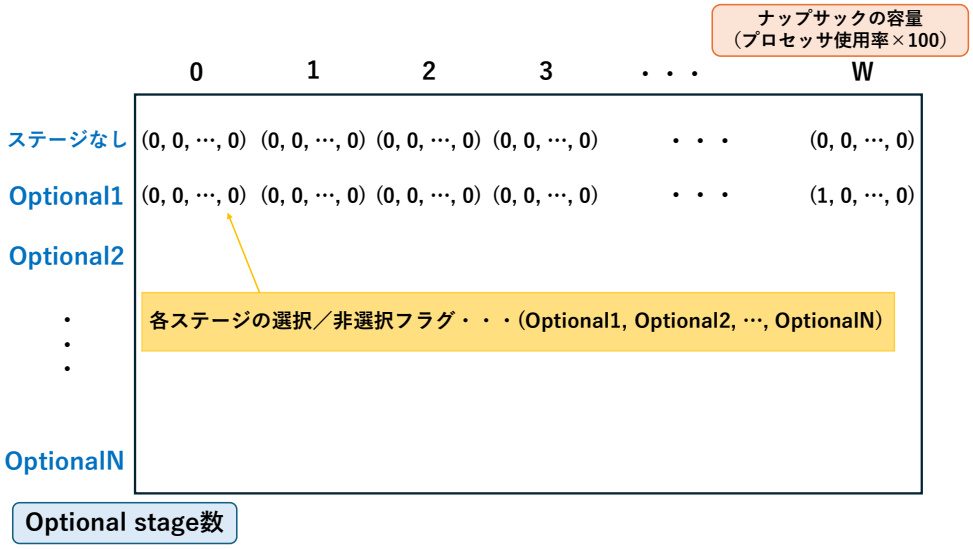

図 4.2.2.11：現在ステージ選択決定のための各ステージの選択／非選択フラグ

● Optional stage の組合せの算出

上記のように全 Optional stage について合計の総価値を表へ記入していく。表の全要素の記入が終了すれば、最終行（N 番目の Optional stage の行）を走査し、ナップサックの容量 W（(プロセッサ数) − (全 Mandatory stage のプロセッサ使用率)）の時の要素を参照する。この要素には、Optional1～OptionalN までの全 Optional stage の中で、容量 W のナップサックへ入れる組合せのうち、最大の総価値が記入されている。前節において、表の各要素では各 Optional stage について選択済みか非選択かを示すフラグを導入していると述べた。そのため、N 番目の Optional stage の行で容量 W の時のフラグを参照すれば、最終的な Optional stage の組合せを算出することができる（図 4.2.2.12）。

以上の動的計画法アルゴリズムによって、Optional stage の組合せが算出される。得られた組合せについて、プロセッサ使用率の合計は”(プロセッサ数) − (全 Mandatory stage のプロセッサ使用率)”以下となっており、既存の Mandatory stage と Optional stage 選択部分で得られた Optional stage 群を合わせて P-Fair スケジューリングすることが可能である。

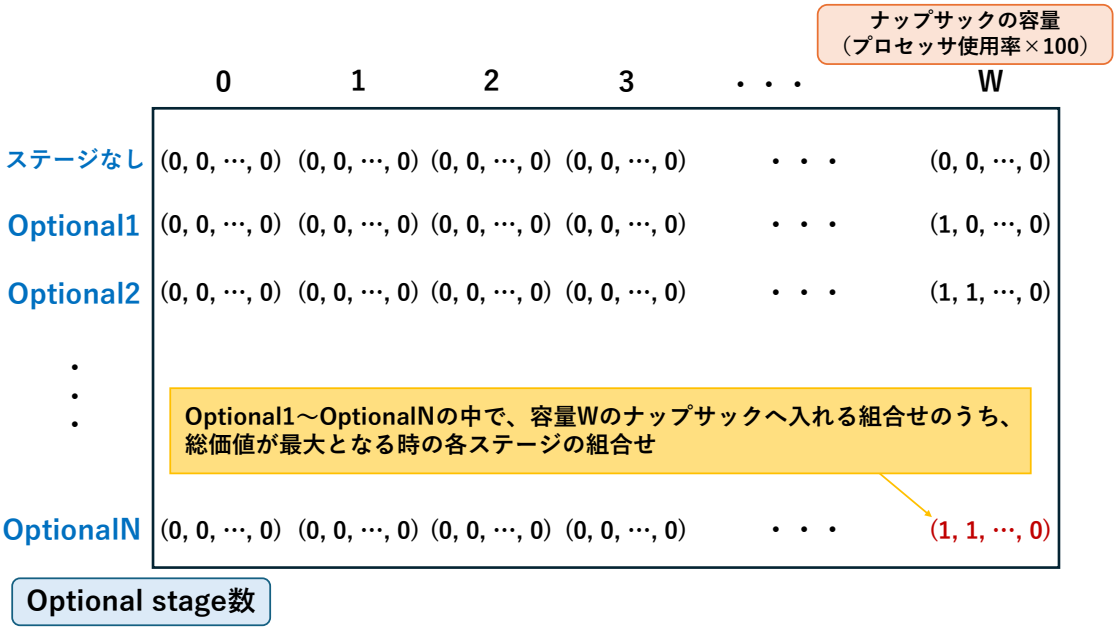

図 4.2.2.12：動的計画法アルゴリズムにおける Optional stage の組合せの算出

第5章 比較手法

5.1 全体のアルゴリズム

第 4 章の構成で提案手法をスケジューリングシミュレータとして設計する。しかし、マルチプロセッサ環境上で深層学習タスク群をリアルタイムスケジューリングする手法は先行研究が存在しないため、提案手法を比較評価することができない。そこで、第 3 章で解説したシングルプロセッサ環境かつ非周期タスク版の先行研究[2]を、マルチプロセッサ環境かつ周期タスク版へ拡張することにより、性能評価用の比較手法を開発する。

先行研究[2]のアルゴリズムは、図 3.1.1 に示す通りである。初めに Mandatory stage をデッドラインが短いタスク順にスケジューリングし、実行する。次に、Optional stage 選択部分で動的計画法アルゴリズムにより、最適な Optional stage を選択する。最後に、選択した Optional stage をデッドラインが短いタスク順にスケジューリングし、実行する。

比較手法のアルゴリズムを図 5.1.1 に示す。比較手法では、初期のタスクセットから各プロセッサへタスクのサブセットを割り当てる。各プロセッサでは、割り当てられたタスクのサブセットをシングルプロセッサ環境として Optional stage の選択と EDF スケジューリングを行う。先行研究[2]では、最初に Mandatory stage のスケジューリングと実行を行い、Mandatory stage の実行終了後に Optional stage の選択、最後に Optional stage のスケジューリングと実行を行っていた。しかし、比較手法では、周期タスクとしてスケジューリングを行うために、最初に Optional stage の選択を行い、次に Mandatory stage と選択後の Optional stage をまとめて周期タスクとして EDF スケジューリングを行うように拡張した。

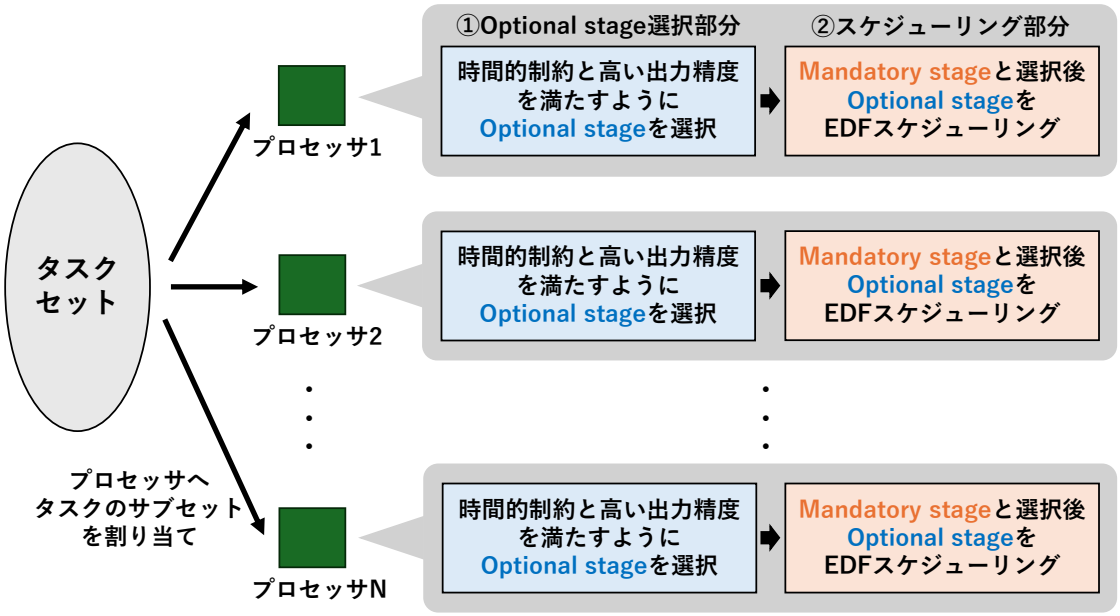

図 5.1.1：比較手法のアルゴリズム

5.2 タスクのサブセットの割り当て

初期のタスクセットから、タスクのサブセットを各プロセッサへ割り当てる際、各プロセッサでは最大のプロセッサ使用率は 100%以下となるように割り当てる必要がある。特に比較手法では、Mandatory stage は必ず実行しなければいけないステージのため、割り当て後にサブセットの Mandatory stage のプロセッサ使用率の合計が 100%以下となるようにサブセットを決める必要がある。しかし、全てのプロセッサで Mandatory stage のプロセッサ使用率の合計が 100%以下となるような、サブセットの組合せを決めることはビンパッキング問題と呼ばれる NP 困難問題に帰着する。

そこで、比較手法のタスクのサブセットの割り当てには、ビンパッキング問題の近似解のアルゴリズムを応用することにより、サブセットの最適な組合せを決定できるようにした。一般的にビンパッキング問題とは与えられたアイテムを容量の決まったビンに詰める時、ビンの数が最小となるようなアイテムの組合せは何かという問題である。それに対してビンの数が最小となるような組合せの近似解を求める代表的なアルゴリズムの一つに、FFD（First Fit Decreasing）アルゴリズムと呼ばれるものがある。FFD アルゴリズムは以下の通りである。

FFD アルゴリズム

①	空のビンを用意する。
②	アイテムをサイズの大きい順にソートする。
③	ソートした各アイテムについて、アイテムが収まるビンを（添え字が小さい順に）探す。
③-①	ビンが見つかれば、アイテムをそのビンへ入れる。
③-②	ビンが見つからなければ、新しいビンを開けて入れる。

比較手法のサブセット割り当て問題とは、タスクを容量の決まったビン（プロセッサ）へ与える時、ある数以下のビンへ割り当てるタスクの組合せは何かという問題である。この問題に対して FFD アルゴリズムを拡張したものを以下に示す。

FFD アルゴリズム（比較手法のサブセット割り当て問題へ拡張）

①	決まった数のビン（プロセッサ）をあらかじめ用意する。
②	初期タスクセットを Mandatory stage のプロセッサ総使用率の高い順にソートする。
③	ビンの容量を 1%に設定する。
④	ソートしたタスクを順に添え字の小さいビンから入れていく。
④-①	ビン全てにタスクが入れば終了する。
④-②	全タスクがビン全てに入りきらない場合は、容量を+1%して④を繰り返す。
⑤	タスクが全て入ったが、空のビンが余る場合、複数のタスクが入っているビンから１つのタスクを空のビンへ移動する。

上記のアルゴリズムは、ビンの容量を徐々に大きくしながら FFD アルゴリズムを適用することで、決まった数のビンの時の最適なタスクの組合せを見つけることができる。しかし、空のビンが余る時がある。例として、4 つのタスクを持つ初期タスクセットと 4 つのビンあるとする。そのうちの 1 つのタスク A の Mandatory stage のプロセッサ総使用率が 37.5%であるとする。上記のアルゴリズムの通り、容量 1%からビンの容量を徐々に大きくしていき、容量が 37%の時は、タスク A は詰め込むことはできない。そこで容量を 38%にした時、タスク A を含めて全てのタスクを詰め込めたが、1 つの空のビンが発生する（図 5.2.1）。

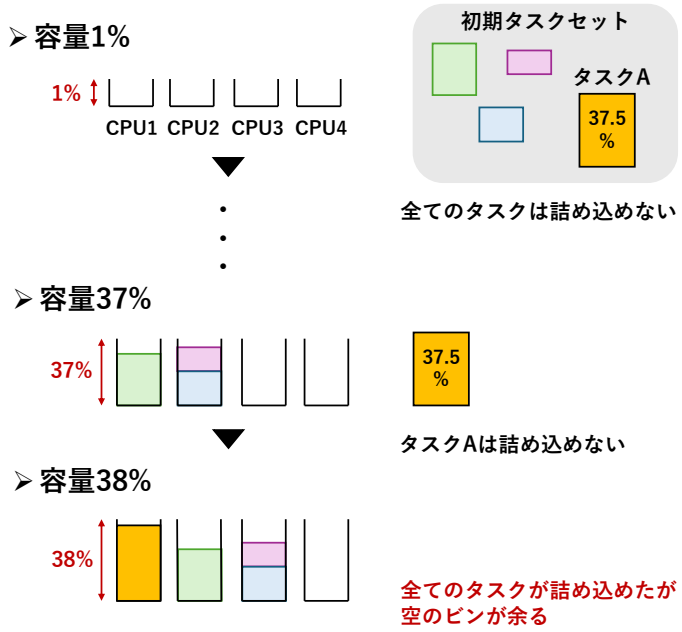

図 5.2.1：サブセット割り当ての FFD アルゴリズムで空のビンが発生する現象

比較手法では、全体的な性能をなるべく引き上げるため、全てのプロセッサを使用することを想定している。そのため、使用していないプロセッサ（空のビン）が発生しないように改良を行った。それは、空のビンが発生した場合、複数のタスクが入っているビンからタスクを 1 つ空のビンへ移動させるという処理である。

比較手法の FFD アルゴリズムでは、各ビン（プロセッサ）でいずれのタスクが割り当てられているかを示すフラグを導入している。例として、図 5.2.2 のように、Task1～Task4 の 4 つのタスクを持つタスクセットと 4 つのビンがあるとする。フラグは(Task1, Task2, Task3, Task4)となり、各ビンに該当するタス

クが入っている場合は 1、入っていない場合は 0 となる。FFD アルゴリズムでは添え字が小さいビンからタスクを入れていくため、空のビンは添え字の最も大きい右のビンで発生する。各ビンを占めるプロセッサ使用率を順に調べていき、プロセッサ使用率が 0 のビン（空のビン）が発見された場合、それ以前のビンでタスクが 2 個以上入っているビンを探す。タスクが 2 個以上入っているビンを発見すれば、タスクの添え字が小さいタスクから（Task1 から）フラグが 1 のタスクが入っているかを調べていく。フラグが 1 のタスクが入っていれば、そのタスクの Mandatory stage のプロセッサ総使用率をビンから引き、フラグを 0 にする。そして、空のビンにそのタスクの Mandatory stage のプロセッサ使用率を加算し、フラグを 1 にする。一連の処理により、複数のタスクが入っているビンから、空のビンへタスクを移動させる。以上から、全てのプロセッサは 1 つ以上のタスクが割り当てられ、プロセッサの使用個数についての提案手法と比較手法の不一致の問題は回避できる。

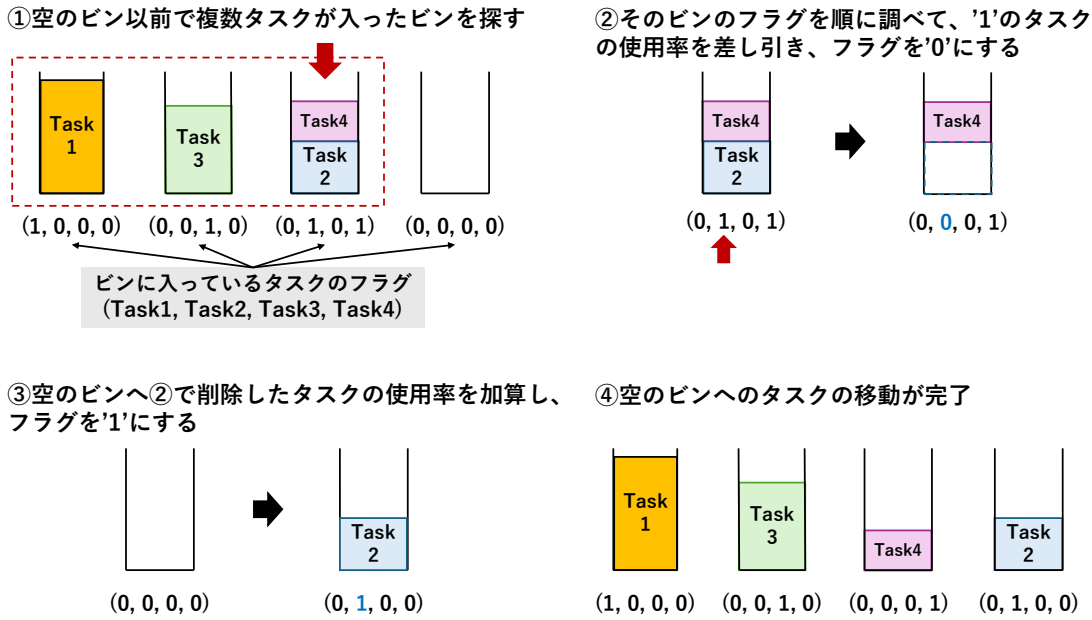

図 5.2.2：空のビンへタスクを移動させる処理

5.3 Optional stage 選択部分の拡張

比較手法では、Mandatory stage と選択後の Optional stage をまとめて EDF スケジューリングする。そのため Optional stage 選択部分では、選択された Optional stage のプロセッサ使用率の合計が”1－(全 Mandatory stage のプロセッサ使用率の合計)”以下となるように選択する必要がある。Optional stage 選択部分のアルゴリズムは 3.2 節で述べた通りであるが、比較手法では選択の過程でプロセッサ使用率の合計を計算するように拡張した。

例として、図 3.2.2 のタスク（Task1）を使用する。このタスクは Optional1、Optional2、Optional3 の 3 つの Optional stage を持っており、実行時間はそれぞれ 1、3、2 であり、出力精度はそれぞれ a%、b%、c%とする。更に、プロセッサ使用率をそれぞれ l%、m%、n%とする（図 5.3.1)。

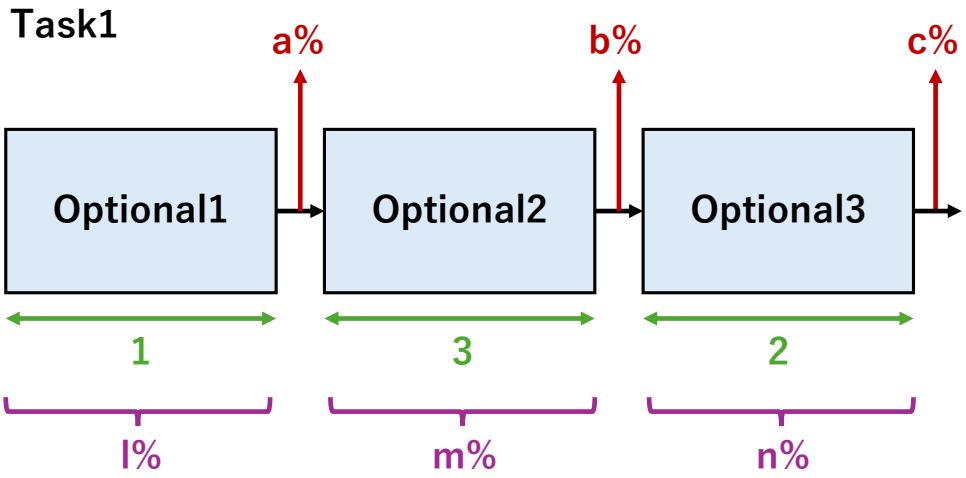

図 5.3.1：図 3.2.2 のタスク（Task1）の例
（プロセッサ使用率はそれぞれ l%、m%、n%（紫色））

Task1 の Optional stage で合計精度を獲得する時の最小実行時間は、図 3.2.3 の通りである。比較手法では最小実行時間の表に加えて、プロセッサ使用率の合計の表を追加で作成する。図 3.2.3 では最小実行時間は、合計精度が a%以下の時は Optional1 の 1、合計精度が b%以下の時は Optional1 と Optional2 の 4、合計精度が c%以下の時は Optional1 と Optional2 と Optional3 の 6、合計精度が c%を超える時は獲得できないため×としている。プロセッサ使用率の合計も

最小実行時間に合わせて作成すると、Task1 の Optional stage のみの場合では図 5.3.2 の通りである。

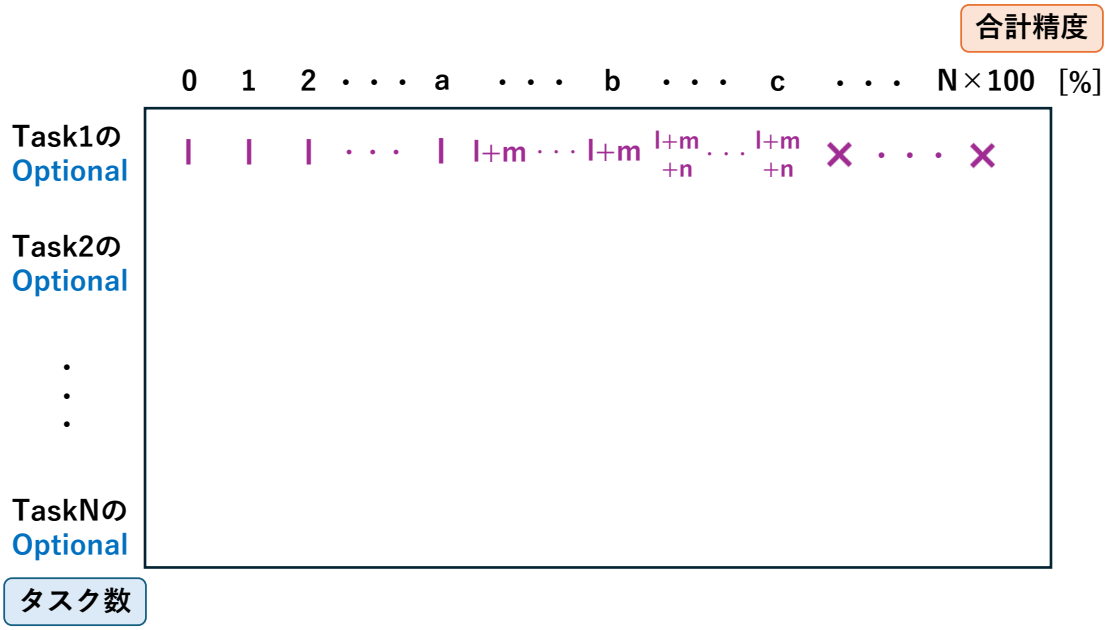

図 5.3.2：Task1 の Optional stage のみで合計精度を獲得できる時のプロセッサ使用率の合計

2 番目のタスク以降も同様にプロセッサ使用率を計算していく。2 番目のタスク（Task2）は図 3.2.4 を例として使用する。このタスクは Optional4 と Optional5 の 2 つの Optional stage を持っており、実行時間はそれぞれ 1、3 であり、出力精度はそれぞれ d%、a+d%である。更にプロセッサ使用率はそれぞれ p%、q%とする（図 5.3.3）。

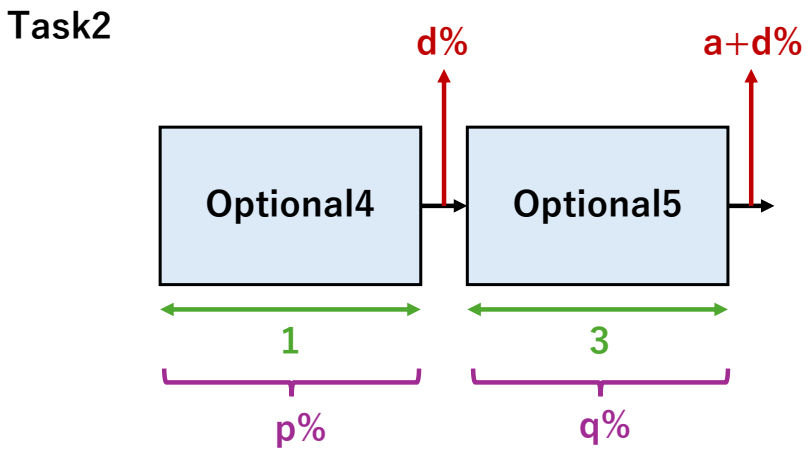

図 5.3.3：図 3.2.4 のタスク（Task2）の例
（プロセッサ使用率はそれぞれ p%、q%（紫色））

Task2 までで合計精度 a+d%を獲得する時の最小実行時間は、3.2 節の動的計画法アルゴリズムによると、Task1 までの Optional stage に Task2 の Optional4 を加えた時の実行時間（2）である（図 3.2.5）。プロセッサ使用率の合計も最小実行時間の計算に合わせて更新していく。従って、Task1 までの Optional stage で合計精度 a%を獲得できる時のプロセッサ使用率の合計に、Task2 の Optional4 のプロセッサ使用率を加算した値が記入される（図 5.3.4）。

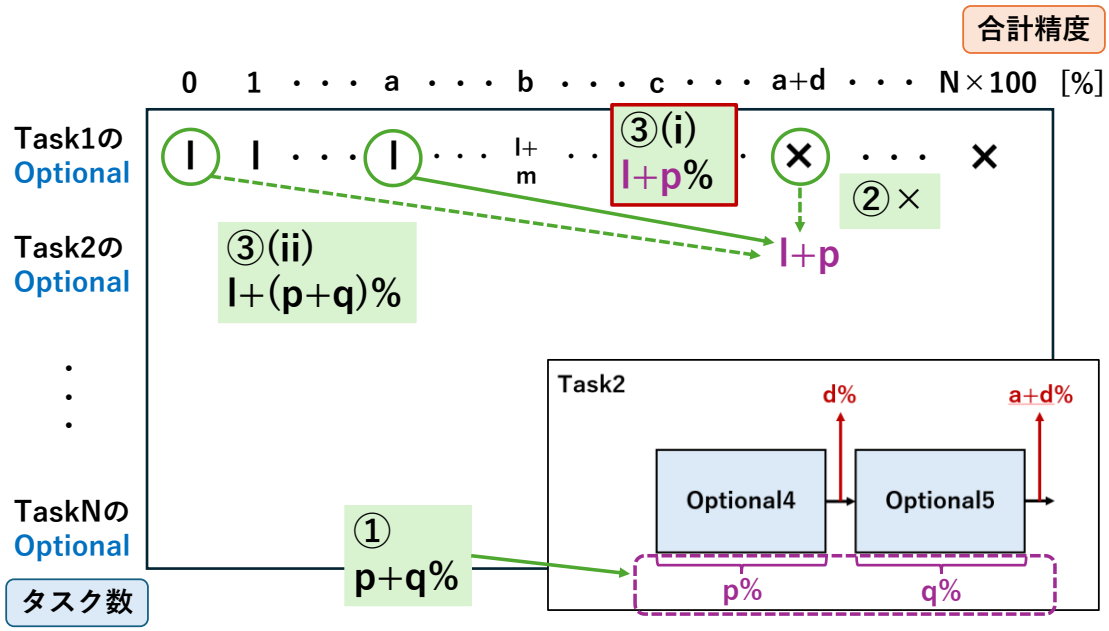

図 5.3.4：図 3.2.5 の選択肢に合わせて計算したプロセッサ使用率

動的計画法アルゴリズムに従って最小実行時間を計算し、それと並行してプロセッサ使用率の合計も計算する。先行研究[2]では表の全ての行を記入すると、最終行（N 番目のタスクの行）で最も高い合計精度を得られる時の組合せを調べるが、比較手法ではプロセッサ使用率の合計が”1－(全 Mandatory stage のプロセッサ使用率の合計)”以下で最も高い合計精度を得られる時の組合せを調べている。プロセッサ使用率の合計という制約を加えて選択したことで Optional stage の組合せは既存の Mandatory stage と組み合わせて EDF スケジューリングを行うことができ、尚且つ最も高い合計精度を獲得することができる。

以上から比較手法は、シングルプロセッサ環境かつ非周期タスク版の先行研究[2]を、マルチプロセッサ環境かつ周期タスク版へ拡張することで開発した。比較手法では主に、初期タスクセットのサブセットを各プロセッサへ割り当てる際に、ビンパッキング問題の FFD アルゴリズムを応用している点、先行研究[2]の Optional stage 選択部分でプロセッサ使用率の合計を計算している点の二点において拡張を行っている。

第6章 評価実験

6.1 タスクセット生成部

マルチプロセッサ環境を対象とした深層学習タスク群のリアルタイムスケジューリングアルゴリズムの開発を行った。本研究における評価実験では、仮想的に深層学習タスク群を生成し、それらを基にスケジューリングのシミュレーションを行う。この評価実験を通して、開発したアルゴリズムの有用性や問題点を明らかにすることが評価実験の目的である。従って、エッジサーバ上にスケジューリングプログラムやニューラルネットワークを実装し、実際の組込み機器と通信しながら推論を実行するのではない。評価実験は機能確認と性能評価の二段階で構成される。機能確認とは、開発したシミュレータが正常動作することを確認することである。

開発したアルゴリズムの評価実験（機能確認・性能評価）のために、深層学習タスク群を仮想的に生成する必要があると述べた。そこで、タスクセット生成部を設計する。深層学習タスクが持つパラメータは以下の表 6.1.1 の通りであり、各パラメータを図示したものを図 6.1.1 に示す。

①	タスク番号
②	全ステージ数（Mandatory stage＋Optional stage）
③	タスクの総実行時間
④	初期リリース
⑤	相対デッドライン
⑥	周期
⑦	Mandatory stage 数
⑧	Optional stage 数
⑨	Mandatory stage 総実行時間
⑩	Optional stage 総実行時間
⑪	Mandatory stage 各実行時間
⑫	Optional stage 各実行時間
⑬	Mandatory stage の最終出力精度
⑭	Optional stage の各獲得精度
⑮	Optional stage の各出力精度

表 6.1.1：深層学習タスクが持つパラメータ

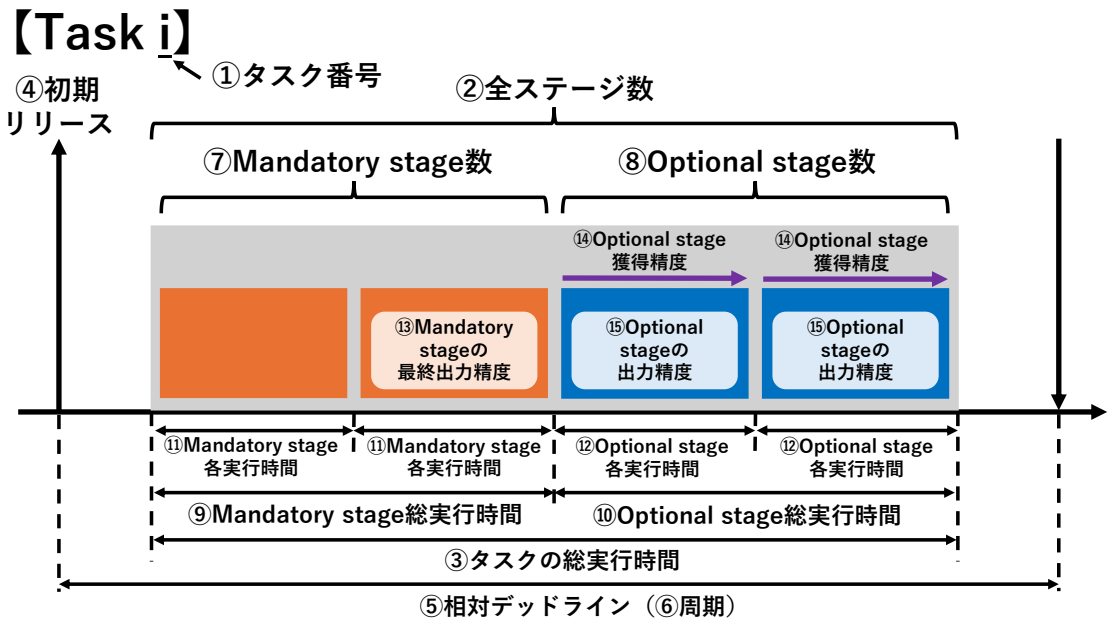

図 6.1.1：深層学習タスクが持つパラメータ

深層学習タスクのパラメータは乱数の種をもとに計算される。タスクセット生成関数の taskset_generate 関数の概要は以下のコード 6.1.1 の通りであり、

```
taskset_generate()
{
    • 乱数の種を指定

    • 擬似乱数をもとに全タスク数Nを決定

    for(i = 0 to N-1)
    {
      • 擬似乱数をもとにパラメータを決定

    }
}
```

コード 6.1.1：タスクセット生成関数（taskset_generate 関数）の擬似コード

タスクセット生成部は C 言語で設計している。乱数の種をもとにした擬似乱数列の計算には標準ライブラリ（stdlib.h）の rand 関数が用いられ、擬似乱数列生成のための乱数の種を指定するためには標準ライブラリ（stdlib.h）の srand 関数が用いられる。srand 関数の引数へ乱数の種を与え、rand 関数を実行することにより、乱数の種をもとに計算された乱数を取得できる。全タスク数 N やパラメータの決定のために、指定した範囲内で乱数を取得する get_random 関数を設計した。関数の概要は以下のコード 6.1.2 の通りである。

```
get_random(min, max)
{

 x = min + (rand() * (max – min + 1.0) / (1.0 + RAND_MAX))


 return x

}
```

コード 6.1.2：指定した範囲内で乱数を取得する関数（get_random 関数）の擬似コード

get_random 関数では、2 つの引数の min と max を与えて min～max の間の値を無作為に取得している。値の取得の計算には rand 関数が用いられており、0～RAND_MAX の範囲で乱数を取得する。全タスク数の決定には get_random 関数が用いられ、例として、3～10 個の間から全タスク数を決定するためには get_random(3, 10)としてその返り値を取得すればよい。

以上のタスクセット生成関数（taskset_generate 関数）と乱数取得の関数（get_random 関数）から、各パラメータは以下の表 6.1.2 ように決定される。

タスクが持つパラメータ		メンバ名（変数名）	計算式	
①	タスク番号	ori_num	taskset_generate 関数内の for 文のインデックス i	
②	全ステージ数	stage_num	get_random(MIN_STAGE_NUM, MAX_STAGE_NUM)	
③	タスクの総実行時間	exe_time	get_random(stage_num, stage_num + EXETIME_MAX_DIV)	
④	初期リリース	release	0	
⑤	相対デッドライン	r_deadline	get_random(exe_time + DEADLINE_MIN_DIV, exe_time + DEADLINE_MAX_DIV)	
⑥	周期	period	r_deadline	
⑦	Mandatory stage 数	mStage_num	get_random(MANDATORY_NUM_MIN, stage_num – MANDATORY_NUM_MAX_DIV)	
⑧	Optional stage 数	oStage_num	stage_num – mStage_num	
⑨	Mandatory stage 総実行時間	mStage_etime	get_random(mStage_num, exe_time – oStage_num)	
⑩	Optional stage 総実行時間	oStage_etime	exe_time – mStage_etime	
⑪	Mandatory stage 各実行時間	mEtime_per[j] (j = 0 ~ mStage_num − 1)	(mStage_etime + j) / mStage_num	
⑫	Optional stage 各実行時間	oEtime_per[j] (j = 0 ~ oStage_num − 1)	(oStage_etime + j) / oStage_num	
⑬	Mandatory stage 最終出力精度	mOut	get_random(MANDATORY_OUT_MIN, MANDATORY_OUT_MAX) / 100.0	
⑭	Optional stage 各獲得精度	getAc[j] (j = 0 ~ oStage_num − 1)	j = 0	0.5 * (1 – mOut)
			j > 0	0.5 * (1 – oOut[j-1])
⑮	Optional stage 各出力精度	oOut[j] (j = 0 ~ oStage_num − 1)	j = 0	mOut + getAc[j]
			j > 0	oOut[j-1] + getAc[j]

表 6.1.2：深層学習タスクのパラメータ計算

パラメータ計算に使用しているマクロ定数の説明は以下の表 6.1.3 の通りであり、シミュレーション実行前に手動で初期設定する。

マクロ定数名	設定内容
MIN_STAGE_NUM	全ステージ数の下限
MAX_STAGE_NUM	全ステージ数の上限
EXETIME_MAX_DIV	総実行時間の上限について、全ステージ数からどれくらい増分するか
DEADLINE_MIN_DIV	相対デッドラインの下限について、総実行時間からどれくらい増分するか
DEADLINE_MAX_DIV	相対デッドラインの上限について、総実行時間からどれくらい増分するか
MANDATORY_NUM_MIN	Mandatory stage 数の下限
MANDATORY_NUM_MAX_DIV	Mandatory stage 数の上限について、全ステージ数からどれだけ減分するか
MANDATORY_OUT_MIN	Mandatory stage の最終出力精度の下限
MANDATORY_OUT_MAX	Mandatory stage の最終出力精度の上限

表 6.1.3：深層学習タスクのパラメータ計算に使用しているマクロ定数

以上から、タスクセット生成部の擬似コードは以下のコード 6.1.3 の通りである。乱数の種の指定に与えているマクロ定数は RAND_SEED、全タスク数 N の決定のために get_random 関数に与えているマクロ定数は MIN_TASKNUM と MAX_TASKNUM である。RAND_SEED は乱数の種の値、MIN_TASKNUM は全タスク数の下限、MAX_TASKNUM は全タスク数の上限を表す。そして、各タスクは C 言語で構造体によって表現され、パラメータは構造体のメンバとして格納される（図 6.1.2)。

```
get_random(min, max)
{
        x = min + (rand() * (max - min + 1.0) / (1.0 + RAND_MAX))
        return x
}

taskset_generate()
{
        srand(RAND_SEED)
        N = get_random(MIN_TASKNUM, MAX_TASKNUM)

        for(i = 0 to N - 1)
        {
                Task[i].ori_num = i
                Task[i].stage_num = get_random(MIN_STAGE_NUM, MAX_STAGE_NUM)
                Task[i].exe_time
                = get_random(Task[i].stage_num, Task[i].stage_num + EXETIME_MAX_DIV)
                                                        .
                                                        .
                                                        .
        }
}
```

最小値(min) ～ 最大値(max)の間で乱数取得

乱数の種の指定

全タスク数Nの決定

全タスクについてパラメータの決定

コード 6.1.3：タスクセット生成部の擬似コード

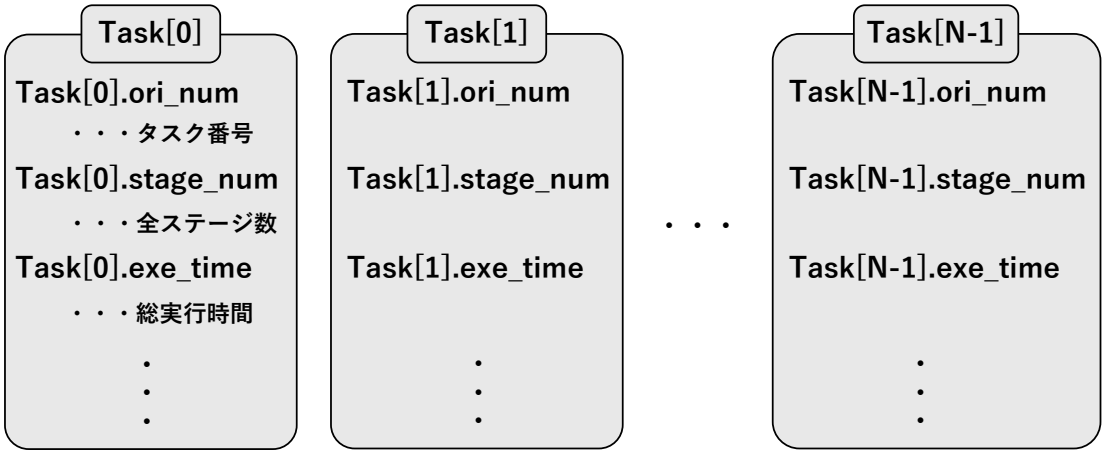

図 6.1.2：深層学習タスク（構造体）とパラメータ（メンバ）

6.2 実験環境

実験環境を図 6.2.1 に示す。タスクセット生成部とスケジューリングシミュレータは C 言語で開発する。開発と実験に使用した器材とソフトウェアを表 6.2.1 に示す。評価実験では、タスクセット生成部で生成された深層学習タスクセットを、提案手法と比較手法のスケジューリングシミュレータへ入力させる。機能確認では、それぞれの手法の Optional stage 選択部分とスケジューリング部分が正常に動作するかを確認する。性能評価では、Optional stage 選択部分で得られた Optional stage から、全体的な精度の優劣を評価する。また、Optional stage 選択部分のプログラムの実行時間の変化を評価する。

器材名	製品名（型番）	用途	備考
パーソナルコンピュータ	GALLERIA RL5C-G50	シミュレータの開発・実験	CPU：11th Gen Intel(R) Core(TM) i5-11400H（2.7GHz） OS：Windows11 Home（64ビット版） メモリ：16GB
統合開発環境	Microsoft Visual Studio Community 2022	シミュレータプログラムの設計・ デバッグ・実験	64ビット版 Version 17.11.1
シングルボードコンピュータ	Raspberry Pi 4B	Optional stage選択部分のCPU時間の計測	CPU：ARM Cortex-A72（1.5GHz） OS：Raspbian GNU/Linux 10 (buster)（32ビット版） メモリ：4GB

表 6.2.1：評価実験の使用器材

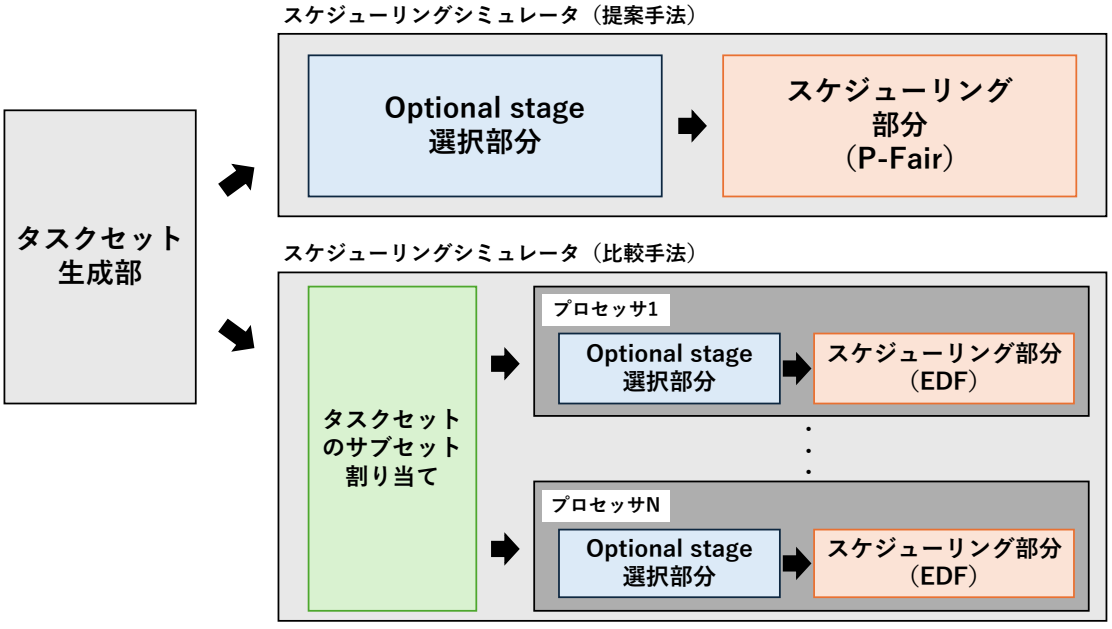

図 6.2.1：実験環境

6.3 機能確認

本節では、開発したスケジューリングシミュレータの Optional stage 選択部分とスケジューリング部分が正常に機能しているかを確認する。タスクセット生成部のマクロ定数は以下の表 6.3.1 のように設定し、表 6.1.2 をもとにパラメータが計算される。

スケジューリングシミュレータはマルチプロセッサ環境で実行することを想定しており、本実験ではプロセッサ数は 4 とする。深層学習タスクの実行時間は最悪実行時間として扱い、実行時間や周期、相対デッドラインは各周期で変動せず常に一定とする。扱うスケジューリングシミュレータはオフラインアルゴリズムとして機能し、タスクセット生成部で生成されたタスクセットは、実行を通して個数やパラメータは変動しないものとする。初期リリースの到来はすべて tick = 0 とし、周期と相対デッドラインは等しく、絶対デッドラインと同時刻に次のリリースが到来するものとする。深層学習タスクはプリエンプティブなタスクとし、ステージ間の境界にかかわらず、ステージの実行中に他のタスクへ実行を切り替えることも可能とする。

マクロ定数名	設定値
RAND_SEED	1
MIN_TASKNUM	12
MAX_TASKNUM	12
MIN_STAGE_NUM	3
MAX_STAGE_NUM	10
EXETIME_MAX_DIV	3
DEADLINE_MIN_DIV	3
DEADLINE_MAX_DIV	5
MANDATORY_NUM_MIN	1
MANDATORY_NUM_MAX_DIV	1
MANDATORY_OUT_MIN	70
MANDATORY_OUT_MAX	80

表 6.3.1：機能確認実験のタスクセット生成部のマクロ定数の設定

以上のマクロ定数の設定から、タスクセット生成部では以下の表 6.3.2 のように各パラメータが計算される。値 A～値 B の表示は、値 A～値 B の範囲から無作為に選択するという意味である。

タスクセット生成部のパラメータ	設定値	
全タスク数	12	
全ステージ数	3 ～ 10	
タスクの総実行時間	全ステージ数 ～ 全ステージ数＋3	
初期リリース	0	
相対デッドライン	タスクの総実行時間＋3 ～ タスクの総実行時間＋5	
周期	相対デッドライン	
Mandatory stage 数	1 ～ 全ステージ数－1	
Mandatory stage 総実行時間	Mandatory stage 数 ～ タスクの総実行時間－Optional stage 数	
Mandatory stage 各実行時間 （j = 0 ～ Mandatory stage 数－1）	(Mandatory stage 総実行時間＋j) / Mandatory stage 数	
Optional stage 各実行時間 （j = 0 ～ Optional stage 数－1）	(Optional stage 総実行時間＋j) / Optional stage 数	
Mandatory stage 最終出力精度	0.7 ～ 0.8	
Optional stage 各獲得精度 （j = 0 ～ Optional stage 数－1）	j = 0	0.5 * (1－(Mandatory stage 最終出力精度))
	j > 0	0.5 * (1－(j－1 番目の Optional stage の出力精度))
Optional stage 各出力精度 （j = 0 ～ Optional stage 数－1）	j = 0	(Mandatory stage 最終出力精度) ＋(j 番目の Optional stage の獲得精度)
	j > 0	(j－1 番目の Optional stage の出力精度) ＋(j 番目の Optional stage の獲得精度)

表 6.3.2：機能確認実験のタスクセットのパラメータ

タスクセット生成部で生成された 12 個の深層学習タスクの結果は図 6.3.1～図 6.3.3 に示される。図 6.3.1～図 6.3.3 より、正常に 12 個の深層学習タスクが生成されていることを確認した。

```
[ タスクセット生成部 ]
[Task1]
        ステージ数 ... 3
        総実行時間 ... 4
        初期リリース ... 0
        相対デッドライン ... 8
        Mandatory stage数 ... 1
        Optional stage数 ... 2
        Mandatory stage各実行時間 ... 1
        Optional stage各実行時間 ... 1 2
        Mandatory stage最終精度 ... 0.800000
        Optional stage各出力精度 ... 0.900000 0.950000
        Optional stage各獲得精度 ... 0.100000 0.050000

[Task2]
        ステージ数 ... 6
        総実行時間 ... 7
        初期リリース ... 0
        相対デッドライン ... 11
        Mandatory stage数 ... 2
        Optional stage数 ... 4
        Mandatory stage各実行時間 ... 1 1
        Optional stage各実行時間 ... 1 1 1 2
        Mandatory stage最終精度 ... 0.780000
        Optional stage各出力精度 ... 0.890000 0.945000 0.972500 0.986250
        Optional stage各獲得精度 ... 0.110000 0.055000 0.027500 0.013750

[Task3]
        ステージ数 ... 3
        総実行時間 ... 4
        初期リリース ... 0
        相対デッドライン ... 9
        Mandatory stage数 ... 1
        Optional stage数 ... 2
        Mandatory stage各実行時間 ... 2
        Optional stage各実行時間 ... 1 1
        Mandatory stage最終精度 ... 0.790000
        Optional stage各出力精度 ... 0.895000 0.947500
        Optional stage各獲得精度 ... 0.105000 0.052500

[Task4]
        ステージ数 ... 3
        総実行時間 ... 3
        初期リリース ... 0
        相対デッドライン ... 8
        Mandatory stage数 ... 2
        Optional stage数 ... 1
        Mandatory stage各実行時間 ... 1 1
        Optional stage各実行時間 ... 1
        Mandatory stage最終精度 ... 0.720000
        Optional stage各出力精度 ... 0.860000
        Optional stage各獲得精度 ... 0.140000

[Task5]
        ステージ数 ... 8
        総実行時間 ... 11
        初期リリース ... 0
        相対デッドライン ... 14
        Mandatory stage数 ... 4
        Optional stage数 ... 4
        Mandatory stage各実行時間 ... 1 2 2 2
        Optional stage各実行時間 ... 1 1 1 1
        Mandatory stage最終精度 ... 0.730000
        Optional stage各出力精度 ... 0.865000 0.932500 0.966250 0.983125
        Optional stage各獲得精度 ... 0.135000 0.067500 0.033750 0.016875
```

図 6.3.1：タスクセット生成部の結果（Task1～Task5）

```
[Task6]
        ステージ数 ... 6
        総実行時間 ... 8
        初期リリース ... 0
        相対デッドライン ... 12
        Mandatory stage数 ... 2
        Optional stage数 ... 4
        Mandatory stage各実行時間 ... 1 1
        Optional stage各実行時間 ... 1 1 2 2
        Mandatory stage最終精度 ... 0.770000
        Optional stage各出力精度 ... 0.885000 0.942500 0.971250 0.985625
        Optional stage各獲得精度 ... 0.115000 0.057500 0.028750 0.014375

[Task7]
        ステージ数 ... 5
        総実行時間 ... 5
        初期リリース ... 0
        相対デッドライン ... 10
        Mandatory stage数 ... 2
        Optional stage数 ... 3
        Mandatory stage各実行時間 ... 1 1
        Optional stage各実行時間 ... 1 1 1
        Mandatory stage最終精度 ... 0.740000
        Optional stage各出力精度 ... 0.870000 0.935000 0.967500
        Optional stage各獲得精度 ... 0.130000 0.065000 0.032500

[Task8]
        ステージ数 ... 10
        総実行時間 ... 11
        初期リリース ... 0
        相対デッドライン ... 16
        Mandatory stage数 ... 7
        Optional stage数 ... 3
        Mandatory stage各実行時間 ... 1 1 1 1 1 1 2
        Optional stage各実行時間 ... 1 1 1
        Mandatory stage最終精度 ... 0.740000
        Optional stage各出力精度 ... 0.870000 0.935000 0.967500
        Optional stage各獲得精度 ... 0.130000 0.065000 0.032500

[Task9]
        ステージ数 ... 9
        総実行時間 ... 11
        初期リリース ... 0
        相対デッドライン ... 15
        Mandatory stage数 ... 4
        Optional stage数 ... 5
        Mandatory stage各実行時間 ... 1 1 1 1
        Optional stage各実行時間 ... 1 1 1 2 2
        Mandatory stage最終精度 ... 0.720000
        Optional stage各出力精度 ... 0.860000 0.930000 0.965000 0.982500 0.991250
        Optional stage各獲得精度 ... 0.140000 0.070000 0.035000 0.017500 0.008750

[Task10]
        ステージ数 ... 8
        総実行時間 ... 9
        初期リリース ... 0
        相対デッドライン ... 12
        Mandatory stage数 ... 4
        Optional stage数 ... 4
        Mandatory stage各実行時間 ... 1 1 1 2
        Optional stage各実行時間 ... 1 1 1 1
        Mandatory stage最終精度 ... 0.750000
        Optional stage各出力精度 ... 0.875000 0.937500 0.968750 0.984375
        Optional stage各獲得精度 ... 0.125000 0.062500 0.031250 0.015625
```

図 6.3.2：タスクセット生成部の結果（Task6～Task10）

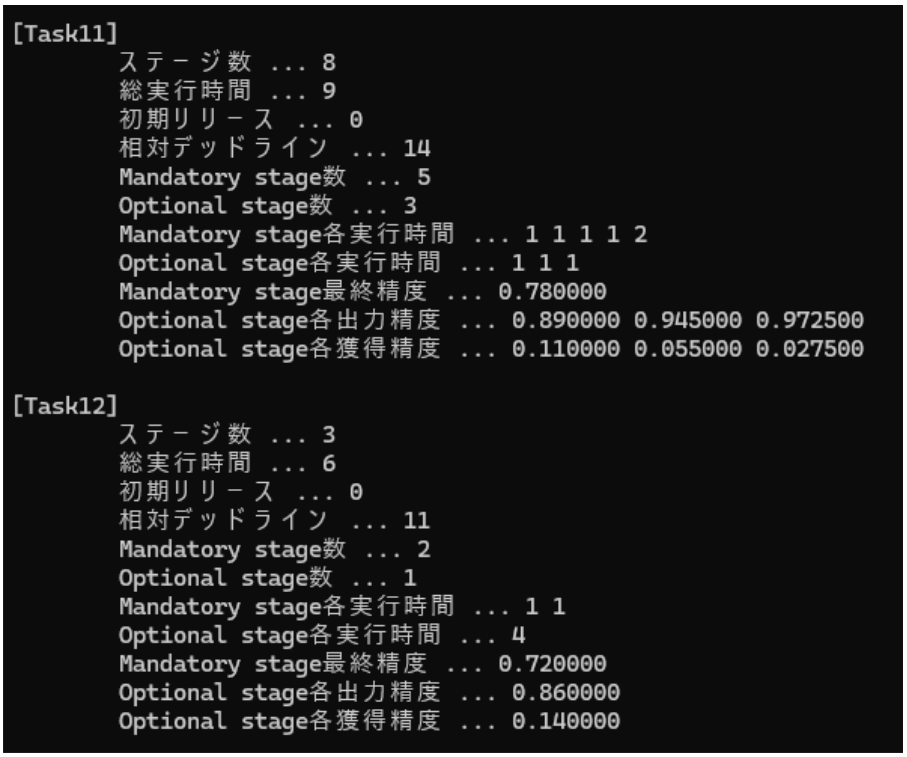

図 6.3.3：タスクセット生成部の結果（Task11～Task12）

6.3.1 提案手法の確認

6.3.1.1 近似解法（貪欲法）

提案手法のスケジューリングシミュレータの Optional stage 選択部分について、貪欲法アルゴリズムが正常に機能しているかを確認する。Optional stage 選択部分の結果を図 6.3.1.1.1 に示す。

```
[ Optional stage選択部分 ]
[Task1]
        ステージ数(選択後 / 選択前) ... 0 / 2
        Optional stage総使用率(選択後 / 選択前) ... 0.000000 / 0.375000
[Task2]
        ステージ数(選択後 / 選択前) ... 0 / 4
        Optional stage総使用率(選択後 / 選択前) ... 0.000000 / 0.454545
[Task3]
        ステージ数(選択後 / 選択前) ... 0 / 2
        Optional stage総使用率(選択後 / 選択前) ... 0.000000 / 0.222222
[Task4]
        ステージ数(選択後 / 選択前) ... 0 / 1
        Optional stage総使用率(選択後 / 選択前) ... 0.000000 / 0.125000
[Task5]
        ステージ数(選択後 / 選択前) ... 1 / 4
        Optional stage総使用率(選択後 / 選択前) ... 0.071429 / 0.285714
[Task6]
        ステージ数(選択後 / 選択前) ... 1 / 4
        Optional stage総使用率(選択後 / 選択前) ... 0.083333 / 0.500000
[Task7]
        ステージ数(選択後 / 選択前) ... 1 / 3
        Optional stage総使用率(選択後 / 選択前) ... 0.100000 / 0.300000
[Task8]
        ステージ数(選択後 / 選択前) ... 1 / 3
        Optional stage総使用率(選択後 / 選択前) ... 0.062500 / 0.187500
[Task9]
        ステージ数(選択後 / 選択前) ... 1 / 5
        Optional stage総使用率(選択後 / 選択前) ... 0.066667 / 0.466667
[Task10]
        ステージ数(選択後 / 選択前) ... 1 / 4
        Optional stage総使用率(選択後 / 選択前) ... 0.083333 / 0.333333
[Task11]
        ステージ数(選択後 / 選択前) ... 1 / 3
        Optional stage総使用率(選択後 / 選択前) ... 0.071429 / 0.214286
[Task12]
        ステージ数(選択後 / 選択前) ... 0 / 1
        Optional stage総使用率(選択後 / 選択前) ... 0.000000 / 0.363636

        Mandatory stageの総使用率 ... 3.439430
        選択前Optional stageの総使用率 ... 3.827904
        選択後Optional stageの総使用率 ... 0.538690

[Optional stage選択前]
        Mandatory stageと選択後Optional stageの総使用率 ... 7.267334
[Optional stage選択後]
        Mandatory stageと選択後Optional stageの総使用率 ... 3.978120
```

図 6.3.1.1.1：Optional stage 選択部分の結果（提案手法・貪欲法）

図 6.3.1.1.1 の結果より、12 個のタスクについて、ステージ数（選択後 / 選択前）、Optional stage 総使用率（選択後 / 選択前）が表示されている。ステージ数は（Optional stage 選択後の数）/（Optional stage 選択前の数）の形式で表示されており、例として、Task1 については 0 / 2 と表示されているが、Optional stage 選択前は全部で Optional stage 数は 2 個であるのに対し、Optional stage 選択の組合せ最適化問題を貪欲法アルゴリズムで解いたことにより、Task1 の

組合せは 0 個という解が得られたことを示している。次に Optional stage 総使用率は（選択後の Optional stage のプロセッサ総使用率）/（選択前の Optional stage のプロセッサ総使用率）の形式で表示されている。例として、Task1 については 0.000000 / 0.375000 と表示されているが、Optional stage 選択前の 2 個の Optional stage のプロセッサ総使用率は 0.375 であるのに対し、Optional stage 選択後の組合せは 0 個のため、プロセッサ総使用率は 0 と表示されている。

次に、Mandatory stage の総使用率が表示されており、12 個のタスクの全ての Mandatory stage のプロセッサ総使用率が 3.439430 であることを示している。そして、選択前 Optional stage の総使用率と選択後の Optional stage の総使用率も表示されている。全ての Optional stage のプロセッサ使用率は 3.827904 であるのに対し、Optional stage 選択の結果、プロセッサ使用率は 0.538690 に変化している。

最後に、Optional stage 選択前と選択後での、全てのステージ（Mandatory stage と Optional stage）のプロセッサ使用率の合計が表示されている。Optional stage 選択前では、全てのステージのプロセッサ使用率の合計は 7.267334 である。現在プロセッサ数は 4 としているため、式(2.2.7.3)の P-Fair スケジューリングアルゴリズムの条件を満たさない。従って Optional stage 選択前のタスクセットで P-Fair スケジューリングを行うとデッドラインミスが発生する。”(プロセッサ数)－(全 Mandatory stage のプロセッサ使用率)”をナップサックの容量として Optional stage の組合せ最適化問題を解き、Optional stage の組合せを算出した。Optional stage 選択後のプロセッサ使用率の合計は 3.978120 である。式(2.2.7.3)の条件を満たすため、スケジューリング部分で P-Fair アルゴリズムが適用できる。

以上から、提案手法の Optional stage 選択部分（貪欲法アルゴリズム）が正常に機能していることを確認した。Optional stage 選択部分では、スケジューリング部分で P-Fair アルゴリズムが適用できるように最適な Optional stage の組合せを算出している。次にスケジューリング部分の結果を図 6.3.1.1.2 に示す。図中の赤色のサブタスクは Mandatory stage、青色のサブタスクは Optional stage を示す。図 6.3.1.1.2 の結果より、正常に P-Fair スケジューリングが実行されていることを確認した。

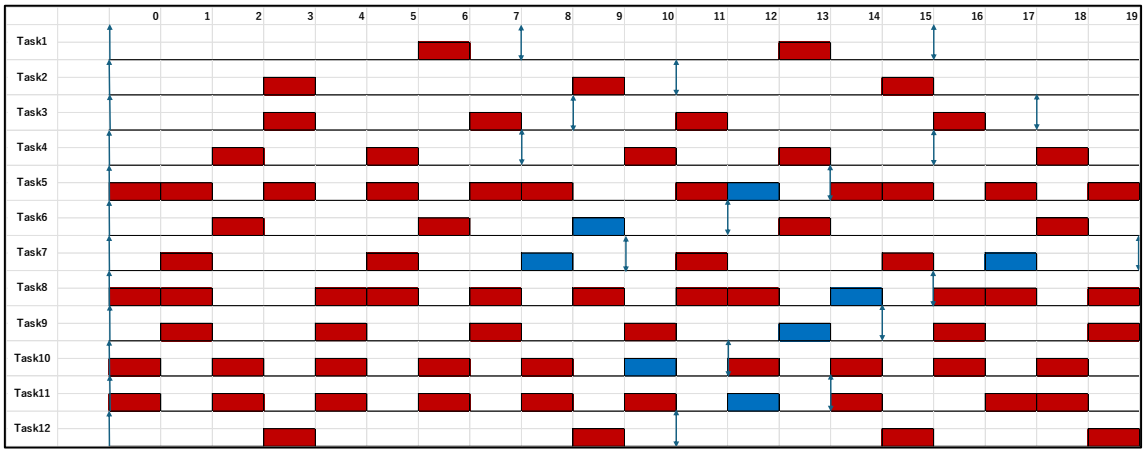

図 6.3.1.1.2：スケジューリング部分の結果（提案手法・貪欲法）

（赤色：Mandatory stage、青色：Optional stage）

6.3.1.2 厳密解法（動的計画法）

提案手法のスケジューリングシミュレータの Optional stage 選択部分について、動的計画法アルゴリズムが正常に機能しているかを確認する。Optional stage 選択部分の結果を図 6.3.1.2.1 に示す。

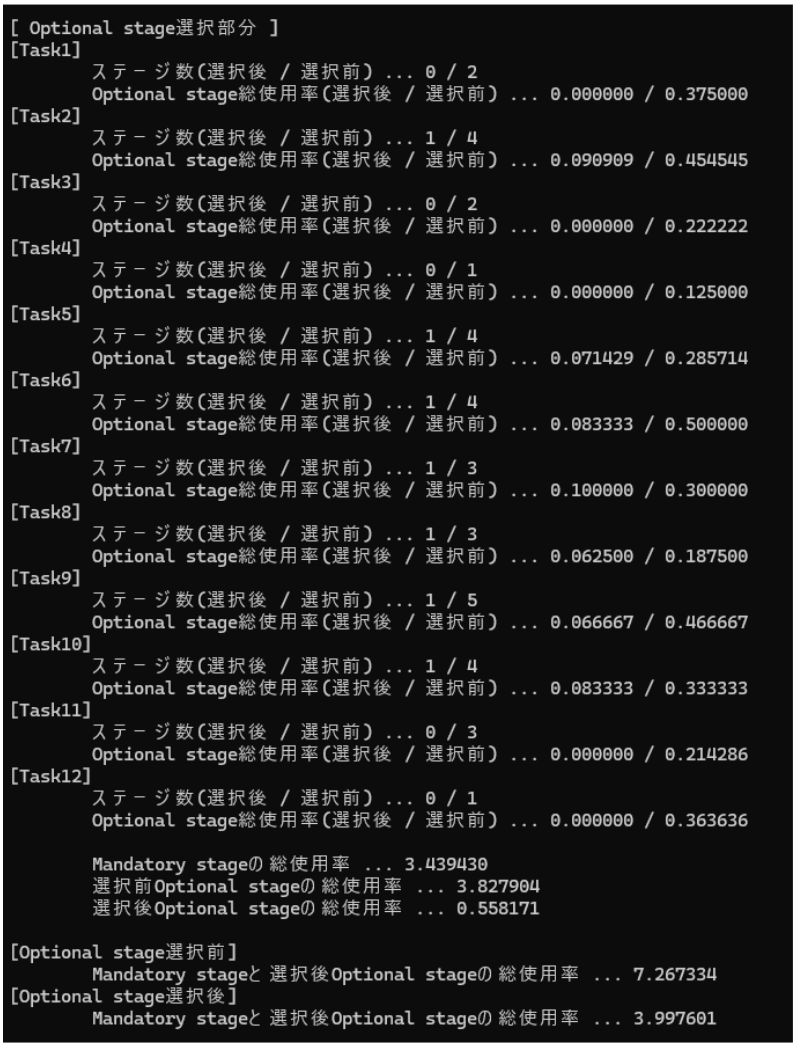

図 6.3.1.2.1：Optional stage 選択部分の結果（提案手法・動的計画法）

貪欲法アルゴリズムと同様、動的計画法アルゴリズムについても、Optional stage 選択部分が正常に機能していることを確認した。スケジューリング部分の結果についても図 6.3.1.2.2 に示す。図中の赤色のサブタスクは Mandatory stage、青色のサブタスクは Optional stage を示す。図 6.3.1.2.2 の結果より、正常に P-Fair スケジューリングが実行されていることを確認した。

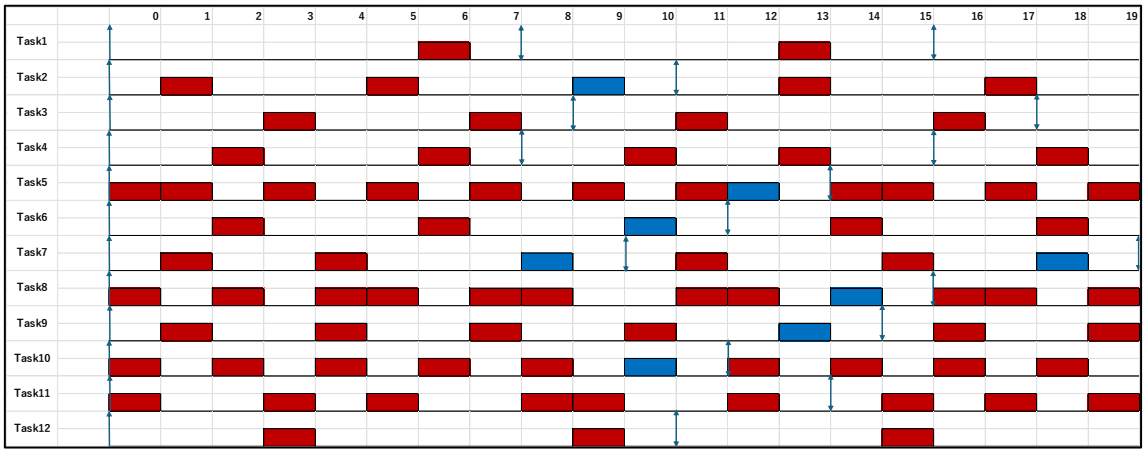

図 6.3.1.2.2：スケジューリング部分の結果（提案手法・動的計画法）
（赤色：Mandatory stage、青色：Optional stage）

6.3.2 比較手法の確認

比較手法のスケジューリングシミュレータの Optional stage 選択部分について、先行研究[2]から拡張した動的計画法アルゴリズムが正常に機能しているかを確認する。比較手法では、初期タスクセットからサブセットを 4 個のプロセッサへ割り当て、各プロセッサで独立してスケジューリングシミュレータを実行する。従って、Optional stage 選択部分とスケジューリング部分の結果はプロセッサの個数分（4 個分）だけ存在する。4 個のプロセッサをそれぞれプロセッサ 1、プロセッサ 2、プロセッサ 3、プロセッサ 4 とすると、プロセッサ 1 についての Optional stage 選択部分の結果を図 6.3.2.1、スケジューリング部分の結果を図 6.3.2.2 に示す。

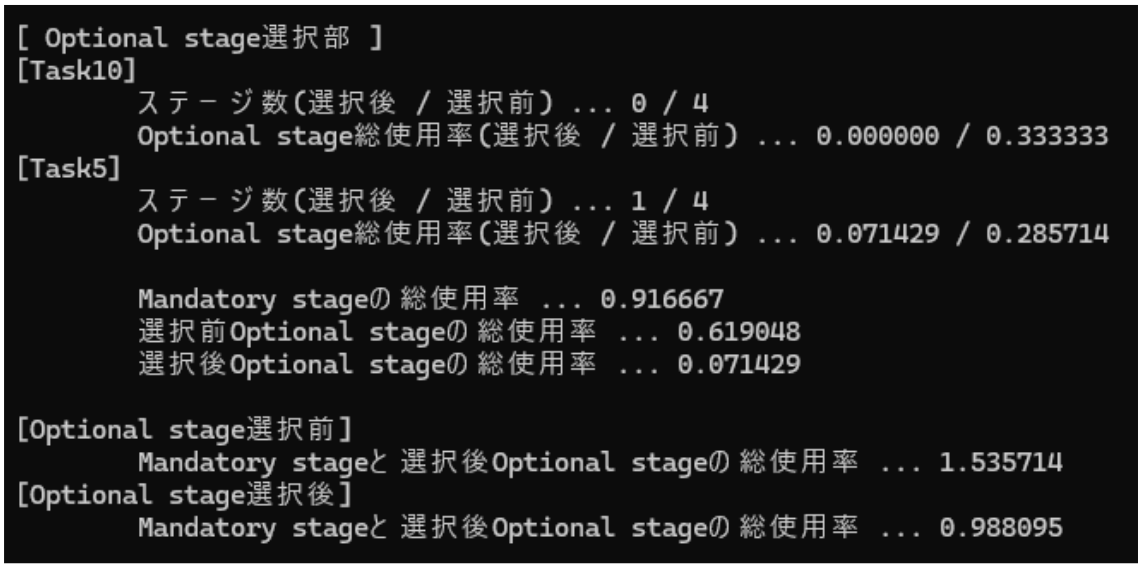

図 6.3.2.1：Optional stage 選択部分の結果（比較手法・プロセッサ 1）

比較手法では、初期タスクセットから 4 個のプロセッサへタスクのサブセットを割り当てる。プロセッサ 1 には Task10 と Task5 が割り当てられている。Task10 の Optional stage 数は、元々は 4 であるのに対し、動的計画法アルゴリズムの結果、選択後は 0 となっていることを確認した。一方、Task5 の Optional stage 数は、元々は 4 であるのに対し、選択後は 1 となっていることを確認した。

Optional stage 選択前と選択後での、全てのステージ（Mandatory stage と Optional stage）のプロセッサ使用率の合計より、Optional stage 選択前では、全てのステージのプロセッサ使用率の合計は 1.535714 である。式(2.2.6.1)を満たさないため、EDF スケジューリングアルゴリズムの条件を満たさない。従って Optional stage 選択前のタスクセットで EDF スケジューリングを行うとデッドラインミスが発生する。Optional stage 選択部分で、プロセッサ使用率の合計が”1－(全 Mandatory stage のプロセッサ使用率)”以下となる時の、最大の合計精度を獲得できる最小実行時間の組合せを算出する。Optional stage 選択後では、全てのステージのプロセッサ使用率の合計は 0.988095 である。式(2.2.6.1)を満たすため、スケジューリング部分で EDF スケジューリングアルゴリズムを適用できる。

次に、スケジューリング部分の結果を図 6.3.2.2 に示す。図中の赤色のタスクは Mandatory stage、青色のタスクは Optional stage を示す。図 6.3.2.2 の結果より、正常に EDF スケジューリングが実行されていることを確認した。

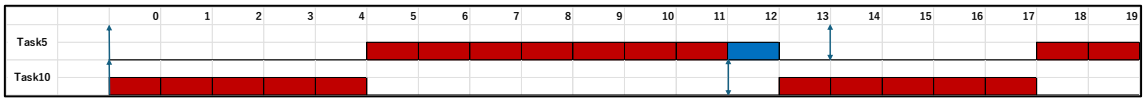

図 6.3.2.2：スケジューリング部分の結果（比較手法・プロセッサ 1）
（赤色：Mandatory stage、青色：Optional stage）

同様に、プロセッサ 2 についての Optional stage 選択部分の結果を図 6.3.2.3、スケジューリング部分の結果を図 6.3.2.4、プロセッサ 3 についての Optional stage 選択部分の結果を図 6.3.2.5、スケジューリング部分の結果を図 6.3.2.6、プロセッサ 4 についての Optional stage 選択部分の結果を図 6.3.2.7、スケジューリング部分の結果を図 6.3.2.8 に示す。いずれのプロセッサにおいても、Optional stage 選択部分とスケジューリング部分が正常に機能していることを確認した。

```
[ Optional stage選択部 ]
[Task1]
        ステージ数(選択後 / 選択前) ... 0 / 2
        Optional stage総使用率(選択後 / 選択前) ... 0.000000 / 0.375000
[Task9]
        ステージ数(選択後 / 選択前) ... 1 / 5
        Optional stage総使用率(選択後 / 選択前) ... 0.066667 / 0.466667
[Task8]
        ステージ数(選択後 / 選択前) ... 0 / 3
        Optional stage総使用率(選択後 / 選択前) ... 0.000000 / 0.187500

        Mandatory stageの 総使用率 ... 0.891667
        選択前Optional stageの 総使用率 ... 1.029167
        選択後Optional stageの 総使用率 ... 0.066667

[Optional stage選択前]
        Mandatory stageと 選択後Optional stageの 総使用率 ... 1.920833
[Optional stage選択後]
        Mandatory stageと 選択後Optional stageの 総使用率 ... 0.958333
```

図 6.3.2.3：Optional stage 選択部分の結果（比較手法・プロセッサ 2）

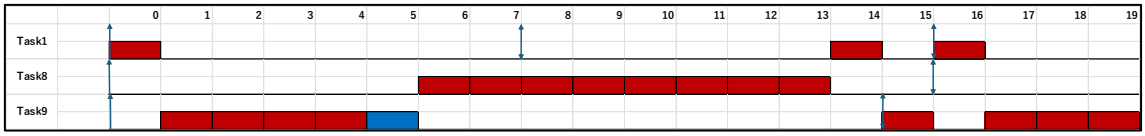

図 6.3.2.4：スケジューリング部分の結果（比較手法・プロセッサ 2）
（赤色：Mandatory stage、青色：Optional stage）

```
[ Optional stage選択部 ]
[Task4]
        ステージ数(選択後 / 選択前) ... 0 / 1
        Optional stage総使用率(選択後 / 選択前) ... 0.000000 / 0.125000
[Task3]
        ステージ数(選択後 / 選択前) ... 0 / 2
        Optional stage総使用率(選択後 / 選択前) ... 0.000000 / 0.222222
[Task11]
        ステージ数(選択後 / 選択前) ... 1 / 3
        Optional stage総使用率(選択後 / 選択前) ... 0.071429 / 0.214286

        Mandatory stageの総使用率 ... 0.900794
        選択前Optional stageの総使用率 ... 0.561508
        選択後Optional stageの総使用率 ... 0.071429

[Optional stage選択前]
        Mandatory stageと選択後Optional stageの総使用率 ... 1.462302
[Optional stage選択後]
        Mandatory stageと選択後Optional stageの総使用率 ... 0.972222
```

図 6.3.2.5：Optional stage 選択部分の結果（比較手法・プロセッサ 3）

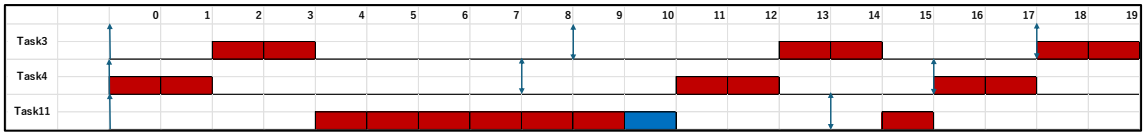

図 6.3.2.6：スケジューリング部分の結果（比較手法・プロセッサ 3）

（赤色：Mandatory stage、青色：Optional stage）

```
[ Optional stage選択部 ]
[Task7]
        ステージ数(選択後 / 選択前) ... 2 / 3
        Optional stage総使用率(選択後 / 選択前) ... 0.200000 / 0.300000
[Task2]
        ステージ数(選択後 / 選択前) ... 0 / 4
        Optional stage総使用率(選択後 / 選択前) ... 0.000000 / 0.454545
[Task12]
        ステージ数(選択後 / 選択前) ... 0 / 1
        Optional stage総使用率(選択後 / 選択前) ... 0.000000 / 0.363636
[Task6]
        ステージ数(選択後 / 選択前) ... 0 / 4
        Optional stage総使用率(選択後 / 選択前) ... 0.000000 / 0.500000

        Mandatory stageの総使用率 ... 0.730303
        選択前Optional stageの総使用率 ... 1.618182
        選択後Optional stageの総使用率 ... 0.200000

[Optional stage選択前]
        Mandatory stageと選択後Optional stageの総使用率 ... 2.348485
[Optional stage選択後]
        Mandatory stageと選択後Optional stageの総使用率 ... 0.930303
```

図 6.3.2.7：Optional stage 選択部分の結果（比較手法・プロセッサ 4）

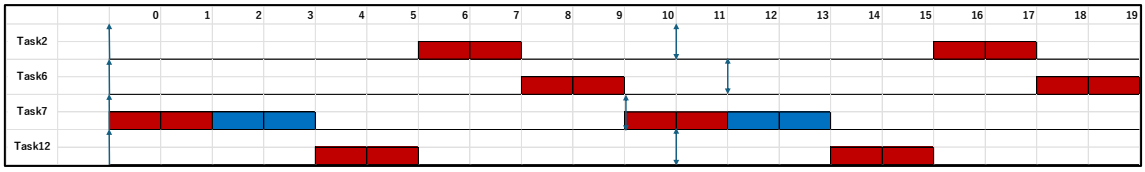

図 6.3.2.8：スケジューリング部分の結果（比較手法・プロセッサ 4）

（赤色：Mandatory stage、青色：Optional stage）

6.4 性能評価

本節では、開発したスケジューリングシミュレータを使用して提案スケジューリング手法の性能を評価する。性能評価では、Optional stage 選択部分で得られた Optional stage における出力精度を評価する。加えて、Optional stage 選択部分の実行時間（CPU 時間）を計測する。

本実験ではプロセッサ数は 4 とする。深層学習タスクの実行時間は最悪実行時間として扱い、実行時間や周期、相対デッドラインは各周期で変動せず常に一定とする。扱うスケジューリングシミュレータはオフラインアルゴリズムとして機能し、タスクセット生成部で生成されたタスクセットは、実行を通して個数やパラメータは変動しないものとする。初期リリースの到来はすべて tick＝0 とし、周期と相対デッドラインは等しく、絶対デッドラインと同時刻に次のリリースが到来するものとする。深層学習タスクはプリエンプティブなタスクとし、ステージ間の境界にかかわらず、ステージの実行中に他のタスクへ実行を切り替えることも可能とする。

6.4.1 出力精度の評価方法

乱数の種をもとにタスクセットを生成し、Optional stage 選択部分により各タスクの実行するステージが確定する。その最終ステージの出力精度をタスクの出力精度として、タスクの個数分の平均値（平均出力精度）を算出する。出力精度の評価では、100 パターンの乱数の種を設定し、各パターンでの平均出力精度を算出する。そして最後に、100 パターン分の平均値を算出し、1 試行の結果（最終平均出力精度）とする（図 6.4.1.1）。本実験では、長さが異なる相対デッドラインを 3 パターン用意し、それぞれのパターンでタスク数を変化させた時の最終平均出力精度を、①比較手法、②提案手法（近似解法モデル）、③提案手法（厳密解法モデル）の 3 手法で調べる（図 6.4.1.2）。近似解法・貪欲法や厳密解法・動的計画法は、Optional stage 選択部分で使用している組合せ最適化問題の解法を示す。

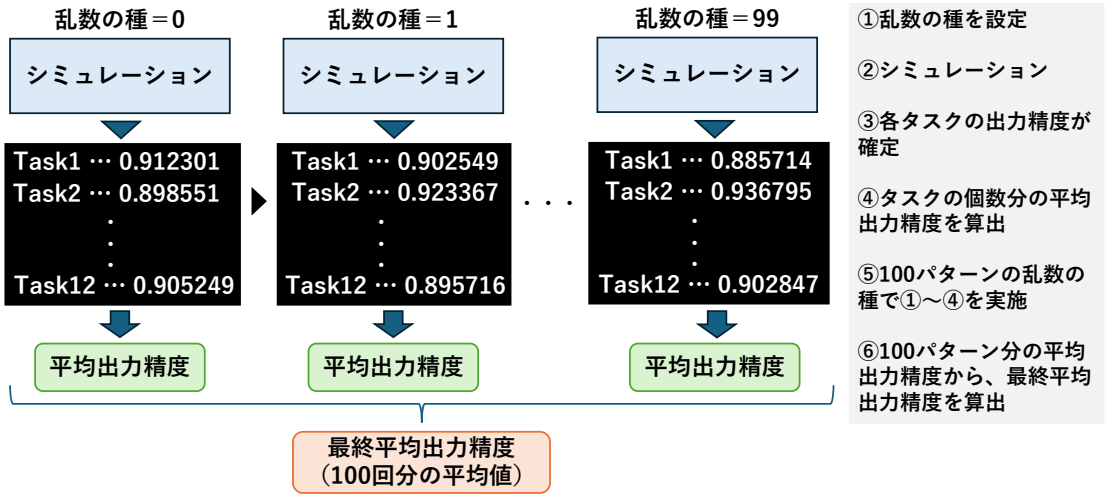

図 6.4.1.1：100 パターンの乱数の種から最終平均出力精度を算出

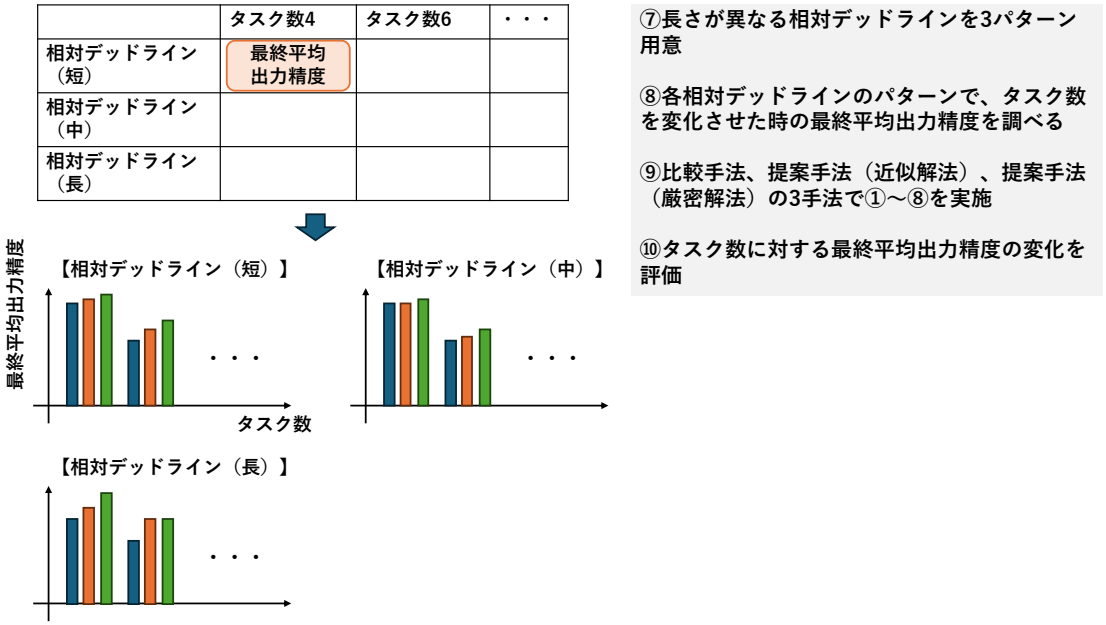

図 6.4.1.2：各相対デッドラインのパターンで、3 手法について
タスク数に対する最終平均出力精度を評価

提案手法と比較手法では、スケジューリング不可能の状態がある。比較手法では、(i)サブセット割り当て後の Mandatory stage のプロセッサ使用率の合計が 1 を超えた時である。あるいは(ii)全タスクの Mandatory stage のプロセッサ使用率の合計がプロセッサ数を超えた時である。提案手法では、(ii)全タスクの Mandatory stage のプロセッサ使用率の合計がプロセッサ数を超えた時である。Mandatory stage は必ず実行するステージのため、そのプロセッサ使用率の合

計がスケジューリング部分の条件を満たさない時、スケジューリングシミュレータはスケジューリング不可能としている。スケジューリング不可能の状態のうち、(i)の状態を Fail1、(ii)の状態を Fail2 として以上のスケジューリング不可能の状態をまとめたものを以下の表 6.4.1.1 に示す。

比較手法	Fail1	サブセット割り当て後の Mandatory stage のプロセッサ使用率の合計が 1 を超えた時
	Fail2	全タスクの Mandatory stage のプロセッサ使用率の合計がプロセッサ数を超えた時
提案手法	Fail2	全タスクの Mandatory stage のプロセッサ使用率の合計がプロセッサ数を超えた時

表 6.4.1.1：スケジューリング不可能の状態

提案手法と比較手法を同じ乱数の種の下で正確に比較するために、このスケジューリング不可能時では試行を両手法でスキップする必要がある。具体的には、乱数の種を更新し、比較手法で Fail1 が発生すれば、比較手法の試行はスキップし提案手法の試行もスキップする。また、比較手法や提案手法で Fail2 が発生すればスキップする。Fail1 や Fail2 が発生すれば、両手法でスキップするようにし、両手法で同じ 100 パターンの乱数の種の下で試行結果を取得するようにした。例を図 6.4.1.3 に示す。比較手法と提案手法で性能評価実験を行っており、乱数の種が 3 の時に比較手法で Fail1 が発生すれば、比較手法ではその乱数の種はスキップし、それに合わせて提案手法でもスキップする。また、比較手法と提案手法で Fail2 が発生すれば、その乱数の種をスキップする。以上のスキップ処理を行った上で 100 パターンの平均出力精度を取得する。

次に、シミュレータプログラムのタスクセット生成部で設定するマクロ定数について述べる。性能評価実験を通して固定のマクロ定数は以下の表 6.4.1.2 の通りである。

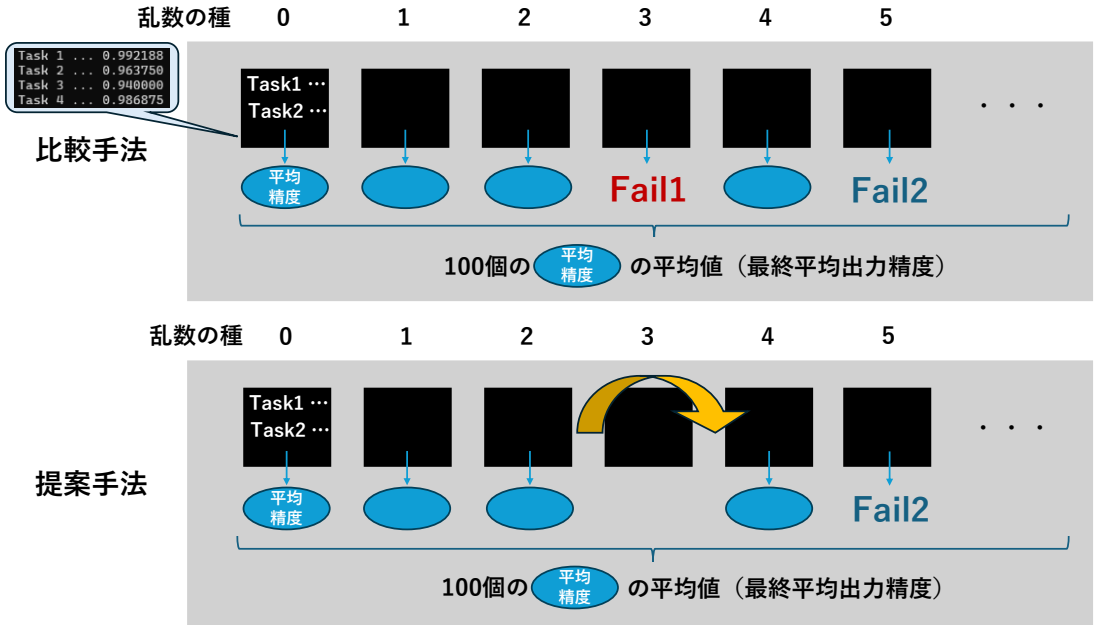

図 6.4.1.3：スケジューリング不可能時の試行のスキップ処理の例

マクロ定数名	設定値（固定値）
MIN_STAGE_NUM	3
MAX_STAGE_NUM	10
EXETIME_MAX_DIV	3
MANDATORY_NUM_MIN	1
MANDATORY_NUM_MAX_DIV	1
MANDATORY_OUT_MIN	70
MANDATORY_OUT_MAX	80

表 6.4.1.2：性能評価実験で固定しているタスクセット生成部のマクロ定数

そして、性能評価実験では 100 パターンの乱数の種、3 パターンの相対デッドライン、タスク数の変化があるため、それに該当するマクロ定数は本実験では変数として扱う。乱数の種を表す変数（RAND_SEED）では、初期値を 0 とし 1 ずつ加算して 100 パターン変化させる。前述の通り、スケジューリング不可能時では RAND_SEED はスキップし、100 パターンを取得するようにする。相対デッドラインは(1)短いパターン、(2)中間のパターン、(3)長いパターンの 3 つを用意する。それぞれのパターンで相対デッドラインに関する変数（DEADLINE_MIN_DIV、DEADLINE_MAX_DIV）は以下の表 6.4.1.3 のように設定する。タスク数の変化は 4, 6, 8, 10, 12, 14 とする。

(1)相対デッドライン（短）	DEADLINE_MIN_DIV・・・0
	DEADLINE_MAX_DIV・・・2
(2)相対デッドライン（中）	DEADLINE_MIN_DIV・・・3
	DEADLINE_MAX_DIV・・・5
(3)相対デッドライン（長）	DEADLINE_MIN_DIV・・・6
	DEADLINE_MAX_DIV・・・8

表 6.4.1.3：3 パターンの相対デッドラインの変数の設定

以上の表 6.4.1.3 の変数の設定によると、タスクセット生成部の相対デッドラインのパラメータは以下の表 6.4.1.4 の範囲の中から無作為に設定される。

(1)相対デッドライン（短）	タスクの総実行時間＋0
	～ タスクの総実行時間＋2
(2)相対デッドライン（中）	タスクの総実行時間＋3
	～ タスクの総実行時間＋5
(3)相対デッドライン（長）	タスクの総実行時間＋6
	～ タスクの総実行時間＋8

表 6.4.1.4：タスクセット生成部における 3 パターンの相対デッドライン

6.4.2 出力精度の評価結果

3 パターンの相対デッドラインにおける、タスク数を 4, 6 8, ..., 14 と変化させた時の 3 手法の最終平均出力精度を示す。3 パターンの相対デッドラインとは、6.4.1 章で述べた通り、(1)相対デッドライン（短）は、タスクの総実行時間＋0～タスクの総実行時間＋2 の間から無作為に設定、(2)相対デッドライン（中）は、タスクの総実行時間＋3～タスクの総実行時間＋5 から無作為に設定、(3)相対デッドライン（長）は、タスクの総実行時間＋6～タスクの総実行時間＋8 から無作為に設定している。それぞれの結果について、(1)相対デッドライン（短）の結果を表 6.4.2.1 と図 6.4.2.1、(2)相対デッドライン（中）の結果を表 6.4.2.2 と図 6.4.2.2、(3)相対デッドライン（長）の結果を表 6.4.2.3 と図 6.4.2.3 に示す。

		タスク数					
		4	6	8	10	12	14
手法	比較手法	0.944034	0.876183	0.815462	0.772935	0.764867	0.758939
	提案手法（近似解法モデル・貪欲法）	0.947831	0.911516	0.841489	0.794415	0.782165	0.773729
	提案手法（厳密解法モデル・動的計画法）	0.947831	0.913904	0.845156	0.795962	0.783158	0.77442

表 6.4.2.1：タスク数変化時の 3 手法の最終平均出力精度
（相対デッドライン（短））

		タスク数					
		4	6	8	10	12	14
手法	比較手法	0.947101	0.923946	0.888739	0.829265	0.784663	0.766329
	提案手法（近似解法モデル・貪欲法）	0.947831	0.948001	0.912758	0.853148	0.808402	0.783966
	提案手法（厳密解法モデル・動的計画法）	0.947831	0.948121	0.913935	0.85476	0.81004	0.784873

表 6.4.2.2：タスク数変化時の 3 手法の最終平均出力精度
（相対デッドライン（中））

		タスク数					
		4	6	8	10	12	14
手法	比較手法	0.947101	0.944144	0.933403	0.897008	0.84038	0.798595
	提案手法（近似解法モデル・貪欲法）	0.947831	0.948687	0.94603	0.915036	0.862733	0.821357
	提案手法（厳密解法モデル・動的計画法）	0.947831	0.948687	0.946141	0.916375	0.864231	0.822992

表 6.4.2.3：タスク数変化時の 3 手法の最終平均出力精度
（相対デッドライン（長））

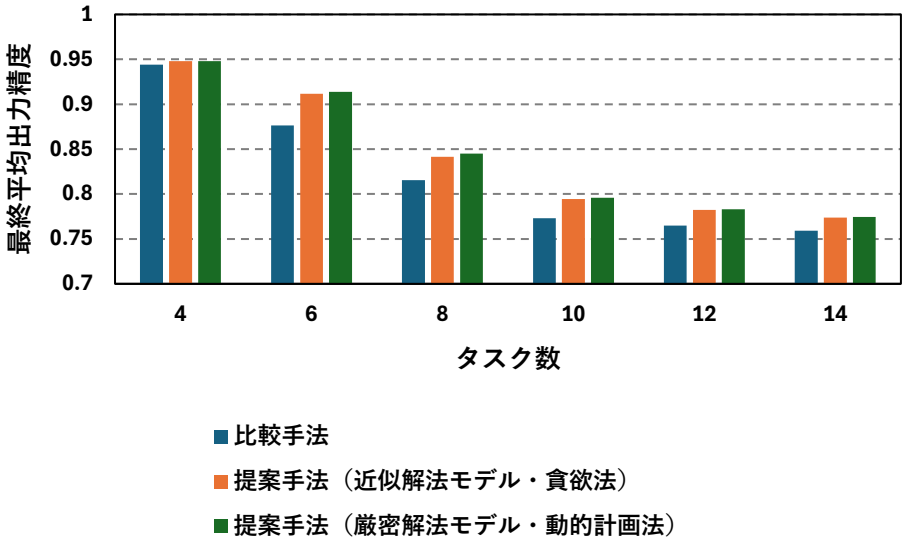

図 6.4.2.1：タスク数変化時の 3 手法の最終平均出力精度
（青色：比較手法、
橙色：提案手法（近似解法モデル）、
緑色；提案手法（厳密解法モデル））
（相対デッドライン（短））

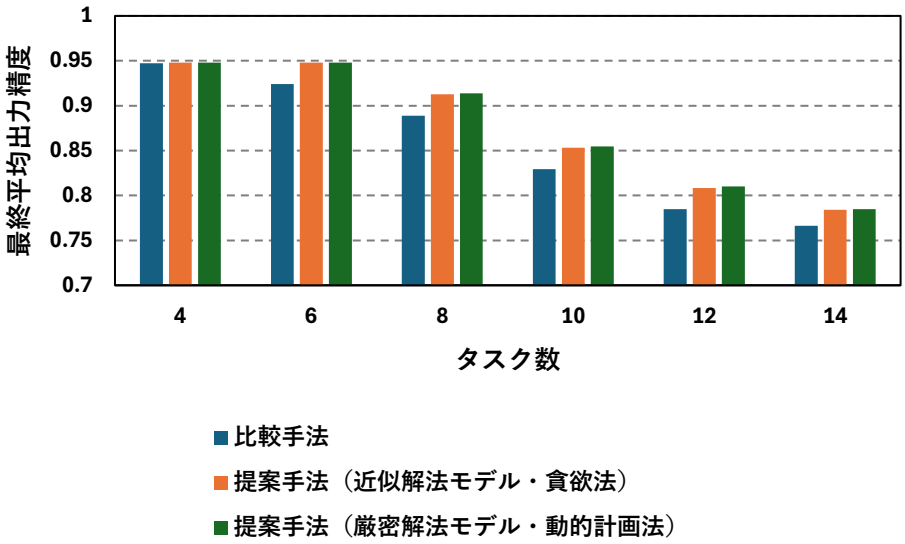

図 6.4.2.2：タスク数変化時の 3 手法の最終平均出力精度
（青色：比較手法、
橙色：提案手法（近似解法モデル）、
緑色；提案手法（厳密解法モデル））
（相対デッドライン（中））

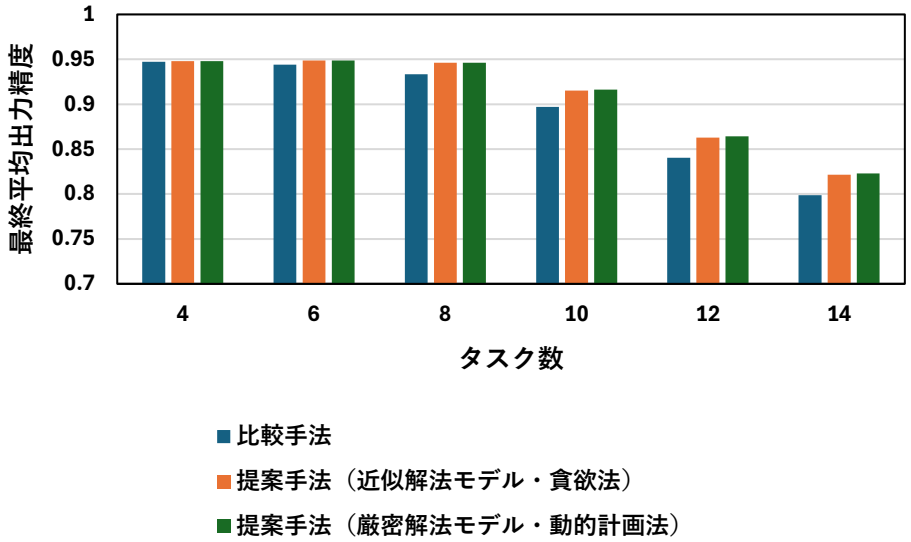

図 6.4.2.3：タスク数変化時の 3 手法の最終平均出力精度
（青色：比較手法、
橙色：提案手法（近似解法モデル）、
緑色；提案手法（厳密解法モデル））
（相対デッドライン（長））

図 6.4.2.1、図 6.4.2.2、図 6.4.2.3 より、全ての相対デッドラインのパターンにおいて、タスク数が増加すれば最終平均出力精度が低下することが 3 手法で確認できた。これは、タスク数が増加すると Mandatory stage の数も増加し、Optional stage の選択の余地が少なくなる。従って、各タスクでは Optional stage が選択されにくくなるため平均出力精度が低下する。

3 手法の最終平均出力精度を比較したところ、比較手法に対して提案手法の 2 手法の結果が高いことを確認した。提案手法の 2 手法については、近似解法（貪欲法）に対して厳密解法（動的計画法）の結果が高いことを確認した。

以上のタスク数変化時の最終平均出力精度の比較から、比較手法よりも提案手法の結果が高いことを確認した。即ち、同じプロセッサ数の環境下で、深層学習タスクセットを比較手法と提案手法へ入力した結果、提案手法の方が時間的制約を保証しつつ高い出力精度を確保できることを確認した。

6.4.3 Optional stage 選択の実行時間の評価方法

Optional stage 選択部分のプログラムの実行時間（CPU 時間）を計測し評価する。実験方法は 6.4.1 節の出力精度の評価実験と同様であり、100 パターンの乱数の種においてシミュレーションを実行し、Optional stage 選択部分の実行時間を計測する。100 パターン分の計測時間の平均値（最終平均実行時間）を算出し、タスク数を変化させた時の 3 手法について調べる。

比較手法では初期タスクセットからサブセットを 4 個のプロセッサへ割り当てて、Optional stage 選択部分が独立で実行されることを想定している。図 6.4.3.1 に比較手法での実行時間の計測環境を示す。シミュレータプログラムでは、4 個のプロセッサの処理を逐次実行しており、4 個のプロセッサでの Optional stage 選択部分での実行時間（(A)～(B)）を計測する。プロセッサの個数分の実行時間を計測すると、プロセッサの個数分の平均実行時間を算出する。そして、100 パターンの乱数の種における平均実行時間を算出すると、100 パターン分の平均値（最終平均実行時間）を算出する。

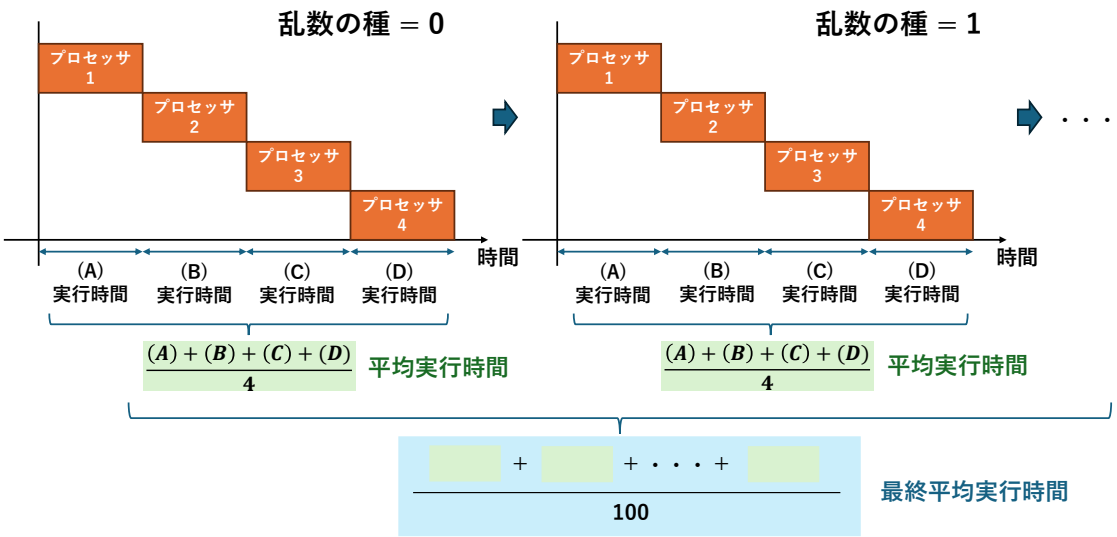

図 6.4.3.1：比較手法での実行時間の計測環境

提案手法では、初期タスクセットをそのまま Optional stage 選択部分へ入力する。図 6.4.3.2 に提案手法での実行時間の計測環境を示す。提案手法では、サブセットを各プロセッサへ割り当てる処理が存在しないため、Optional stage 選択部分の実行時間を計測し、100 パターン分の実行時間の平均値（最終平均実行時間）を算出する。

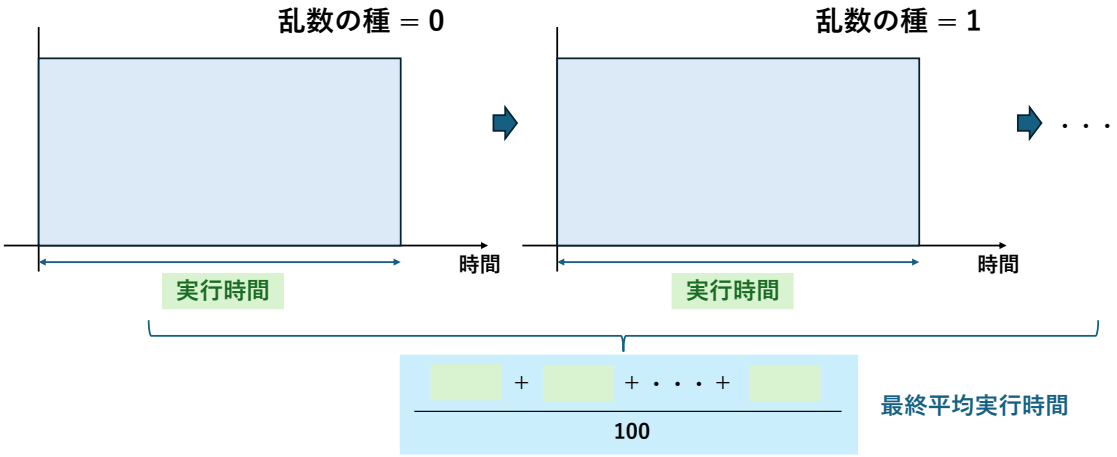

図 6.4.3.2：提案手法での実行時間の計測環境

実行時間の計測には Visual Studio 2022 ではなく、Raspberry Pi 4B を使用している。これは高精度の CPU 時間の計測関数が Linux の環境のみ対応しているためである。CPU 時間の計測には標準ライブラリ（time.h）の clock_gettime 関数を用いる。clock_gettime 関数ではプロセスで消費された CPU 時間を[ns]単位で計測する。実時間ではないためシミュレータプログラムで消費された実行時間を正確に計測できる。被計測部分の前後で clock_gettime 関数を呼び出し、時間差を計算することで被計測部分の実行時間を計測できる。

比較手法の Optional stage 選択部分の optional_stage_select 関数の概要は、以下の図 6.4.3.3 の通りである。optional_stage_select 関数は主に 3 つの処理に分けられ、動的計画法アルゴリズムで使用する表の作成や変数の初期化などの①前処理部分、動的計画法アルゴリズムを使用して表を計算し、組合せを算出する②選択部分、使用した表を解放する③後処理部分である。本実験では動的計画法アルゴリズム本体である②選択部分の実行時間を計測する。そのため、選択部分の前後に clock_gettime 関数を挿入して時間差を計測し、その時間差を図 6.4.3.1 の実行時間とする。

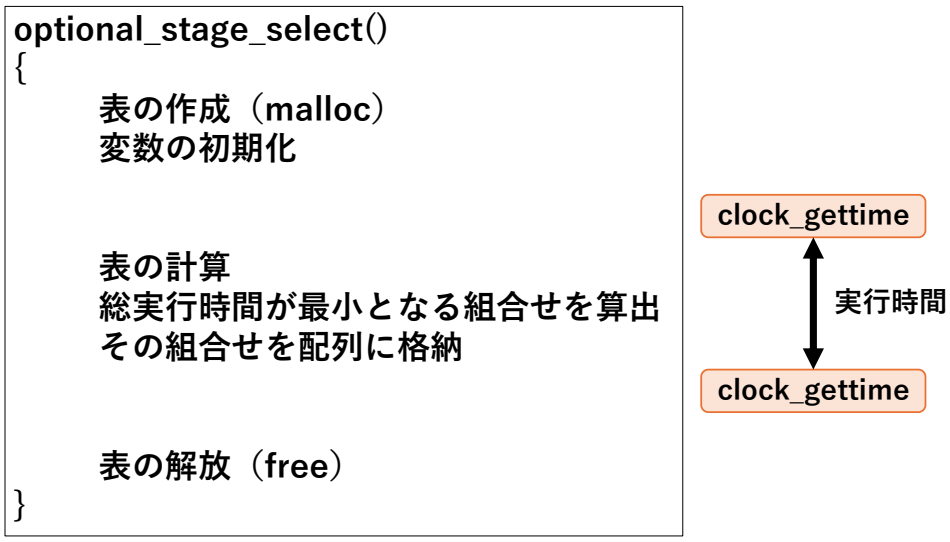

図 6.4.3.3：比較手法の Optional stage 選択部分の計測

提案手法（近似解法モデル）の Optional stage 選択部分の optional_stage_select 関数の概要は、以下の図 6.4.3.4 の通りである。optional_stage_select 関数は主に 3 つの処理に分けられ、貪欲法アルゴリズムで使用する配列（選択・保留用）の作成や変数の初期化などの①前処理部分、貪欲法アルゴリズムを使用して組合せを算出する②選択部分、使用した配列を解放する③後処理部分である。本実験では貪欲法アルゴリズム本体である②選択部分の実行時間を計測する。そのため、選択部分の前後に clock_gettime 関数を挿入して時間差を計測し、その時間差を図 6.4.3.2 の実行時間とする。

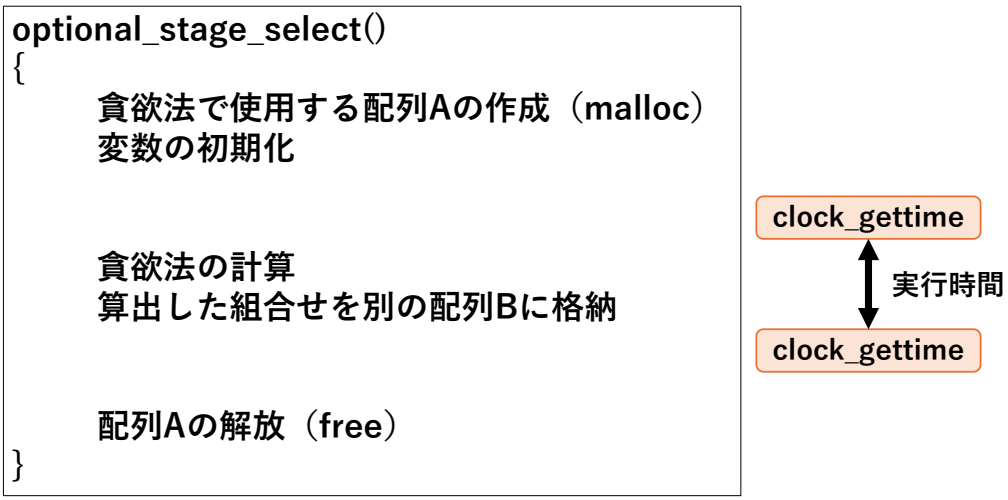

図 6.4.3.4：提案手法（近似解法モデル）の Optional stage 選択部分の計測

提案手法（厳密解法モデル）の Optional stage 選択部分の optional_stage_select 関数の概要は、以下の図 6.4.3.5 の通りである。optional_stage_select 関数は主に 3 つの処理に分けられ、動的計画法アルゴリズムで使用する表の作成や変数の初期化などの①前処理部分、動的計画法アルゴリズムを使用して表を計算し、組合せを算出する②選択部分、使用した表を解放する③後処理部分である。本実験では動的計画法アルゴリズム本体である②選択部分の実行時間を計測する。そのため、選択部分の前後に clock_gettime 関数を挿入して時間差を計測し、その時間差を図 6.4.3.2 の実行時間とする。

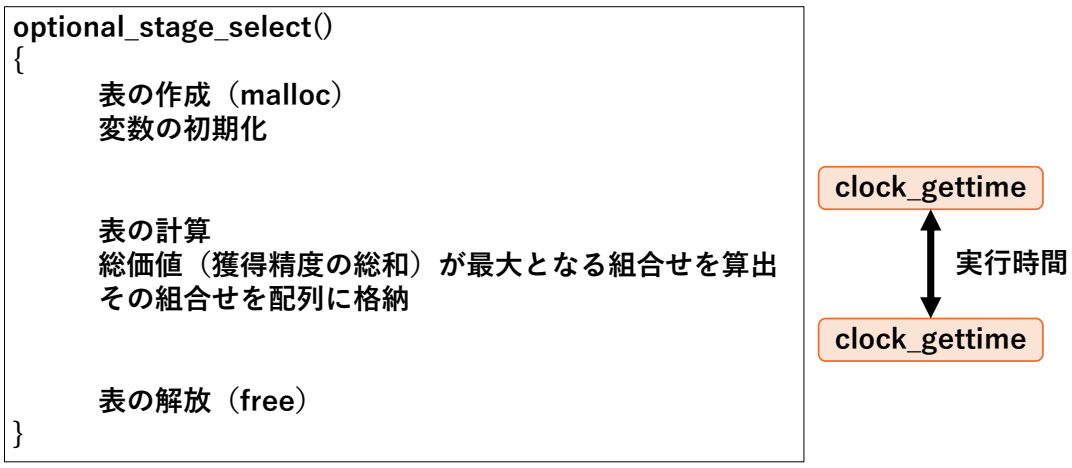

図 6.4.3.5：提案手法（厳密解法モデル）の Optional stage 選択部分の計測

6.4.4 Optional stage 選択の実行時間の評価結果

3 パターンの相対デッドラインにおける、タスク数を 4, 6 8, …, 14 と変化させた時の 3 手法の最終平均実行時間を示す。相対デッドラインのパターンは出力精度の評価実験と同様である。(1)相対デッドライン（短）の結果を表 6.4.4.1 と図 6.4.4.1、(2)相対デッドライン（中）の結果を表 6.4.4.2 と図 6.4.4.2、(3)相対デッドライン（長）の結果を表 6.4.4.3 と図 6.4.4.3 に示す。尚、値の範囲が非常に大きいため、図 6.4.4.1、図 6.4.4.2、図 6.4.4.3 のグラフの縦軸は対数スケールで表示している。

		タスク数					
		4	6	8	10	12	14
手法	比較手法	23858.94	115776.9	216482.1	421471.1	777181.8	1226780
	提案手法（近似解法モデル・貪欲法）	10065.72	13333.74	17842.95	25551.58	35218.54	53605.57
	提案手法（厳密解法モデル・動的計画法）	1178916	1534948	1484383	1540655	1190803	1930524

表 6.4.4.1：タスク数変化時の 3 手法の最終平均実行時間[ns]
（相対デッドライン（短））

		タスク数					
		4	6	8	10	12	14
手法	比較手法	23656.68	110055	205679.2	360771.7	642263.7	1058302
	提案手法（近似解法モデル・貪欲法）	10144.88	13205.2	17682.11	22111.07	27146.45	39958.73
	提案手法（厳密解法モデル・動的計画法）	1397321	2109684	2516447	2100621	1817057	1848341

表 6.4.4.2：タスク数変化時の 3 手法の最終平均実行時間[ns]
（相対デッドライン（中））

		タスク数					
		4	6	8	10	12	14
手法	比較手法	24427.98	112881.4	198731.8	352991.2	584354.7	907305.7
	提案手法（近似解法モデル・貪欲法）	10199.04	14230.8	18660.97	21706.86	27982.94	34190.89
	提案手法（厳密解法モデル・動的計画法）	1533314	2403747	3235220	3318693	3128395	2858491

表 6.4.4.3：タスク数変化時の 3 手法の最終平均実行時間[ns]
（相対デッドライン（長））

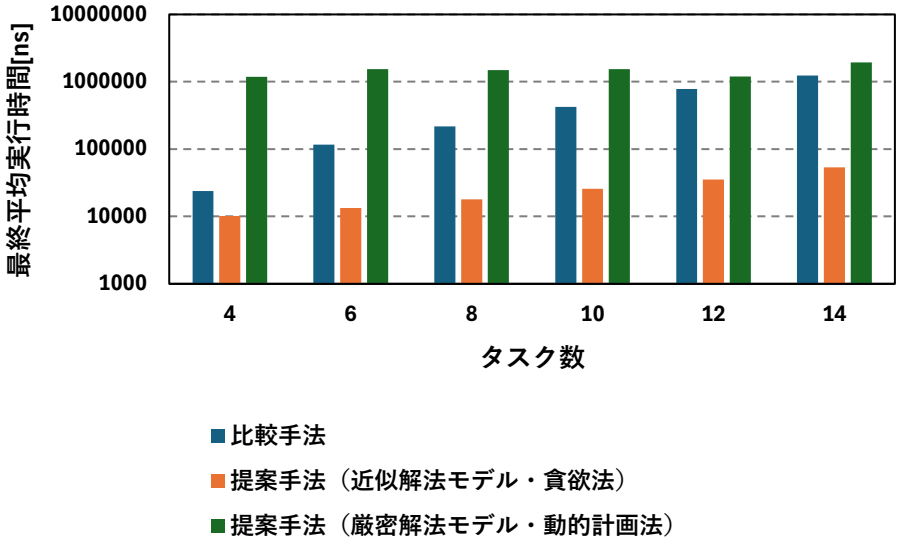

図 6.4.4.1：タスク数変化時の 3 手法の最終平均実行時間[ns]
（青色：比較手法、
橙色：提案手法（近似解法モデル）、
緑色；提案手法（厳密解法モデル））
（相対デッドライン（短））

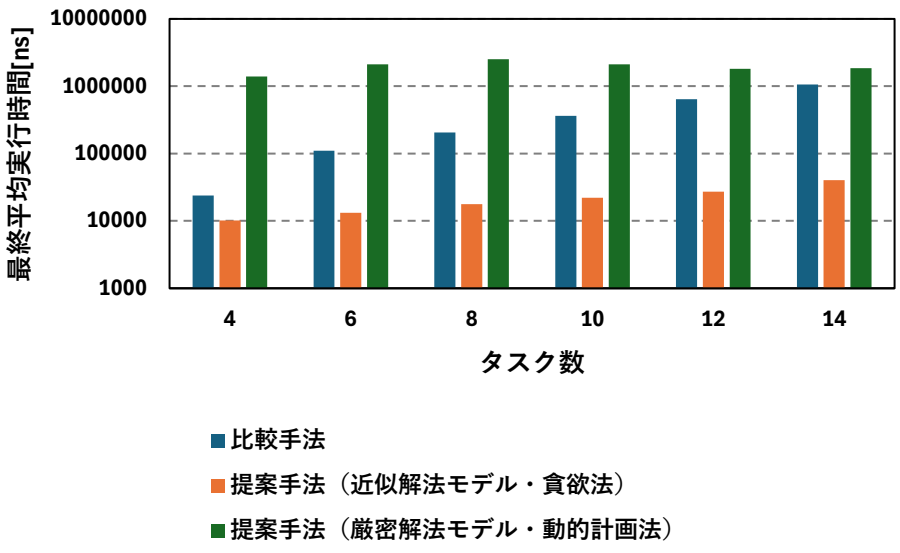

図 6.4.4.2：タスク数変化時の 3 手法の最終平均実行時間[ns]
（青色：比較手法、
橙色：提案手法（近似解法モデル）、
緑色；提案手法（厳密解法モデル））
（相対デッドライン（中））

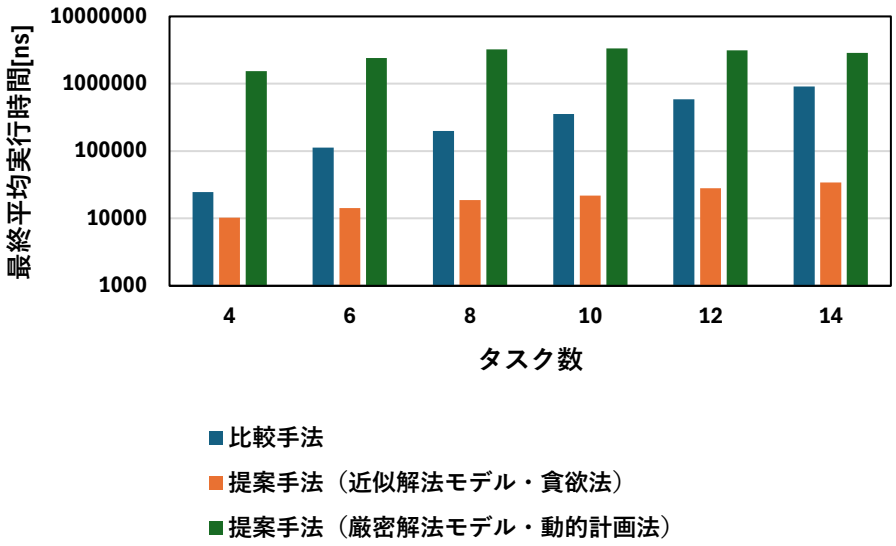

図 6.4.4.3：タスク数変化時の 3 手法の最終平均実行時間[ns]
（青色：比較手法、
橙色：提案手法（近似解法モデル）、
緑色；提案手法（厳密解法モデル））
（相対デッドライン（長））

図 6.4.4.1、図 6.4.4.2、図 6.4.4.3 より、比較手法と提案手法（近似解法モデル）では、タスク数が増加した時に最終平均実行時間は増加している。タスク数が増加した時、比較手法の動的計画法アルゴリズムでは、表の縦軸が増加することになるため計算量が増加する。また、タスク数が増加した時に Optional stage の数も増加するため、比較手法の動的計画法アルゴリズムでは表の要素の計算量は増加する。そして、提案手法の貪欲法アルゴリズムにおいても、扱う Optional stage の数が増加するため計算量が増加する。以上から、比較手法と提案手法（近似解法モデル）の結果は単調増加を示している。

一方で、提案手法（厳密解法モデル）では、タスク数増加時に最終平均実行時間は単調増加していない。この原因は、提案手法の動的計画法アルゴリズムの表の計算にある。提案手法の動的計画法アルゴリズムにおいて、計算する表は縦軸が Optional stage 数、横軸がプロセッサ使用率で最大値が”(プロセッサ数)－(全 Mandatory stage のプロセッサ使用率)”としている。タスク数が少ない時、Optional stage と Mandatory stage の数は少ない傾向にあるため、動的計画法アルゴリズムでは、縦軸が短く横軸が長い表が作成される。一方で、タスク数が多い時、Optional stage と Mandatory stage の数は多い傾向にあるため、動的

計画法アルゴリズムでは、縦軸が長く横軸が短い表が作成される。タスク数の変化に対する動的計画法アルゴリズムの表の面積は、理想的には図 6.4.4.4 のように常に一定となる。動的計画法アルゴリズムの計算量は表の面積で決まるため、タスク数の変化を通して面積が単調増加ではないため、最終平均実行時間も同様の変化になる。

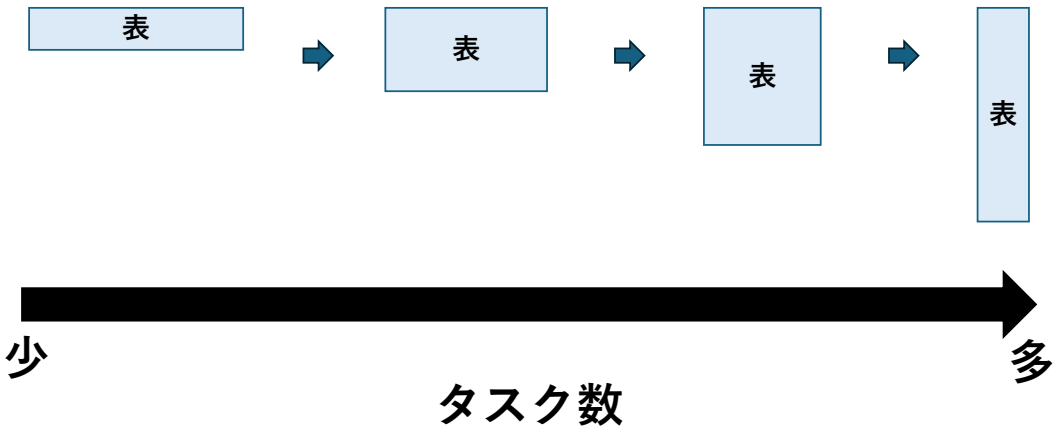

図 6.4.4.4：タスク数の変化に対する動的計画法アルゴリズムの表の面積

提案手法の動的計画法アルゴリズムで使用している表において、縦軸の Optional stage 数を N、横軸の最大値である”(プロセッサ数)－(全 Mandatory stage のプロセッサ使用率)”を W として、表の面積である N*W を計算する。そして実行時間の評価実験と同様の方法で、タスク数の変化に対する N*W を 3 パターンの相対デッドラインについて調べる。その結果を表 6.4.4.4 と図 6.4.4.5 に示す。また、提案手法（厳密解法モデル）のみの、タスク数の変化に対する最終平均実行時間の結果を表 6.4.4.5 と図 6.4.4.6 に示す。図 6.4.4.6 では 3 パターンの相対デッドラインについて結果を示している。

以上から、Optional stage 選択部分の実行時間が最も短いのは提案手法（近似解法モデル）である。最終平均出力精度の結果より、提案手法の 2 手法は大幅な差が無いことから、短い実行時間で高い出力精度を持つ Optional stage の組合せを算出するためには、提案手法（近似解法モデル）が最も適している。

	タスク数					
	4	6	8	10	12	14
相対デッドライン（短）	3111.11	3128.38	2421.61	2023.26	1331.97	1722.97
相対デッドライン（中）	3716.87	4232.07	4220.34	3105.46	2267.3	1982.75
相対デッドライン（長）	4061.39	4993.78	5486.87	4920.49	4073.84	3169.93

表 6.4.4.4：タスク数変化時の 3 パターンの相対デッドラインにおける N*W
（提案手法（厳密解法モデル））

	タスク数					
	4	6	8	10	12	14
相対デッドライン（短）	1178916	1534948	1484383	1540655	1190803	1930524
相対デッドライン（中）	1397321	2109684	2516447	2100621	1817057	1848341
相対デッドライン（長）	1533314	2403747	3235220	3318693	3128395	2858491

表 6.4.4.5：タスク数変化時の 3 パターンの相対デッドラインにおける
最終平均実行時間[ns]
（提案手法（厳密解法モデル））

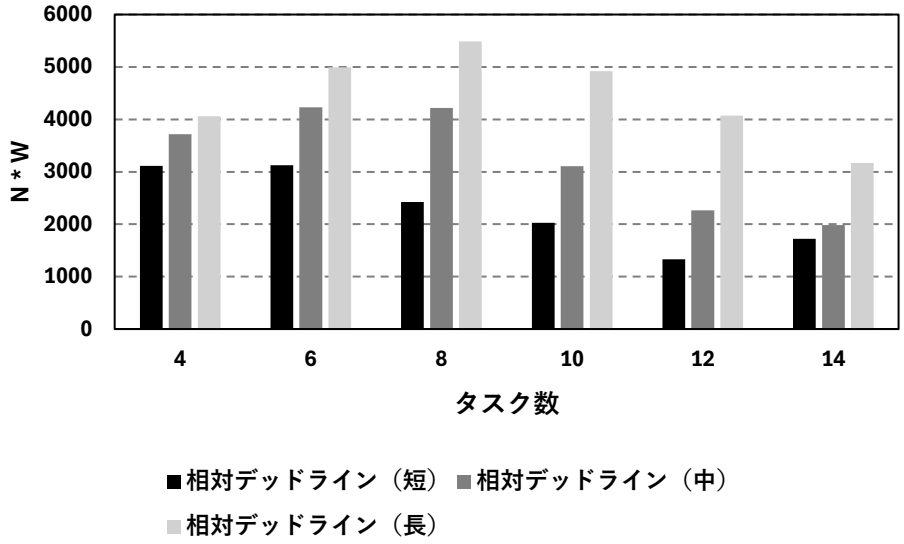

図 6.4.4.5：タスク数変化時の 3 パターンの相対デッドラインにおける N*W
（提案手法（厳密解法モデル））

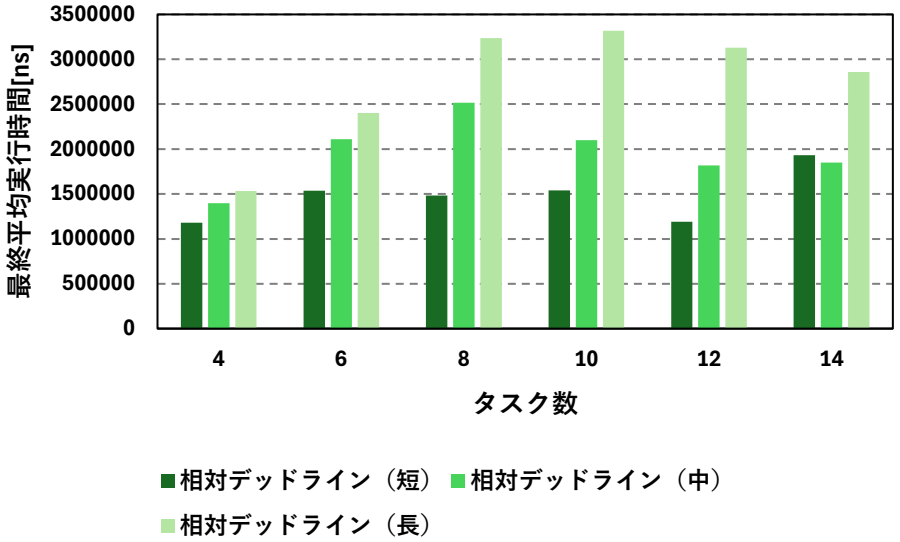

図 6.4.4.6：タスク数変化時の 3 パターンの相対デッドラインにおける
最終平均実行時間[ns]
（提案手法（厳密解法モデル））

図 6.4.4.5 と図 6.4.4.6 より、表の面積を表す N*W と最終平均実行時間は完全に変化は一致しないが、N*W については、3 パターンの相対デッドライン全てにおいて単調増加を示していない。従って、提案手法（厳密解法モデル）において最終平均実行時間が単調増加を示さない原因の一つとして、動的計画法アルゴリズムの表の面積 N*W がタスク数の変化を通して単調増加をしないためであると実験を通して検証した。

提案手法（厳密解法モデル）は、最も高い最終平均出力精度が得られるが、使用する表の面積によって図 6.4.4.1、図 6.4.4.2、図 6.4.4.3 のような長い実行時間となることが分かった。従って、表の面積（計算領域）を削減するようにプログラムを改良することで実行時間を短縮でき、実用的な手法に改良できる。

第7章 おわりに

本研究では、マルチプロセッサ環境で周期的な深層学習タスク群をリアルタイムスケジューリングするための手法を提案し、スケジューリングシミュレータとして開発した。開発した手法は、Optional stage 選択部分の解法により、提案手法（近似解法モデル）と提案手法（厳密解法モデル）に分けられる。そして、開発した手法をスケジューリングシミュレータとして C 言語で実装し機能確認と性能評価を行った。機能確認より、開発したスケジューリングシミュレータは正常に機能していることを確認した。また、性能評価より、タスク数を変化させた時の出力精度は、多様なタスクセットの入力時において、提案手法が高い結果を得られることが分かった。以上から時間的制約を守りつつ比較手法よりも高い出力精度を確保し、マルチプロセッサ環境上でスケジューリング可能なアルゴリズムを開発できた。そして、タスク数を変化させた時の Optional stage 選択部分の実行時間を評価した結果、提案手法（近似解法モデル）は比較手法に対して短い実行時間となった。一方で提案手法（厳密解法モデル）は比較手法に対して長い実行時間の結果を示した。以上から、マルチプロセッサ環境上で深層学習タスク群を高速にスケジューリングするには、提案手法（近似解法モデル）が最適であると実証した。

評価実験を通して、提案手法（近似解法モデル）が最適な手法であることが分かったが、一方、提案手法（厳密解法モデル）は実行時間が長い結果が得られた。提案手法（厳密解法モデル）で使用しているアルゴリズムの改良や最適化などが今後の課題である。また、本研究のスケジューリングシミュレータはオフラインアルゴリズムとして機能している。即ち、タスクの個数や実行時間、相対デッドラインなどは既知かつ常に不変として理想化している。実際には、タスクの個数や推論時の実行時間などは変動するため、タスクが動的に変化する環境で機能するオンラインアルゴリズムとして改良することも課題である。

参考文献

[1] Shuochao Yao, Yifan Hao, Yiran Zhao, Ailing Piao, Huajie Shao, Dongxin Liu, “Eugene: Towards Deep Intelligence as a Service”, 2019 IEEE 39th International Conference on Distributed Computing Systems (ICDCS), pp.1630-1640, (2019)

[2] Shuochao Yao, Yifan Hao, Yiran Zhao, Huajie Shao, Dongxin Liu, Shengzhong Liu, Tianshi Wang, Jinyang Li, Tarek Abdelzaher, “Scheduling Real-time Deep Learning Services as Imprecise Computations”, 2020 IEEE 26th International Conference on Embedded and Real-Time Computing Systems and Applications (RTCSA), pp.1-10, (2020)

[3] Giorgio C. Buttazzo, “Hard Real-Time Computing Systems, Predictable Scheduling Algorithms and Applications, Third Edition”, Springer, pp.23-51, (2011)

[4] 馬場久, “分散リアルタイムシステムにおけるスケジューリングに関する研究（修士論文）”, JAIST Repository, pp.1-49, (2004)

[5] C. L. Liu, James W. Layland, “Scheduling Algorithms for Multiprogramming in a Hard-Real-Time Environment”, Journal of the ACM (JACM), Volume 20, Issue 1, pp.46-61, (1973)

[6] S. K. Baruah, N. K. Cohen, C. G. Plaxton, and D. A. Varvel, “Proportionate Progress: A Notion of Fairness in Resource Allocation”, Algorithmica, 15: pp.600-625, (1996)

[7] S.K. Baruah, J.E. Gehrke, C.G. Plaxton, “Fast scheduling of periodic tasks on multiple resources”, Proceedings of 9th International Parallel Processing Symposium, pp.280-288, (2002)

[8] J.H. Anderson, A. Srinivasan, “Early-release fair scheduling”, Proceedings 12th Euromicro Conference on Real-Time Systems. Euromicro RTS-2000, pp.1-15, (2000)

[9] A. Srinivasan, P. Holman, J.H. Anderson, S. Baruah, “The case for fair multiprocessor scheduling”, Proceedings International Parallel and Distributed Processing Symposium,pp.1-10, (2003)

[10] J.H. Anderson, A. Block, A. Srinivasan, “Quick-release fair scheduling”, RTSS 2003. 24th IEEE Real-Time Systems Symposium, pp.1-12, (2003)

[11] 涌井貞美, “高校数学でわかるディープラーニングのしくみ” pp10-300, ベレ出版(2019)

[12] Surat Teerapittayanon, Bradley McDanel, H.T. Kung, “BranchyNet: Fast Inference via Early Exiting from Deep Neural Networks”, arXiv: 1709.01686, (2017)

[13] J.W.S. Liu, Wei-Kuan Shih, Kwei-Jay Lin, R.Bettati, Jen-Yao Chung, “Imprecise computations”, Proceedings of the IEEE, pp-83-94, (1994)

対外発表

【1】 石井 響, 田中 清史, “深層学習タスクを対象としたマルチプロセッサ用リアルタイムスケジューリング”, 2024 年度電気・情報関係学会北陸支部連合大会, 2024 年 9 月 14 日

謝辞

本研究を進めるにあたり、北陸先端科学技術大学院大学先端科学技術研究科先端科学技術専攻の田中清史教授には、主指導教員として終始適切なご指導をいただきました。心から感謝いたします。また、同専攻の丹教授、リム准教授、木谷講師には副査として適切なご助言をいただきました。お礼申し上げます。

最後に、田中研究室の皆様には、本研究を進めるにあたって多くのご協力をいただきました。また、私の大学院生活を支えてくれた家族に心より感謝いたします。