組込みシステムを対象とした並列実装及びコードの生成と移植に関する研究

明治大学大学院理工学研究科

2022年度

博士学位請求論文

組込みシステムを対象とした並列実装及びコードの生成と移植に関する研究

A study on Parallel Implementation and Transcoder for Embedded System

学位請求者 情報科学専攻

津山 雅彦

論文要旨

近年，計算機のクロック周波数の向上によるプログラムの動作速度の向上は頭打ちになりつつあり，プログラムの高速化を行うための手段としてソフトウェア側の最適化が主流となっている．しかし並列に動作するプログラムの動作や最適化なプログラムの設計には，専用の計算装置や計算機のアーキテクチャに関する知識が必要である．そこで，著者は以下の 2 つを目的として研究を行っている．

- 並列実装のエッジコンピューティングへの応用
- マシンコードの自動生成

著者が所属する研究室では運動中の選手にセンサを取り付け，センサネットワークを動的に構築し，スポーツシーンにおける生体情報取得のためのシステムを実現しようと試みている．そこで，研究室では画像情報に基づいてセンサの位置関係を把握し，ネットワークを構築する Image-Assisted Routing(IAR) を提案しているが，そのためにはリアルタイムな人検出手法が必要であり，それには Informed-Filters とその並列実装が有望であるという研究がある．電力や場所の限られた屋外ではデスクトップ PC 用の強力な GPU を搭載したサーバマシンを用いて Informed-Filters の並列実装をリアルタイムに動作させるのは困難である．そのため,「並列実装のエッジコンピューティングへの応用」は，並列実装された人検出手法を屋外で動作させるために必要なプロセスである．これを行う上で問題となるのはハードウェアの制約や低消費電力，移植先のプラットフォームの制約であり，元の実装の処理性能を保ちつつこれらの問題に対処しなければならない．

また，計算機アーキテクチャに関する知識がなくても最適なプログラムを得られるようにするためにはコンパイラによるプログラムの最適化が重要である．今日の計算機においては，定数の畳み込み，ループの展開といった典型的なコード変換が行われている．しかし，それだけでは人手での最適化に比べると効果が薄い．そこで近年ではプログラムの最適化に機械学習を導入する研究が行われている．そのためのアプローチとして gcc や LLVM といった既存のコンパイラを用いる手法が一般的であるが，著者は「マシンコードの自動生成」というアプローチでこの問題に対処しようと考えている.「マシンコードの自動生成」を行うためのアプローチとして，近年盛んに研究が行われている自然言語処理を用いる方法が考えられるが，自然言語処理を行うためには強力な GPU と莫大な電力が必要であるといった問題がある．

本論文では,「並列実装のエッジコンピューティングへの応用」を目的として Jetson および C++ AMP を用いた移植方法の 2 つの手法を,「マシンコードの自動生成」を目的として模倣学習を用いてマシンコードの生成 · 移植を行う 2 つの手法について提案する．

1 章においては，始めに著者が取り組んでいるプログラムの並列実装のエッジコンピューティング向け移植の目的となる，運動中の選手の生体情報をリアルタイムにモニタリングするためのシステムについて述べる．そして，著者がもうひとつ取り組んでいるマシンコードの自動生成の目的であるプログラムの最適化を行う意義，マシンコードの生成に一見有効そうである自然言語処理の問

題点について述べる．本論文における研究目的は生体情報モニタリングシステムの実現に向けた検出プログラムのエッジコンピューティング向けの移植および，計算リソースの問題を解決したプログラムの自動生成であり，以降の章においてこれらの目的を達成するために行った取り組みについて述べる．

2 章においてはプログラムの並列実装のエッジコンピューティング向け移植の目的となる Image-Assisted Routing(IAR) の核となる技術である画像内の物体検出の既存研究および技術について述べる．まずは，より一般的な画像内から物体を検出しそれに対してクラス分類を行う一般物体検出と呼ばれるタスクとそれに対して提案されてきた手法について述べる．そして，IAR では対象を限定して検出を行う特定物体検出を行うため，それに用いる Informed-Filters とその関する特徴量設計および本論文でエッジコンピューティング向けに移植を行う CUDA を用いた Informed-Filters の並列実装について述べる．

3 章においてはプログラムの生成に用いた模倣学習の手法である Neural Programmer-Interpreters (NPI) について，模倣学習の手法に用いられる時系列データを扱うことのできるネットワークの登場からその応用例について述べる．そして本章の最後に NPI について，そのアーキテクチャから訓練手法まで述べる．

4 章においては Informed-Filters を NVIDIA Jetson Xavier 上への移植を行い，人検出を Xavier 上で動作させるためのシステムの構築を行う．IAR で用いられるカメラから得られる画像の解像度は 3840×2160 である．そこで，その解像度でもリアルタイムに Informed-Filters が実行可能かつ単体で運用可能な Jetson の選定を行い，さらにカメラから Jetson に 4K 画像を入力するための HDMI 出力を UVC に変換する UVC ビデオキャプチャの選定も行った．その結果，3840×2160 の画像 1 枚あたり約 22 fps で処理できることがわかり，リアルタイム処理に十分な速度が示された．

5 章においては IAR に用いるドローン DJI Phantom 4 Pro V2.0 から画像を受け取り，Informed-Filters を用いてリアルタイムに処理をするための移植を行う．Phantom 4 に対し計算機から画像をリクエストするためには DJI Windows SDK を用いる必要があるが，DJI Windows SDK は Universal Windows Platform(UWP) 向けのライブラリであり，UWP のアプリケーションはセキュリティの関係でサンドボックス上実行されるため，CUDA を用いることができないという問題があった．そこで，本論文では UWP 上で GPU を用いて並列処理を行うための手段である C++ AMP を用いて Informed-Filters の並列実装を行い，実装したシステムの検出精度および実行速度の検証を行った．検出精度の評価では深層学習を用いた手法である EfficientDet-D0 および RetinaNet との比較を行った．EfficientDet-D0 および RetinaNet では通常通り訓練しても対象をほとんど検出できないため特別な方法で訓練を行ったが，その 2 つと比べても高い精度が示された．また速度の評価では約 42 fps で動作することが確認され，リアルタイム処理に十分かつシステムの他の部分で遅延が起きても問題がないことが示された．

6 章においては Neural Programmer-Interpreters を用いて RISC-V 命令セット向けのマシンコード生成手法の提案を行う．この手法はマシンコードの生成に向けた最初の手法である．この章では NPI の学習器の訓練方法を工夫することで，NPI を用いたソフト乗算を行うマシンコードの生成を実現する．本章では提案手法に先立って行った予備実験について述べその後に提案手法において行ったことを説明し評価を行う．予備実験では NPI の先行研究で実現されていた加算の筆算をベースとして乗算の筆算の実装を行ったが学習が安定せず，精度もあまりふるわなかった．著者らは予備実験の結果の原因を学習器への入力の多様化と考え，それをふまえて提案手法の実装を行っ

た．提案手法では NPI でマシンコードの生成ができるようにするため，NPI において学習対象のタスクのステップとして与えられるサブプログラムのうち，即時サブプログラムを RISC-V 命令セットの命令に置き換えた．さらに置き換えた即時サブプログラムが実際の RISC-V 命令と同様に動作するように NPI において学習器の外部メモリとしての役割をする Scratch-Pad の改良を行った．提案手法における Scratch-Pad は CPU のレジスタ群としてふるまい，そこから出力される観測情報も RISC-V アーキテクチャの CPU のレジスタから自然に取得できるものとした．最後に提案手法を用いて訓練した NPI の学習器を用いて評価を行った．その結果学習に用いたデータが 5 bit 以内の乗算であったのにも関わらず，32 bit の入力値からなる評価データセットを用いた評価において 100% の精度が得られた．

7 章においては 6 章で提案した訓練手法を拡張し，NPI を用いてコード移植器の構築を行う．これにより，Neural Programmer-Interpreters を用いたコード生成手法がより実用的なものになる．コード移植器は x86_64 から RISC-V へのマシンコードの変換を対象としている．そこで，本章では 6 章で実装した Scratch-Pad が RISC-V だけでなく x86_64 の命令も受けつけるように拡張し，さらに学習器出力するサブプログラムも両方のアーキテクチャ向けに出力できるように学習器訓練方法の変更を行っている．本章で提案する手法においては学習器の最初の訓練に加えて学習器が出力するプログラムのアーキテクチャを変更するための Fine-Tuning を行うが，その方法が問題となった．著者はこの問題に対して，訓練データにおいて即時プログラムを呼ぶ前に Wrapper サブプログラムを挿入することで解決することにした．ここで，最初の訓練は Wrapper サブプログラムを挿入した x86_64 向けのデータセットを用いて行い，Fine-Tuning は RISC-V 向けの Wrapper サブプログラムのデータセットを用いて行う．本章の提案手法に先立ち予備実験も行っている．予備実験では上記の工夫のみを施して訓練を行っているが，Wrapper サブサブプログラムを開始プログラムとして与えた際は RISC-V のマシンコードを出力するものの，本来学習器で行いたいソフト乗算の開始命令を開始プログラムとして与えた際は x86_64 のマシンコードが得られるという結果となった．著者はこの結果が学習器の内部状態によるものと考え，RISC-V 向けに作成した Wrapper サブプログラムのデータセットを x86_64 向けのデータセットの中に入れて訓練することにした．この手法を用いて訓練を行った学習器の評価を行ったところ，サブプログラムの置換はうまくいっていたがその引数がうまく学習できていないという結果になった．この結果から引数の問題を解決することで，NPI を用いてマシンコードの移植を行うためのプラットフォームの実現可能性が示された．

8 章では，本論文の結論および，今後の展望について述べる．

目 次

図 目 次

表 目 次

第1章　序論

1.1　研究背景

近年，計算機のクロック周波数によるプログラムの動作速度の向上は頭打ちになりつつあり，プログラムの高速化を行なうための手段としてはソフトウェア側の最適化が主流となっている．しかし，並列に動作するプログラムの動作や最適なプログラムの設計には専用の計算装置や計算機のアーキテクチャに関する知識が必要である．そこで，著者は以下の 2 つを目的として研究を行なっている．

- 並列実装のエッジコンピューティングへの応用
- マシンコードの自動生成

著者が所属する研究室では運動中の選手にセンサを取り付け，センサネットワークを動的に構築し，スポーツシーンにおける生体情報取得のためのシステムを実現しようと試みている．そこで，研究室では画像情報に基づいてセンサの位置関係を把握し，ネットワークを構築する Image-Assisted Routing(IAR) を提案しているが，そのためにはリアルタイムな人検出手法が必要であり，それには Informed-Filters [1] とその並列実装が有望であるという研究がある [2,3]．電力や場所の限られた屋外ではデスクトップ PC 用の強力な GPU を搭載したサーバマシンを用いて Informed-Filters の並列実装をリアルタイムに動作させるのは困難である．そのため,「並列実装のエッジコンピューティングへの応用」は，並列実装された人検出手法を屋外で動作させるために必要なプロセスである．これを行う上で問題となるのはハードウェアの制約や低消費電力，移植先のプラットフォームの制約であり，元の実装の処理性能を保ちつつこれらの問題に対処しなければならない．

また，計算機アーキテクチャに関する知識がなくても最適なプログラムを得られるようにするためにはコンパイラによるプログラムの最適化が重要である．

今日の計算機においては，以下のような典型的なコード変換が自動で行われている．

- ループ不変式の削除
- ループの展開
- 定数の畳み込み
- 処理の並列化

以降ではこれらについて説明する．

ループ不変式の削除というのは，ループ内で同じ値を常に返す式が存在するときにその式をループの前に移動させる手法である．例えばソースコード 1.1 に示すようなコードにおいて，ループ内

で b と c の値が変化しないとき b + c は for 文の前であらかじめ行っておくことで加算命令をループの回数分だけ減らすことができる．

ソースコード 1.1: sample1.c

```
1 int a, b = 10, c = 100;
2 ...
3 for(int i = 0; i < 10; ++i)
4 {
5     ...
6     a = i * (b + c);
7     ...
8 }
```

ループの展開とは，ソースコード 1.2 のように for 文や while 文で表された繰り返し処理の繰り返し回数を削減する，複数行のコードとして展開するといった処理のことである．変数の演算を行う際はメモリからレジスタに値がロードされ，演算を行い，最後に結果をメモリに戻すというような処理が行われるが，これを行うことによってレジスタをより効率的に扱うことができ，さらにループ処理における加算や条件判定を削減することができる．

ソースコード 1.2: sample2.c

```
1  int a[200] = {...};
2
3
4  // for loop
5  for(int i = 0; i < 200; ++i) {
6      ...
7      a[i] = i * b + c;
8  }
9
10 // unrolled example
11 for(int i = 0; i < 200; i+=2) {
12     ...
13     a[i] = i * b + c;
14     a[i+1] = i * b + c;
15 }
```

定数の畳み込みとはソースコード内で不変な式をコンパイル時に評価することである．例えばソースコード 1.1 において，b と c の値が常に変化しないとき (b + c) の部分をコンパイル時に計算して 110 としてしまっても実行結果に影響はない．これにより，加算命令とレジスタの割り当てを削減することができる．

プログラムの並列化は各イテレーションで依存関係のない繰り返し処理を並列に実行することであり，処理速度が頭打ちとなりつつある今日の計算機において最も効果的な高速化手法のひとつである．

しかし，これらの自動最適化だけでは人手での最適化に比べると効果が薄い．そこで近年ではプログラムの最適化に機械学習を導入する研究が行われている [4, 5]．そのためのアプローチとして gcc [6] や LLVM [7] といった既存のコンパイラを用いる手法が一般的である．また，画像処理の分野では意味論的領域分割や物体検出といったタスクが行われているが，これらのタスクを CPU による逐次処理や SIMD で実装するとリアルタイムな処理が実現できず，自動運転や UAV による物体の検出 · 追跡といったタスクに応用できないという問題がある．そこで，これらのタスクをリアルタイムに行えるようにするため，GPU や FPGA を用いた並列実装が行われる．

1.2　研究目的

前節にて述べたように，画像処理の分野において物体検出をリアルタイムで行うために GPU を用いた並列実装が行われるが，強力なデスクトップ向け GPU を用いて UAV から画像を取得し，それに対して検出を行うことは困難である．そして，本研究の 2 つ目の目的であるマシンコードの生成は自然言語生成ですでに行われている，プログラムのソースコードの生成をマシンコードに応用すれば可能ではあるが，モデルの訓練には膨大なコストがかかるという問題がある．そこで著者はこれらの問題の解決に向けた手法の提案を行う．

まず，屋外で人検出手法のリアルタイム実装を行うために NVIDIA Jetson Xavier [8]，および C++ AMP [9] を用いた移植を行う．Jetson Xavier は ARM アーキテクチャの CPU に NVIDIA 製の GPU が搭載されたシングルボードコンピュータもしくはモジュールであり，NVIDIA CUDA [10] で実装されたプログラムを動作させることができるという特徴がある．これを用いることで，CUDA を用いて実装された人検出の並列実装 [3] をほぼそのままエッジコンピューティング向けに移植できると考えられる．また，IAR を行うためには Unmanned Air Vehicle(UAV) から画像を取得し，それに対して人検出処理を行うことも考えなければならない．そこで著者は IAR で用いる UAV として DJI Phantom 4 Pro V2.0 [11] を用いることにしたが，Phantom 4 Pro V2.0 から画像を取得するためには，ユニバーサル Windows プラットフォーム (UWP) [12] を対象とした DJI Windows SDK を用いる必要がある．しかし，UWP 上のアプリケーションはサンドボックス環境で実行されるため，CUDA 環境が利用できないという問題がある．そこで著者は UWP 上で並列計算を行う手段である C++ AMP を用いることでこの問題を解決する．

そして，マシンコードの生成にはよりモデルの訓練コストの小さい模倣学習の手法を用いることで自然言語生成モデルを用いた手法の問題点の解決を試みる．前節で述べたように模倣学習では入出力に工夫をすることで長い入出力を行っても高い精度を示す NTM や，さまざまな長さの出力を行う Pointer Networks という手法が提案されてきた．このような手法の中でタスクをステップ毎に分割し，プログラムのように組み立てる Neural Programmer-Interpreters (NPI) という手法が提案された．著者は NPI におけるタスクのステップを CPU の命令として表現し，生成するプログラムの動作をトレースするようにコードを生成することで，自然言語生成モデルにおける訓練コストの問題を解決できるのではないかと考えた．そこでこの着想が実現可能であることを確かめるために，NPI を用いてマシンコードの生成を行う．これにより自然言語生成モデルのようなパラメータ数が膨大な学習器を用いることなく，マシンコードの生成が行えることを示す．さらに NPI によるマシンコード生成手法を拡張して，あるアーキテクチャに対して訓練された学習器に対して Fine-Tuning することによりコード移植器の実装を試みる．これを実現することができればあるタスクに対して訓練された NPI の学習器を再学習することで，学習器が似たような手順を踏むタスクに応用できることを示すことができ，さらにマシンコード生成器の手法を用いて訓練された NPI の学習器があらかじめ学習していないアーキテクチャ向けのマシンコードも生成できることを示すことができる．

1.3　本論文の構成

本論文は全 8 章で構成されている．

2 章においてはプログラムの並列実装のエッジコンピューティング向け移植の目的となる Image-Assisted Routing(IAR) や，その核となる技術である画像内の物体検出の既存研究および技術について述べる．まずは，より一般的な画像内から物体を検出しそれに対してクラス分類を行う一般物体検出と呼ばれるタスクとそれに対して提案されてきた手法について述べる．そして，IAR では対象を限定して検出を行う特定物体検出を行うため，それに用いる Informed-Filters とその関する特徴量設計および本論文でエッジコンピューティング向けに移植を行う CUDA を用いた Informed-Filters の並列実装について述べる．

3 章においてはプログラムの生成に用いた模倣学習の手法である Neural Programmer-Interpreters (NPI) について，模倣学習の手法に用いられる時系列データを扱うことのできるネットワークの登場からその応用例について述べる．そして本章の最後に NPI について，そのアーキテクチャから訓練手法まで述べる．

4 章においては Informed-Filters を NVIDIA Jetson Xavier 上への移植を行い，人検出を Xavier 上で動作させるためのシステムの構築を行う．IAR で用いられるカメラから得られる画像の解像度は 3840×2160 である．そこで，その解像度でもリアルタイムに Informed-Filters が実行可能かつ単体で運用可能な Jetson の選定を行い，さらにカメラから Jetson に 4K 画像を入力するための HDMI 出力を UVC に変換する UVC ビデオキャプチャの選定も行った．その結果，3840×2160 の画像 1 枚あたり約 22 fps で処理できることがわかり，リアルタイム処理に十分な速度が示された．

5 章においては IAR に用いるドローン DJI Phantom 4 Pro V2.0 から画像を受け取り，Informed-Filters を用いてリアルタイムに処理をするための移植を行う．Phantom 4 に対し計算機から画像をリクエストするためには DJI Windows SDK を用いる必要があるが，DJI Windows SDK は Universal Windows Platform(UWP) 向けのライブラリであり，UWP のアプリケーションはセキュリティの関係でサンドボックス上実行されるため，CUDA を用いることができないという問題があった．そこで，本論文では UWP 上で GPU を用いて並列処理を行うための手段である C++ AMP を用いて Informed-Filters の並列実装を行い，実装したシステムの検出精度および実行速度の検証を行った．検出精度の評価では深層学習を用いた手法である EfficientDet-D0 および RetinaNet との比較を行った．EfficientDet-D0 および RetinaNet では通常通り訓練しても対象をほとんど検出できないため特別な方法で訓練を行ったが，その 2 つと比べても高い精度が示された．また速度の評価では約 42 fps で動作することが確認され，リアルタイム処理に十分かつシステムの他の部分で遅延が起きても問題がないことが示された．

6 章においては Neural Programmer-Interpreters を用いて RISC-V 命令セット向けのマシンコード生成手法の提案を行う．この手法はマシンコードの生成に向けた最初の手法である．この章では NPI の学習器の訓練方法を工夫することで，NPI を用いたソフト乗算を行うマシンコードの生成を実現する．本章では提案手法に先立って行った予備実験について述べその後に提案手法において行ったことを説明し評価を行う．予備実験では NPI の先行研究で実現されていた加算の筆算をベースとして乗算の筆算の実装を行ったが学習が安定せず，精度もあまりふるわなかった．著者らは予備実験の結果の原因を学習器への入力の多様化と考え，それをふまえて提案手法の実装を行った．提案手法では NPI でマシンコードの生成ができるようにするため，NPI において学習対象のタスクのステップとして与えられるサブプログラムのうち，即時サブプログラムを RISC-V 命令セットの命令に置き換えた．さらに置き換えた即時サブプログラムが実際の RISC-V 命令と同様に動作するように NPI において学習器の外部メモリとしての役割をする Scratch-Pad の改良を行っ

た．提案手法における Scratch-Pad は CPU のレジスタ群としてふるまい，そこから出力される観測情報も RISC-V アーキテクチャの CPU のレジスタから自然に取得できるものとした．最後に提案手法を用いて訓練した NPI の学習器を用いて評価を行った．その結果学習に用いたデータが 5 bit 以内の乗算であったのにも関わらず，32 bit の入力値からなる評価データセットを用いた評価において 100% の精度が得られた．

7 章においては 6 章で提案した訓練手法を拡張し，NPI を用いてコード移植器の構築を行う．これにより，Neural Programmer-Interpreters を用いたコード生成手法がより実用的なものになる．コード移植器は x86_64 から RISC-V へのマシンコードの変換を対象としている．そこで，本章では 6 章で実装した Scratch-Pad が RISC-V だけでなく x86_64 の命令も受けつけるように拡張し，さらに学習器出力するサブプログラムも両方のアーキテクチャ向けに出力できるように学習器訓練方法の変更を行っている．本章で提案する手法においては学習器の最初の訓練に加えて学習器が出力するプログラムのアーキテクチャを変更するための Fine-Tuning を行うが，その方法が問題となった．著者はこの問題に対して，訓練データにおいて即時プログラムを呼ぶ前に Wrapper サブプログラムを挿入することで解決することにした．ここで，最初の訓練は Wrapper サブプログラムを挿入した x86_64 向けのデータセットを用いて行い，Fine-Tuning は RISC-V 向けの Wrapper サブプログラムのデータセットを用いて行う．本章の提案手法に先立ち予備実験も行っている．予備実験では上記の工夫のみを施して訓練を行っているが，Wrapper サブサブプログラムを開始プログラムとして与えた際は RISC-V のマシンコードを出力するものの，本来学習器で行いたいソフト乗算の開始命令を開始プログラムとして与えた際は x86_64 のマシンコードが得られるという結果となった．著者はこの結果が学習器の内部状態によるものと考え，RISC-V 向けに作成した Wrapper サブプログラムのデータセットを x86_64 向けのデータセットの中に入れて訓練することにした．この手法を用いて訓練を行った学習器の評価を行ったところ，サブプログラムの置換はうまくいっていたがその引数がうまく学習できていないという結果になった．この結果から引数の問題を解決することで，NPI を用いてマシンコードの移植を行うためのプラットフォームの実現可能性が示された．

8 章では，本論文の結論および，今後の展望について述べる．

第2章　物体検出

物体検出技術は機械学習および画像処理分野における重要なタスクのひとつであり，今日においても精度向上のためさまざまな研究が行われている [13–17]．また，我々の日常生活においてもさまざまな場所で物体検出が応用されつつある [18]．

それらの中でも著者はスポーツシーン解析に着目し，本稿ではそれに用いるためのシステムの提案を行う．まずはスポーツシーン解析について述べる．

2.1　スポーツシーン解析

現在，運動中のスポーツ選手のバイタルデータをリアルタイムに取得し体調管理やトレーニング効率の向上に役立てることを目指した研究が行われている [19]．この目的を達成するためには，運動中に着用可能なセンサノード，センサノードから情報を適切に集めるための無線ネットワーキング方式およびリアルタイムにデータ収集を行うことができるシステムの開発が不可欠である．

2.1.1　リアルタイムバイタルセンシングシステム

人に装着された無線センサノードを用いてリアルタイムバイタルセンシングを行う手法として様々なアプローチが考えられるが，著者は最終的なユーザビリティの向上を考慮して無線局免許を必要としない無線通信方式を採用することとした．この制約によって送信電力が制限され，センサノードの通信可能な距離が短くなるため，センサノードの位置に応じて動的にネットワーク構成を変更しデータをバケツリレーのように転送するマルチホップネットワークを利用する必要性が生じる．この方式には無線通信に必要となる電力を少なくすることで，センサノードの小型化にも貢献できるという大きな利点もある．

一般にマルチホップネットワークの動的ルーティングを行うためには，受信信号強度 (Received Signal Strength Indicator; RSSI) がネットワーク構成を変更するための手掛りとして用いられることが多い．しかし，スポーツ選手や運動中の人物に装着されたセンサノードによって構成されるネットワークを対象とする場合にはノードの密度および移動速度が高いという RSSI の利用には不適切な条件があるため，これを手掛かりとする動的ルーティングは困難である．そのため，動的ルーティングに利用可能な RSSI に代わる手掛りが不可欠となる．

2.1.2　Image-Assisted Routing

RSSI に代わる手掛りとして，映像情報から得られる人物の位置情報を用いる Image-Assisted Routing (IAR) [20, 21] が提案されている．この手法は，地上に設置されたカメラや Unmanned

Aerial Vehicle(UAV) に搭載されたカメラから得られる映像からセンサノードを装着した人物の位置を推定し，その位置情報に基づき動的ルーティングを行おうというものである．本研究でもこのアプローチを採用することとした．IAR による位置推定に基づく動的ネットワーキングを実現するためには，空撮画像から得られる人物の位置情報を用いることが有効である．

UAV から撮影された空撮画像には，図 2.1 のように検出対象は非常に小さく写る．そのため，IAR において人物の位置推定を得るための手法としては物体検出手法が適当である．そこで以降の節では物体検出手法について述べ，IAR に用いるのに適切な手法について考察を行う．

図 2.1: UAV から撮影された空撮画像

2.2　一般物体検出

一般物体検出とは図 2.2 に示すように画像内からいくつもの物体の領域を検出し，図 2.3 に示すようにそれぞれの領域に含まれる物体が何であるかを分類するタスクである．

以降では一般物体検出において代表的な手法を紹介する．

2.2.1　R-CNN

Regions with CNN features(R-CNN) [22] は畳み込みニューラルネットワーク (CNN) を特徴抽出器として利用した手法である．この手法では Selective Search を用いて物体が写っている領域の候補を約 2000 個抽出し，それらを CNN に入力して分類することで矩形領域にクラスラベルを付与している．

図 2.2: 一般物体検出の例 (入力)

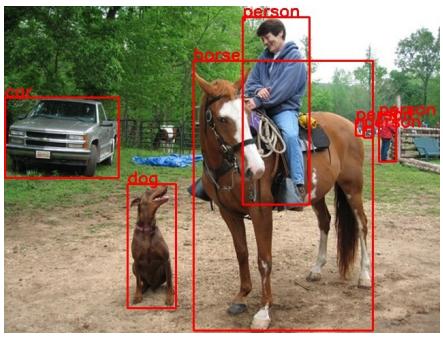

図 2.3: 一般物体検出の例 (出力)

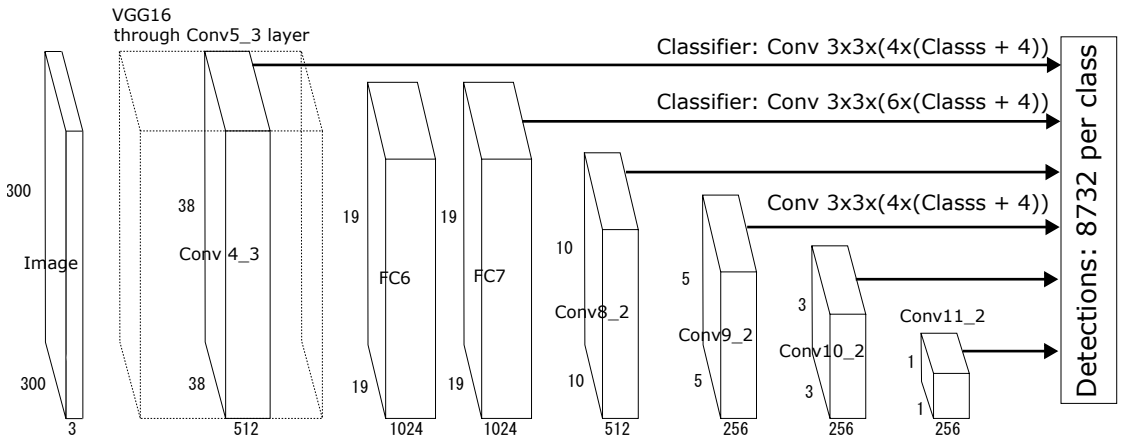

図 2.4: SSD のネットワーク構造

ここで用いている Selective Search とは Segmentation as selective search for object detection [23] で提案された，画像から物体らしい領域を抽出する手法である．[23] の手法は画像に対して，物体毎にセグメンテーションを行い物体領域を抽出する．この手法において画像を塗り分ける際には隣接領域の類似度を比較し統合するということを繰り返すことで物体の領域を確定させ，最後に物体としてラベル付けされた分割領域が収まる矩形を Bounding Box として出力する．Selective Search はそれまで行われていた，画像内で矩形を走査させてサブウィンドウとよばれる候補領域を抽出するスライディングウィンドウと比べて高速に動作するという特徴がある．

2.2.2　Single Shot Multibox Detector

R-CNN までに提案された手法は Selective Search を用いて画像上で矩形を走査させて得られたサブウィンドウを学習器に入力し，クラス分類を行うことで物体検出及びクラス分類を行っていたため検出処理に時間がかかってしまうという問題があった．そこで，Single Shot Multibox Detector(SSD) [24] では画像全体を学習器に入力し，それに対して物体のアンカーボックスを推定しそれらに対してクラス分類を行うことで高速な物体検出器を実現している．

SSD は VGG-16 [25] をベースとしており，図 2.4 のような VGG-16 から全結合層を取り除き Extra Feature Layers を追加したネットワークとなっている．この手法のうち入力画像の解像度が 512 × 512 のネットワーク SSD512 を用いて PASCAL VOC2007 [26] に対して精度評価を行ったところ mAP が 81.6% となり，競合の Fast R-CNN [27] や Faster R-CNN [28] と比べても高い精度を示している．

2.2.3　YOLO v5

YOLO v5 [17] は Ultralytics 社が 2020 年に提案した物体検出手法であり，YOLO [14] をベースに設計されている．YOLO が提案される以前に提案された物体検出手法である R-CNN や SSD では物体の候補領域とそれらのクラス分類は 2 ステップで行うのが一般的であった．それに対し，YOLO は事前に画像をグリッドで分割し，領域毎に物体のクラスや候補領域を求める．それにより以前に提案された Fast R-CNN [27] に比べて精度は劣るものの，速度は大きく向上している．しかし，YOLO にはグリッドサイズより小さな物体に対する検出率が低いという欠点がある．

YOLO v5 が提案されるまでに YOLO v2，YOLO v3 といった改良が行われており [15, 16]，検出対象の候補領域の位置の精度や入力画像のサイズに対するロバスト性，小物体に対する精度が向上してきた．

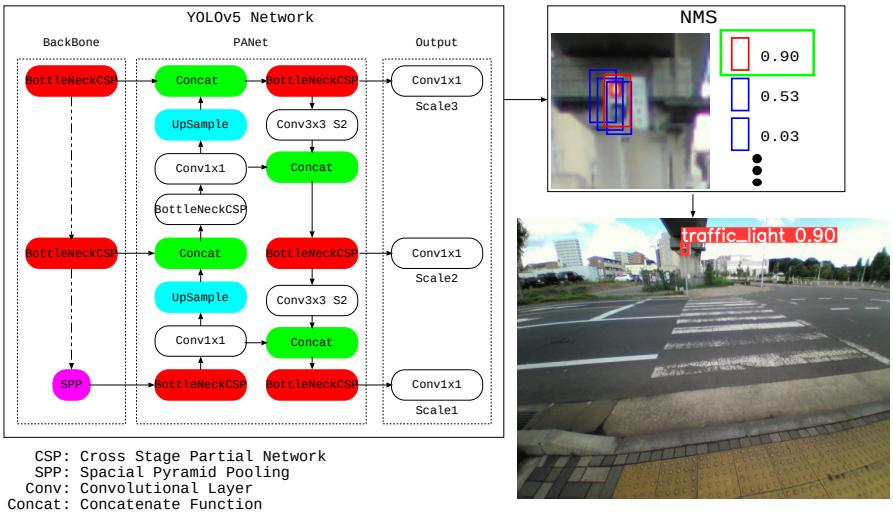

図 2.5: YOLO v5 のネットワーク

図 2.5 に YOLO v5 のネットワークと後処理を示す．YOLO v5 のネットワークは入力された画像全体に対して，アンカーと呼ばれる局所領域を多数設定し，それらに対してクラスラベルとスコアの割り当てを行う．後処理は Non-Maximum Suppression(NMS) と呼ばれ，アンカーの位置情報とスコアを基に，複数のアンカーを適切に統合することにより，矩形の割り当てを行う．

2.3　特定物体検出

複数クラスの物体を画像から抽出し領域にラベルを付与する一般物体検出に対し，特定物体検出はある特定の物体を画像から検出するタスクである．

2.3.1　Informed-Filters

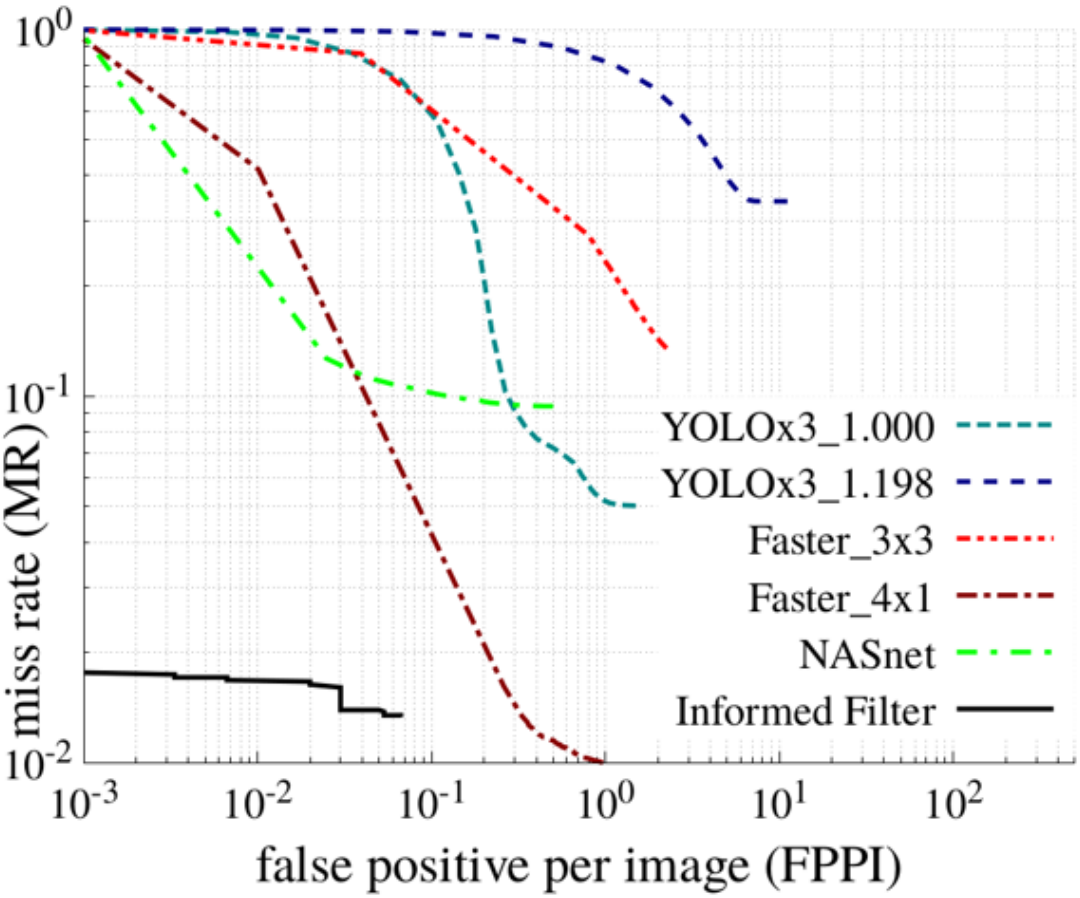

図 2.6: 深層学習を用いた物体検出手法と Informed-Filters との比較．縦軸に見逃し率，横軸に誤検出率を検出処理の閾値を変化させてプロットしたもので，グラフの中で左下にある手法ほど精度がよい手法であることを示している．

Informed-Filters [1] は Haar-Like 特徴 [29] と類似の矩形特徴を用いて検出対象の訓練サンプルの形状に適するように設計された特徴量の計算を行い，算出されたスコアに基づいて検出対象が含まれるか否かの 2 値分類を行う手法である．この手法は古典的な機械学習に基いているが，図 2.6 に示すように深層学習に基づく物体検出精度と比べても検出精度が非常に高いことがわかる．

この手法を検出器に用いる際は一般に，図 2.7 に示すようなスライディングウィンドウにより入力画像からサブウィンドウと呼ばれる単位でサンプル画像を取得し，それらを入力したときの Informed-Filters の識別器の出力によって 2 値分類を行う．

図 2.7: スライディング ウィンドウ

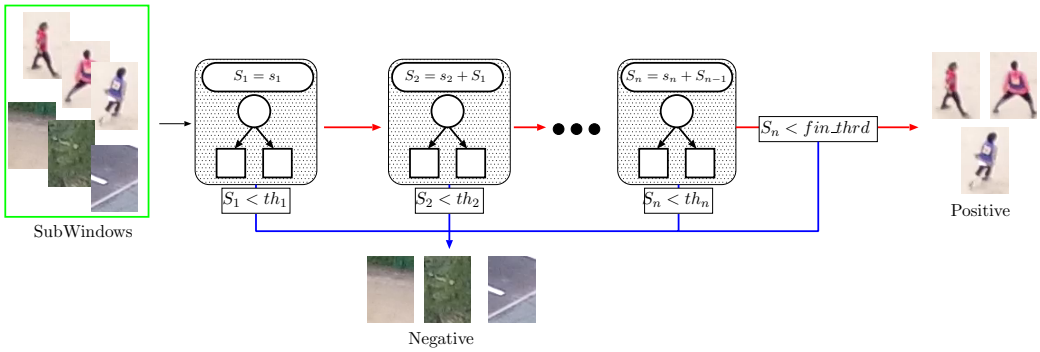

図 2.8: ソフトカスケード構造

入力サンプルのスコアを計算する識別器は図 2.8 に示すようなソフトカスケード構造をとっている．先行研究 [3] では 200 個の深さ 1 の弱識別器でソフトカスケードを構成することにより，実用的な精度と速度を実現している．弱識別器は一般に図 2.21 に示すような決定木の構造をもち，ルートノードから入ったサンプルは弱識別器の深さの数だけ分岐を通り最終的に葉ノードに到達する．

弱識別器の各分岐ノードには特徴テンプレートが割り当てられている．特徴テンプレートは分岐ノードにおいて弱識別器に入力されたサンプルの特徴量計算に用いられるが，これらは検出対象の統計形状を基に人手で設計される．特徴テンプレートの設計は以下のような手順で行われる．

(1)　エッジマップの作成

(2)　エッジマップをセルに分割

(3)　セルにラベルを付与

(4)　部分画像の切り出し

以降ではこれらの手順について説明する．

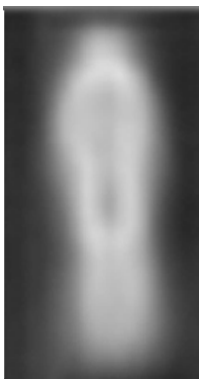

図 2.9: エッジマップ

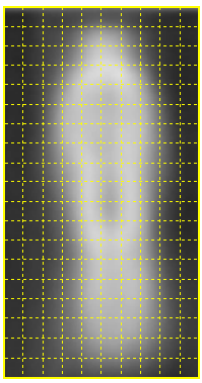

図 2.10: セル分割

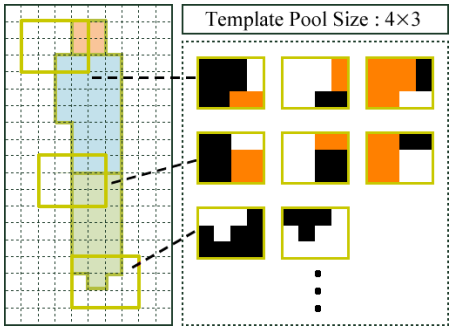

図 2.11: 特徴プール

2.3.1.1　エッジマップの作成

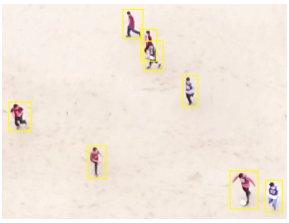

図 2.12: 訓練データの部分画像とそのアノテーション

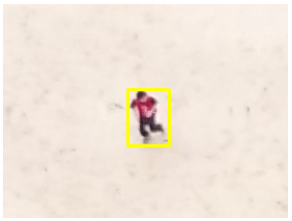

図 2.13: 正解矩形

図 2.14: 正規化した正解矩形

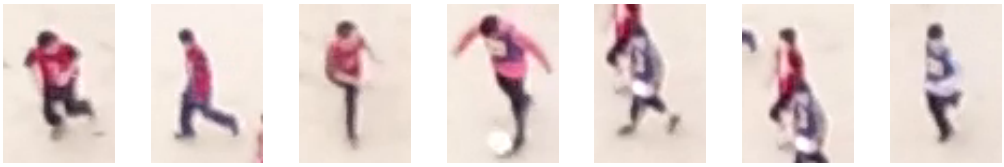

図 2.15: 切り出された画像

エッジマップは図 2.9 に示すような検出対象の統計的形状を表す画像である．以降ではこのエッジマップの作成方法について説明する．

まず，検出対象が含まれる画像を収集しそれらの大きさの正規化を行う．それらのサンプルは主に図 2.12 のような検出器の訓練データの入力画像とその正解座標を基に行われるが，正解矩形は人手で作成されているため，その全ての縦横比が同一であるとは限らない．そこで本稿の 4 章および 5 章で用いる特徴テンプレートの設計においては，図 2.13 および図 2.14 のように正解矩形の中心座標を基に検出器の探索窓と同一の縦横比かつ正解矩形が収まる最小の矩形でサンプルの切り出しを行っている．このように切り出されたサンプルの縦横比は全て同一であるため，最後に識別器に入力される大きさにスケールすることで同一の大きさ，形状のサンプルが得られる．

図 2.16: エッジ画像

図 2.17: 平均画像

さらに，図 2.15 のように上記の方法で切り出されたサンプルそれぞれに対してエッジ検出を行うことで，サンプルに含まれる検出対象の形状情報のみを切り出す．ここで，エッジ検出に Canny Edge Detectior [30] を用いる．すると，図 2.16 のようなエッジ画像が得られる．最後に各サンプル画像から得られたエッジ画像を用いて平均画像を作成することで，図 2.17 のようなエッジマップが得られる．

2.3.1.2　エッジマップの分割

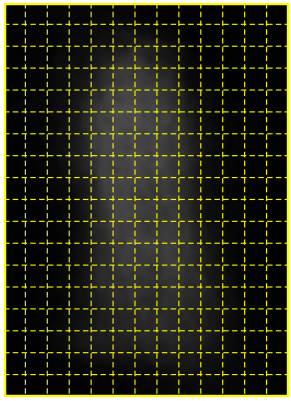

図 2.18: セル単位に分割されたエッジマップ

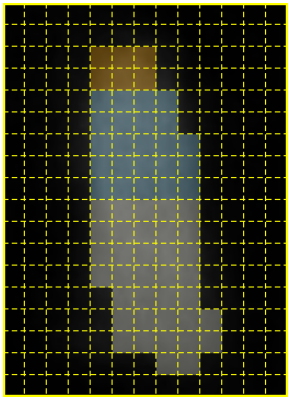

図 2.19: ラベルが付与されたエッジマップ

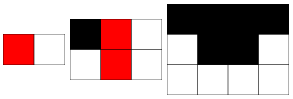

図 2.20: 生成された特徴テンプレート．ラベル: □ 0，■ -1，■ +1

Informed-Filters は計算量の削減のため，画像へのアクセスをセル単位で行う．そのため，特徴量計算に用いられる特徴テンプレートもセル単位のラベルで構成されている．特徴テンプレートはエッジマップにラベル付けしたものを切り出して作成されるため，特徴テンプレートもセル単位に

分割しラベル付けを行う必要がある．そこでラベルの付与を行う前に図 2.15 に示すように，エッジマップをセル単位に分割する．

2.3.1.3　ラベルの付与

Zhang らによって考案された Informed-Filters [1] では，検出対象である人物の統計的形状に関する分析を行っている．この分析の結果，Zhang らは人体の形状は頭，身体，脚の 3 つに大きく分かれると判断している．また，エッジマップにはターゲットの他に背景の情報も含まれるため，ラベルは頭，身体，脚，背景の 4 種類となる．これに基いてエッジマップに対してラベル付けを行うと，図 2.16 のような結果が得られる．

2.3.1.4　部分画像の切り出し

ラベル画像を 1×2 から 4×3 までの大きさで切り出すことで特徴テンプレートが生成される．このとき，テンプレートは図 2.17 に示すように 3 つのラベルで構成されるように生成される．

特徴テンプレートはセルと呼ばれる 4×4 ピクセルの単位で構成される矩形であり，これらは 1×2 から 4×3 までの大きさであらかじめ人手で設計されている．テンプレート内のセルには $+1$, 0, -1 の 3 種類のラベルが付けられている．この特徴テンプレートを図 2.21 のようにサンプル内に配置し， 式 2.1 および式 2.2 のように計算することでサンプルの特徴量を計算している．

$$feature\_value = \sum_{i=1}^{n} label_i \times cell_i \tag{2.1}$$

$$cell_i = \sum_{j=1}^{16} pixel_j \tag{2.2}$$

ここで，$label_i$ はセルに与えられたラベル，$cell_i$ はセル内の画素値の合計，$pixel_j$ はセル内のあるピクセルの画素値をそれぞれ表す．

弱識別器の葉ノードには識別器の学習で得られた，弱識別器が出力する値の候補値がそれぞれ割り当てられている．弱識別器の分岐ノードは特徴テンプレートの他に閾値を保持しており，割り当てられた特徴テンプレートを用いて算出された特徴量と閾値の大小関係により決定木の分岐方向が決定される．そして，最終的に辿り着いた葉ノードに割り当てられている値が入力サンプルに対する弱識別器の出力値となる．

2.3.2　CUDA を用いた Informed-Filters の並列化

これまで述べてきた結果から，IAR に適切な手法は Informed-Filters であるといえる．しかし，IAR においては物体検出手法を用いてセンサの位置を推定してからネットワークを構築するまでにノードの位置が変化しないことが必要である．そのため，IAR に用いる物体検出手法はリアルタイムに動作する必要がある．そこで先行研究 [3] では Informed-Filters を Image-Assisted Routing に適用するため，以下の工夫を行っている．

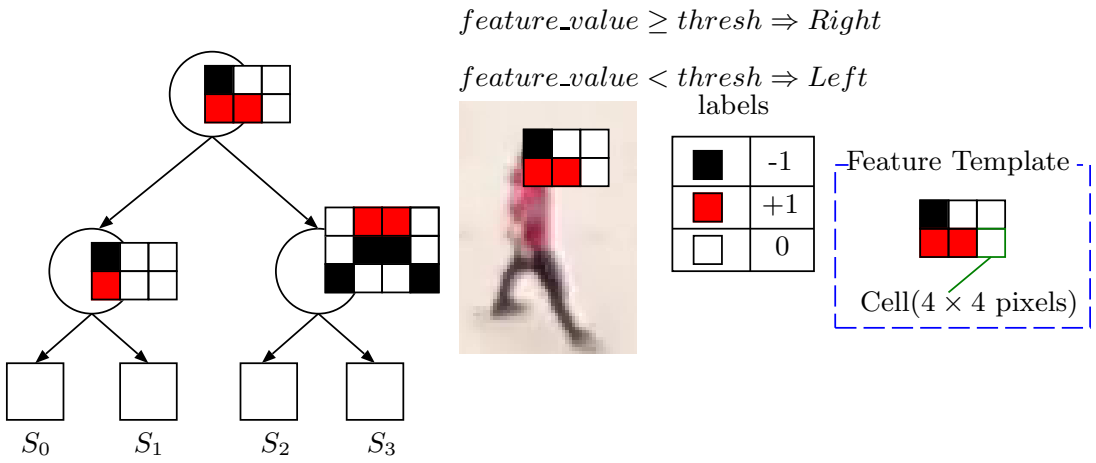

図 2.21: 深さ 2 の弱識別器

- メモリへのアクセス回数の削減
- CUDA を用いた並列化

以降ではこれらについて述べる．

2.3.2.1　メモリへのアクセス回数の削減

先行研究 [3] ではメモリのアクセス回数削減のために

1. 積分画像の使用
2. 特徴の最小矩形分割

を行っている．

Informed-Filters の特徴量計算は式 2.1 および式 2.2 の式に基いて行われるが，その際特徴テンプレートの各セルに割り当てられたラベル領域内の画素値の総和を算出する必要がある．画像にアクセスする度に画素 1 つ 1 つを参照し，それらの総和を計算するとメモリアクセスと計算量が膨大になってしまう．そこで，先行研究 [3] では積分画像 [31] を用いている．積分画像とはそれぞれの画素と左上 (0,0) を対角の頂点とする長方形の画素値の総和が格納された画像のことある．積分画像を用いることで画像内の任意の矩形領域内の画素値の総和は，図 2.22 のように矩形領域の右下と左上の画素値の和から右上と左下の画素値の和を引くだけで求まる．これにより通常 16 回のメモリアクセスと 15 回の演算が必要だったセル内の画素値の総和を，4 回のメモリアクセスと 3 回の演算で求めることができる．

積分画像を用いたとしても，最大サイズの特徴テンプレートの画素値の総和の計算には $4 \times 3 \times 4 = 48$ 回のメモリアクセスが発生する．しかし，特徴量計算には同一ラベルが付与されたセルの画素値の総和があれば十分である．そこで先行研究 [3] では特徴量計算の際に，特徴量テンプレートを図 2.23 のような長方形の集合として扱う．特徴量テンプレートを，複合長方形を最小の個数で分

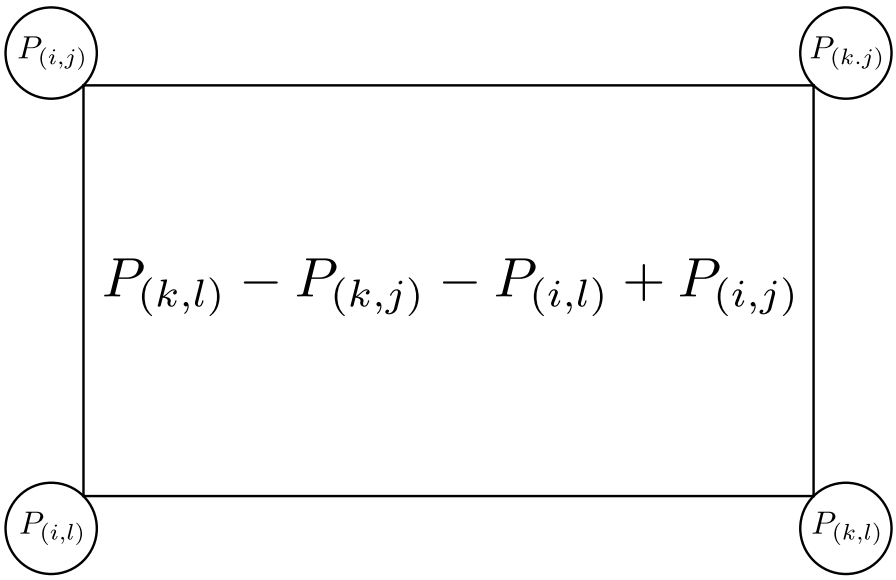

図 2.22: 積分画像の計算

割する手法 [32] を用いてラベル毎に分割すると，黒いラベルの矩形が 1 つ，白いラベルの矩形が 2 つ，赤いラベルの矩形が 1 つの，4 つの矩形からなる領域として再構成でき，$4 \times 4 = 16$ 回のメモリアクセスだけで同一ラベルが付与されたセルの画素値の総和を求めることができる．

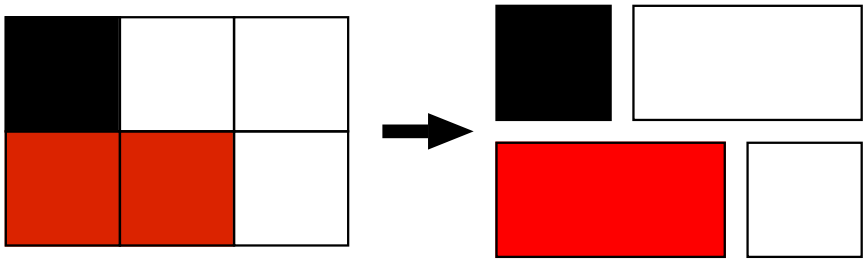

図 2.23: 最小分割矩形

2.3.2.2　CUDA を用いた並列化

CUDA を用いて処理の並列化を行うにあたり，GPU 内部のプロセッサ数や実行時に発生するワープダイバージェンスと呼ばれるオーバーヘッドについて考慮する必要がある．GPU 内部のスレッドは図 2.24 のようにグリッドの中にブロックがあり，さらにその中にスレッドが存在するという構造となっている．最適なブロックの分割数は計算に用いる GPU のプロセッサの個数に依存し，それに伴って適切なグリッドの分割数も変化する．スレッドは 32 スレッド毎にワープという単位で実行され，ワープ内のスレッド間で分岐処理の方向に相違があるとワープダイバージェンス

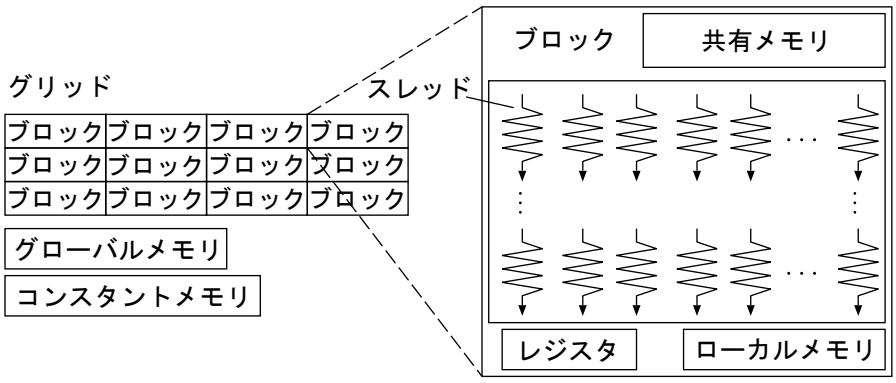

図 2.24: CUDA ライブラリにおける並列プログラミングモデル

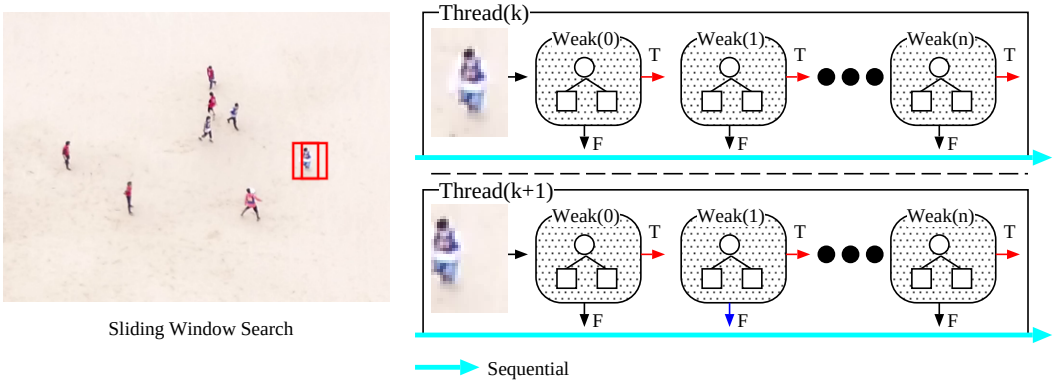

図 2.25: サブウィンドウ毎の並列化

が発生する．

Informed-Filters の並列化においては，図 2.25 および図 2.26 に示すようなスレッド毎にサブウィンドウを割り当てるサブウィンドウ毎の並列化と，スレッド毎に弱識別器を割り当てサブウィンドウ毎に特徴量計算を並列化する弱識別器毎の並列化の 2 種類が考えられる．図 2.27 は，Informed-Filters による検出を行った場合に各弱識別器が棄却するサンプル数を示している．この結果から，多くのサンプルが早い段階で棄却されることが分かる．前者の実装を行う場合，それぞれの入力サンプルすなわちサブウィンドウが棄却されるまでに通過する弱識別器の段数が同一であるかどうかは分からず，もし同一でない場合はワープダイバージェンスが発生する．しかし，図 2.27 の結果から，カスケードの後段まで処理が進むことは稀であり，ワープダイバージェンスによる影響を考慮しても早期棄却の効果が期待できると言える．一方，弱識別器毎の並列化を行うと，それぞれの識別器が演算に用いる特徴の違いに起因するワープダイバージェンスの発生に加えて，早期棄却を行う際に無駄な計算を行う可能性が高くなる．そのため先行研究 [3] ではサブウィンドウ毎の並列化を行っている．

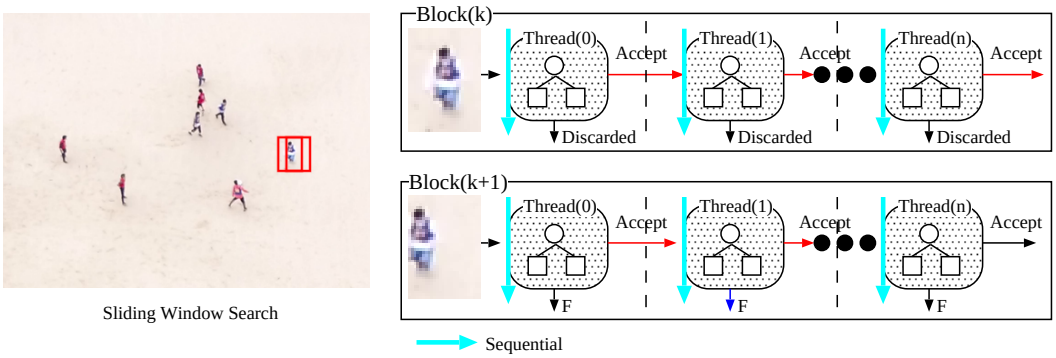

図 2.26: 弱識別器毎の並列化

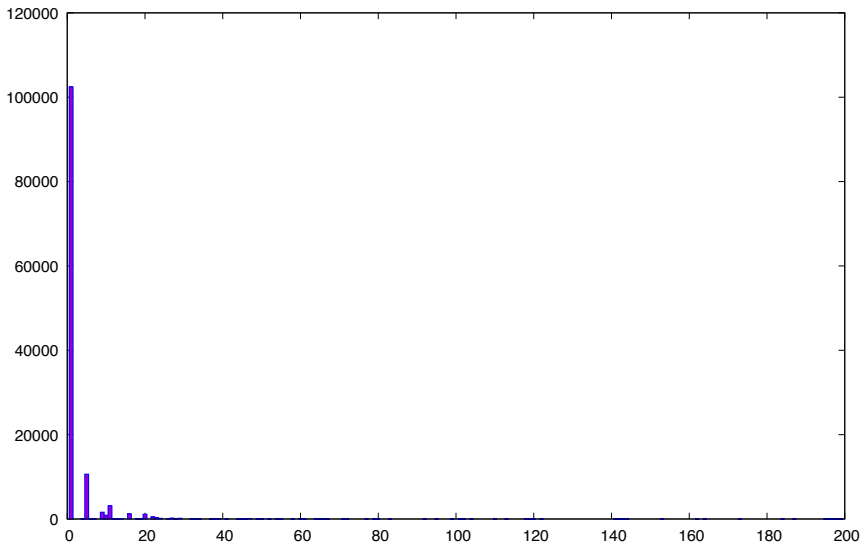

図 2.27: 各弱識別器における棄却サンプル数

第3章　連続的な情報を扱うネットワークの登場

本稿では模倣学習手法のひとつである，Neural Programmer-Interpreters (NPI) を用いたマシンコード生成手法の提案を行う．そこで本章では模倣学習の基となっている情報を扱うネットワークの登場やそれに関連する手法および NPI について述べる．

3.1　はじめに

古典的な機械学習では分類問題や回帰問題を解く手法が主に提案されてきた．一方で計算機の性能向上により，時系列情報を扱うことのできる深層学習のネットワーク構造が提案されてきた [33,34]．

中でも Recurrent Neural Network(RNN) は時系列情報に代表される，前後で依存関係のあるデータに対する学習ができるネットワークレイヤーとして初期に提案されている．この手法では図 3.1 に示すように，現在の入力に加えて時系列的にひとつ前の入力に対する出力を用いることによって，人物の行動分類 [35]，自然言語処理 [36] といったことが可能になる．

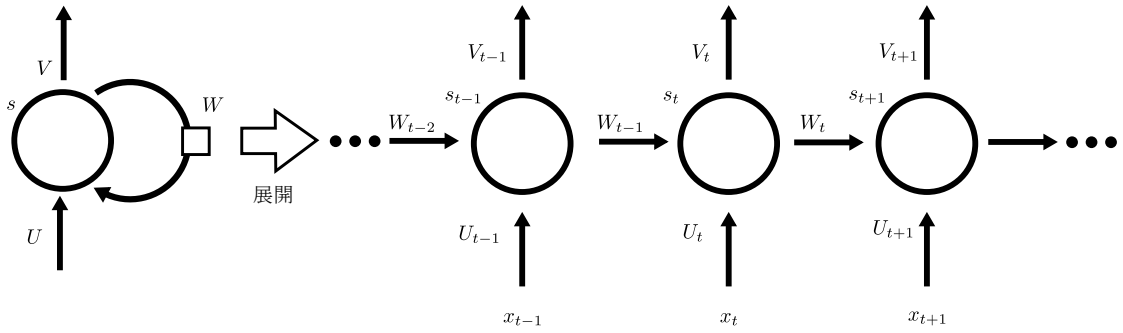

図 3.1: Recurrent Neural Network

以降では連続した情報を入力としたタスクとして，自然言語生成と模倣学習について述べる．

3.2　自然言語生成

近年，自然言語処理に関する論文で提案された Transformer [37] の登場により，自然言語生成に関する手法が多く提案されている [38,39]．中でも GPT-3 [40] は OpenAI と Microsoft が提案している手法で，人間が文章で情報を与えるとその情報を使って完全な文章の生成を行う．また，

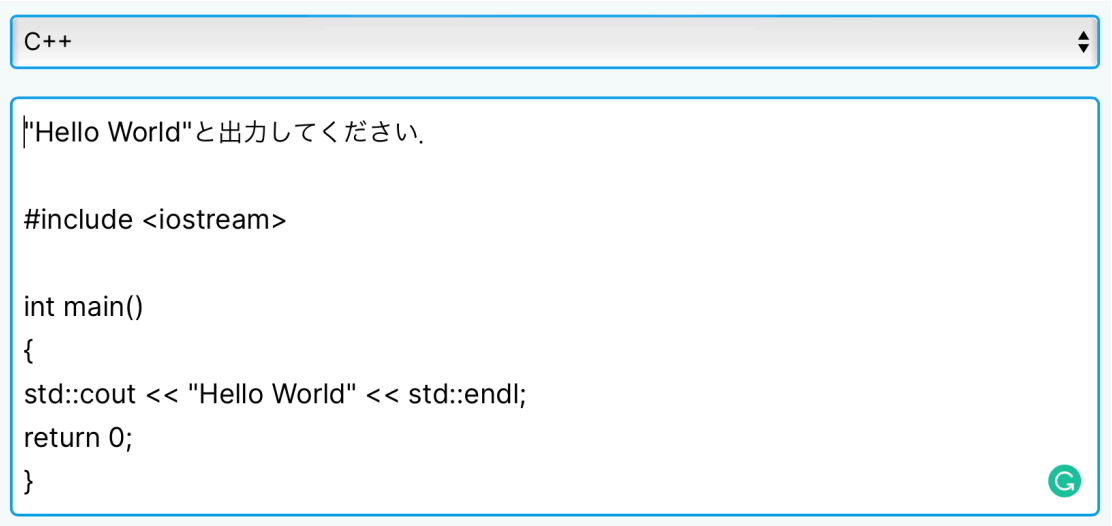

図 3.2: AI Programmer に対してリクエストを行ったときの様子

GPT-3 は文章だけでなくプログラムの生成を行うこともできる．図 3.2 は GPT-3 を用いてプログラムの生成を行うサービスである AI Programmer [41] に対してリクエストを投げたときの様子である．このように CLI で完結する単純なプログラムであれば，深層学習を用いて簡単に生成を行うことができる．ただし，図 3.3 に失敗例を示すように少し複雑なリクエストには対応できない．

GPT-3 を応用することでプログラムの生成だけでなく入力されたプログラムの最適化を行うことができると考えられるが，GPT-3 の学習には膨大なコストがかかり，NVIDIA V100 355 台を用いても 1 年かかる [42, 43].

3.3　模倣学習

RNN に代表される前後関係のデータに対する推論を得意とするネットワークレイヤーを用いることで，時系列に基づく推論や動画を用いた入力とする推論が可能となっている [44, 45]．このような連続的な入出力を行う手法が提案される中で模倣学習という訓練手法が提案されている．模倣学習の手法としては繰り返しコピーや単純な優先ソート，連想想起が可能なニューラルチューリングマシン [46] 入力された可変入出力サイズに対応した Pointer Networks [47] が代表的である．

3.3.1　Neural Turing Machine

Neural Turing Machine(NTM) は アラン・チューリングが「計算可能性」に関する議論のために提唱した計算機を数学的に表現したモデルである Turing Machine をニューラルネットワークを用いて実現した手法である．この手法が提案されるまでの模倣学習の手法ではタスクにおいて必要な「状態」を学習レイヤーの内部で保持するものが一般的であった．しかし，そのような手法には

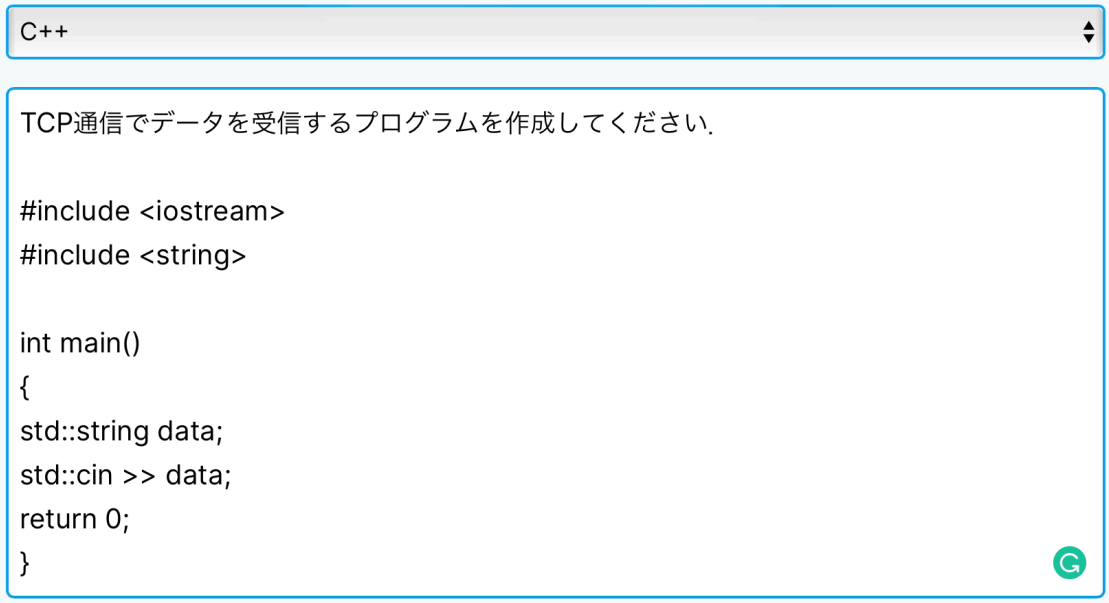

図 3.3: AI Programmer に対してリクエストを行ったときの様子 (失敗例)

保持できる状態の種類が学習レイヤーのパラメータ数に依存してしまう，あまり長いステップ数かかるタスクには適用できないという問題がある．

NTM ではこのような問題を解決するため，タスクの途中状態をニューラルネットワークの外部にある領域を利用する．これにより学習レイヤーの内部パラメータ数に依存しない状態の記憶が可能となる．また，タスクにおける入出力を抽象化し学習器に入力されるデータや出力するデータを限定することで，学習器が訓練する内容を少なくすると共に，実行に多くのステップを要するタスクやさまざまなタスクに対応することができる．

3.3.2　Pointer Networks

NTM では外部記憶装置を用いたアーキテクチャや，入出力の抽象化により実行にかかるステップ数やさまざまなタスクに対応することができているが，タスクの各ステップでの出力クラス数が入力の長さに依存しているため，さまざまな長さのソーティングや組合せ最適化問題といった可変長の出力を扱うタスクに適用するのが難しいという問題がある．

Pointer Networks では Sequence-to-Sequence で提案された Encoder-Decoder モデルを Neural machine translation [48] で提案された attention を用いて拡張することで可変長の出力を可能としている．図 3.4 および，図 3.5 はそれぞれ Sequence-to-Sequence [49] と Pointer Networks で凸包問題を問いている様子を示している．Sequence-to-Sequence では推論時に訓練時と同数かそれ以下の入出力数を行うことしかできないという問題がある．そこで，Pointer Networks の各ステップ Decoder は入力の長さと同じ辞書サイズを持つソフトマックス分布を出力することで，可変長の出力を実現している．

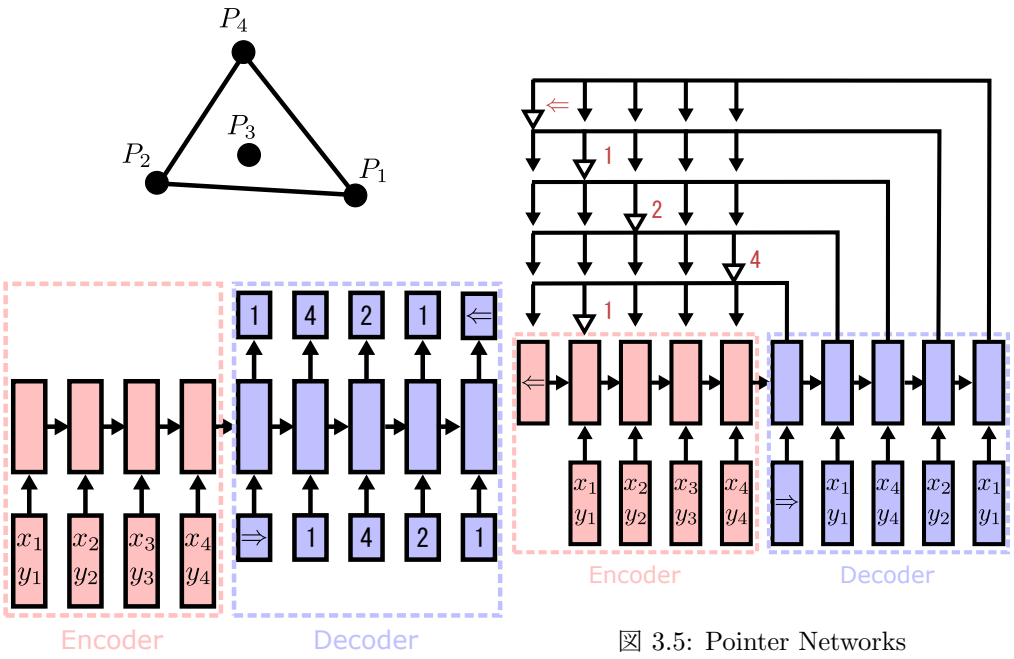

図 3.4: Sequence-to-Sequence

図 3.5: Pointer Networks

3.4　Neural Programmer-Interpreters

Neural Programmer-Interpreters (NPI) [50, 51] NPI は 2015 年に提案された模倣学習を行う手法であり，学習器はタスクの各ステップをサブプログラムと呼ばれる単位で出力することが特徴である．この手法においては学習器には”Scratch-Pad”と呼ばれるタスクの途中過程を記録しておく外部メモリが与えられる．学習器は Scratch-Pad からの出力と実行中のサブプログラム，引数が与えられると，次に実行するべきサブプログラムとその引数を出力する．

学習器が出力するプログラムを構成するサブプログラムには即時サブプログラムとよばれるものがあり，これは Scratch-Pad の内部状態を更新するものである．

以降では NPI の学習器のアーキテクチャ，Scratch-Pad，推論および訓練について述べる．

3.4.1　Scratch-Pad

NPI において Scratch-Pad は学習器の外部メモリのような役割を果たす．タスク毎に図 3.6，図 3.7 および図 3.8 といった異なるスクラッチパッドを用意することで，学習器にさまざまなタスクを実行させることができる．先行研究 [50] では以下のタスクを学習器に訓練している．

1. 加算の筆算: 人間が加算の筆算を解く要領で加算を解くタスク
2. バブルソート: バブルソートを行うタスク
3. 3D モデルの回転: 3D 空間上のカメラをあらかじめ定められた位置に移動するタスク

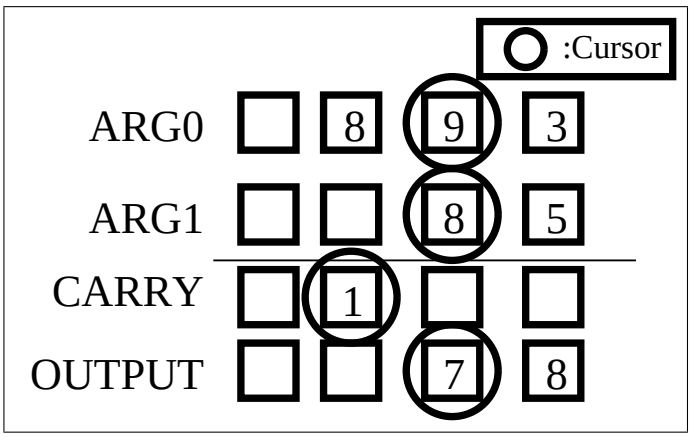

図 3.6: 加算のタスクに用いられる Scratch-Pad

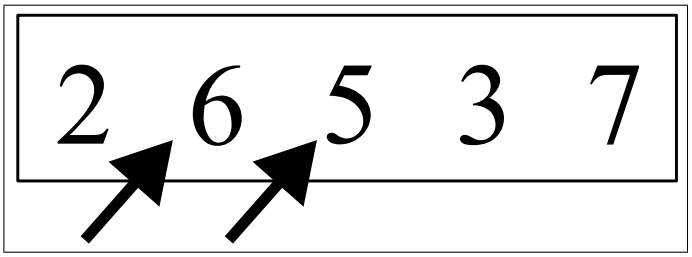

図 3.7: ソーティングのタスクに用いられる Scratch-Pad

加算の筆算においては図 3.6 のような Scratch-Pad が与えられ，この Scratch-Pad は人間が実際に筆算を解く際の紙のような役割を果たす．ソーティングにおいては図 3.7 のような Scratch-Pad が与えられ，NPI の学習器や訓練データ生成器は矢印の移動および数字の入れ替えを行うことで，Scratch-Pad 内の数字を順番に並べ替える．3D モデルの回転においては図 3.8 のような Scratch-Pad が与えられる．Scratch-Pad の内部にはカメラの位置情報があり，学習器やデータ生成器はカメラを 15 度刻みでモデルを中心とする周回軌道上をカメラの画角を基準とした上下中央に動かすことによって，カメラ座標をあらかじめ与えられた位置に修正する．

学習器の訓練時のコストを抑制するため以下のような工夫がされている．

1. Scratch-Pad が出力するのは内部に保持している情報の一部である．例えば，NPI を用いて行われるタスクのうちのひとつである加算の筆算では図 3.6 のような Scratch-Pad が用いられるが，この中で出力されるのはカーソルの位置の数字のみである．

2. Scratch-Pad は即時サブプログラムからのみ更新される．即時サブプログラムを繰り返し呼ぶことによって，NPI の学習器は Scratch-Pad を更新し，タスクを解く．

またこれにより，タスク毎にネットワークの構造を変化させることなく学習器にさまざまなタスクを訓練することができるという利点もある．

図 3.8: 3D モデルの回転タスクに用いられる Scratch-Pad

3.4.2　学習器の構造

NPI の学習器は大きく分けてシーケンシャルモデル，キー値メモリの 2 つのレイヤーで構成されている．シーケンシャルモデルは時系列情報を扱うことが可能な Long Short-Term Memory(LSTM) [34] をベースとしており，学習器がこれまでに出力したサブプログラムや引数を基に次の出力を決定することを可能にしている．キー値メモリはタスクの要所で必要な入出力関係を学習する．例えば，[50] で実装されている加算の筆算では一桁の加算の桁上がりと出力を学習する．

図 3.9 に NPI の学習器のネットワーク図を示す．このネットワークにおけるキー値メモリの出力を用いてシーケンシャルモデルが動的にサブプログラムを出力するという構造は [52] から着想を得ている．これにより，スクラッチパッドからの観測情報の変化をより小さくしてシーケンシャルモデルに入力することができる．例えば一桁の加算の場合，スクラッチパッドから出力される観測値は引数 2 つ，桁上がり，そして出力の値である．引数 2 つと出力の値はそれぞれ空白と 0 から 9 の 11 通り，桁上がりは空白か 1 の 2 通りであり．これらの組み合わせは単純計算で 2662 通りである．一方で加算のタスクにおいてキー値メモリから出力される値は引数からの観測値，桁上がりを合計した結果の 1 の位と 10 の位でその組み合わせは 20 通りである．これにより，シーケンシャルモデルが学習する必要のある入力は 100 分の 1 以下となっている．

以降では NPI の学習器を構成する 2 つのネットワーク層について説明する．

3.4.2.1　キー値メモリ

キー値メモリは上で述べた通りシーケンシャルモデルへの入力の変化を小さくために用いられる．このようなレイヤーは [53–55] で用いられている．

NPI において，キー値メモリは学習器に訓練するタスクによって異なった入出力を学習する．表 3.1 に例を示す．

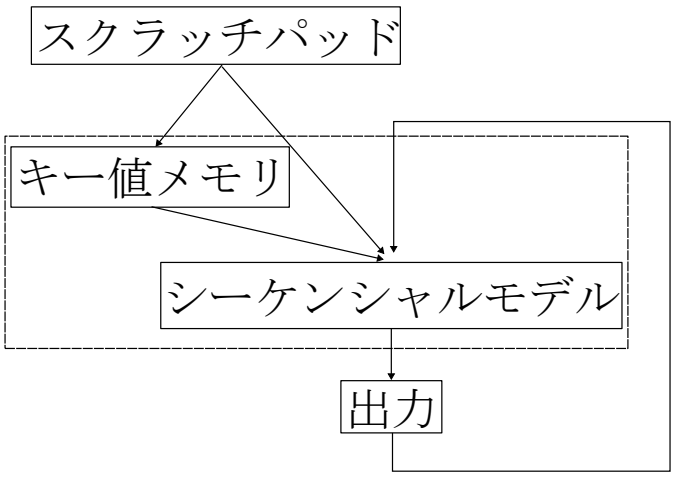

図 3.9: Neural Programmer-Interpreters のモデル図．点線の枠内が NPI の学習モデルである．

表 3.1: キー値メモリの学習例

タスク	キー値メモリの入出力
加算	観測値の合計値及び桁上がり
ソーティング	観測値の大小関係
3D モデルの回転	3D モデルの向きに関する推定情報

3.4.2.2　シーケンシャルモデル

シーケンシャルモデルは呼び出すべきサブプログラムを推論するネットワーク層である．この層ではキー値メモリからの出力，実行中のプログラムの情報および，自身の内部状態を基に出力するサブプログラムおよび，それに与える引数を推論する．

3.3 節で述べたとおり，Recurrent Neural Network を用いることで時系列データを扱うことができる．しかしこの構造には現在の入出力からみて直近のデータは推論に活用できるものの，時系列情報の長さが長くなると，誤差が消失 · 発散する可能性がある．

そこで，シーケンシャルモデルでは Long short-term memory(LSTM) [34] を用いることでこの問題を解決している．以降では LSTM について述べる．

3.4.2.3　Long short-term memory

LSTM では RNN のネットワーク層において誤差が消失 · 発散する問題を解決するため図 3.10 に示すように，ネットワーク層の中に 1 ステップの遅延をもつ自己回帰接続をもつ．この接続がラッチのような役割を果たし，過去のより古い情報を入出力に活かすことができる．図 3.10 中の

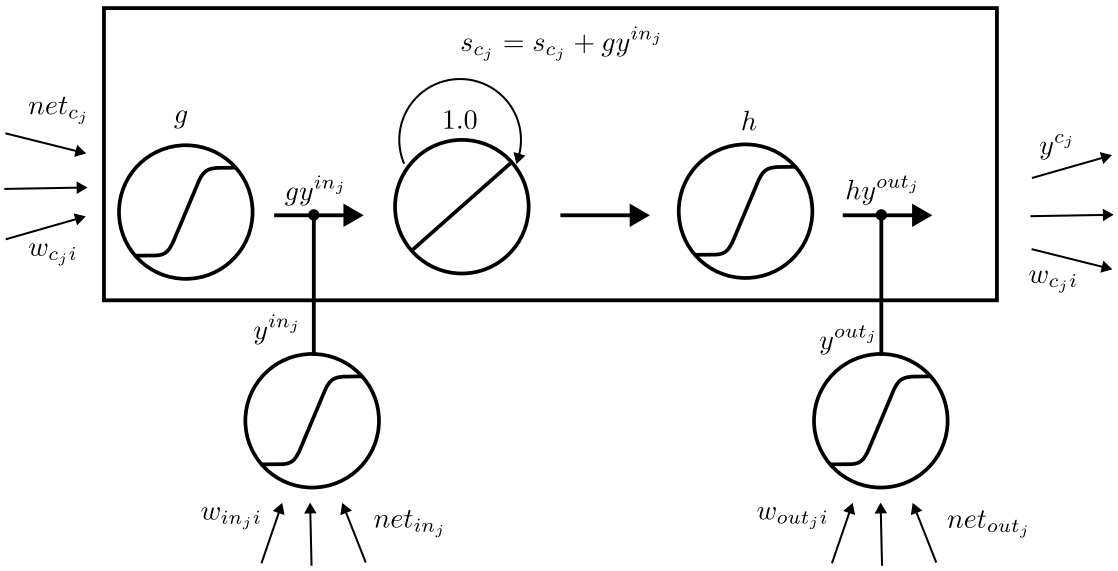

図 3.10: LSTM 層のノード．図中の i はノードが隠れ層の何層目にあるか，c_j は層の中の j 番目のノードを表す．$in_j \cdot out_j$ はそのノードの全 input および output に対する重みを表す外部メモリを表す．

関数はそれぞれ以下の通りになっている．

$$
\begin{aligned}
f(x) &= \frac{1}{1+exp(-x)}. \\
h(x) &= \frac{2}{1+exp(-x)} - 1. \\
g(x) &= \frac{4}{1+exp(-x)} - 2. \\
net_i(t) &= \sum_j w_{ij} y^j (t-1) \\
s_{c_j}(0) &= 0, s_{c_j}(t) = s_{c_j}(t-1) + y^{in_j}(t) g(net_{c_j}(t)) for t > 0. \\
& i : index\ of\ layer,\ j :\ index\ of\ node,\ c_j :\ j_{th}\ node
\end{aligned}
$$

f は 0 から 1，h は-1 から 1，g は-2 から 2 の範囲の出力をもつシグモイド関数，net は 1 ステップ前の Long short-term memory ノードが位置する層の出力を表す関数，s_{c_j} はノード c_j の内部状態を表す関数である．g はこのノードに入ってくる情報の，h は内部的に出力される値の重み付けを行っている．

ここで入力される情報と内部的な出力をそのまま他の層に伝えてしまうと層間で情報の整合性がとれなくなり，学習がうまくいかなくなってしまう．そこで，関数 g および関数 h を用いてノードへの全入力に対する重みを保持している in_j の情報と全出力に対する重みをもっている out_j をもとに，入出力の情報を必要最低限にする．関数 g および関数 h，学習した重み in_j および重み out_j を用いることで時系列のステップ毎に必要な情報を取捨選択することができる．さらにこれは，LSTM のネットワーク層が内部情報を必要になるまで保持することを可能にし，時系列情報

が長くなった際の誤差の消失や重みの発散を防ぐことができる．

しかし，LSTM の内部情報には上記のような利点がある反面，入力の大きな変化に対応できないという欠点もある．[34] では外部から手動で内部情報をリセットすることでこの問題に対応しているがこれは実用的ではない．そこで，[56] では LSTM 層に忘却ゲートを設けることで LSTM 層が適切な場面で内部情報をリセットできるようにしている．

3.4.3　サブプログラム

サブプログラムは NPI の学習器が生成するプログラムを構成する単位である．中でも即時サブプログラムと呼ばれるものは Scratch-Pad の内部状態を更新するために用いられ，こちらの動作についてはあらかじめ用意されて訓練時および推論時に与えられる．学習器は各サブプログラムが即時サブプログラムかどうかと，即時サブプログラム以外のサブプログラムがどのサブプログラムを呼び出すのかということを学習する．

サブプログラムはタスク毎にさまざまな形で用意される．先行研究で実装されているタスクにおけるサブプグラムとしては表 3.2，表 3.3 および表 3.4 といったものがある．

表 3.2: 先行研究の加算のタスクにおけるサブプログラム

サブプログラム	動作
ADD	加算の筆算の開始サブプログラム
ADD1	10 進の半加算を行うサブプログラム
LSHIFT	Scratch-Pad 上の注目座標を全て左に 1 つ動かすサブプログラム
RSHIFT	Scratch-Pad 上の注目座標を全て右に 1 つ動かすサブプログラム
PTR	Scratch-Pad における注目座標を移動する即時サブプログラム
WRITE	注目座標の位置にデータを書き込む即時サブプログラム

表 3.3: 先行研究のバブルソートのタスクにおけるサブプログラム

サブプログラム	動作
BUBBLESORT	バブルソートの開始サブプログラム
BUBBLE	10 進の半加算を行うサブプログラム
RESET	Scratch-Pad 上の全ての注目座標を左端に移動するサブプログラム
BSTEP	Scratch-Pad 上の大小関係を比較 · 入替を行い，注目座標を右に移動するサブプログラム
COMPSWAP	Scratch-Pad 上の注目座標上の数値を大小関係を基に入れ替えるサブプログラム
LSHIFT	Scratch-Pad 上の注目座標を全て左に 1 つ動かすサブプログラム
RSHIFT	Scratch-Pad 上の注目座標を全て右に 1 つ動かすサブプログラム

表 3.4: 先行研究の 3D モデルの回転タスクにおけるサブプログラム

サブプログラム	動作
GOTO	3D モデル回転の開始サブプログラム
HGOTO	横方向のカメラ回転を行うサブプログラム
LGOTO	カメラが正面を向くまで左に回転するサブプログラム
RGOTO	カメラが正面を向くまで右に回転するサブプログラム
VGOTO	縦方向のカメラ回転を行うサブプログラム
UGOTO	カメラが正面を向くまで上に回転するサブプログラム
DGOTO	カメラが正面を向くまで下に回転するサブプログラム
MOVE	Scratch-Pad 上のカメラを回転させるプログラム

3.4.4　推論

本節では NPI のモデルが行う推論について述べる．NPI のモデルではアルゴリズム 1 のようなアルゴリズムに従ってタスクのステップ毎に推論を行う．このアルゴリズムは先行研究 [51] に基づいて，サブプログラムが再帰的に実行されるようになっている．サブプログラムが呼び出されると，Scratch-Pad からの観測情報 e とプログラムに対する引数 $\boldsymbol{a_{in}}$ がキー値メモリに入力として与えらる．キー値メモリはそれに対して自身の内部状態 s を出力し，シーケンシャルモデル $Model$ にはそれに加えて実行中のプログラム p_{in} が与えられる．

各ステップでは表 3.5 のような入力と出力が行われる．以降では先行研究 [50] における加算の筆算のタスクの途中過程で実行される 10 進 1 桁の半加算を行うサブプログラム ADD1 を実行中のモデルの入出力を例にこれらについて説明を行う．そこで，図 3.11，図 3.12，および図 3.13 に NPI のモデルが推論を行っている様子を示す．

タスクの各ステップにおいて，現在実行中のサブプログラム p_{in} とそれに対する引数 $\boldsymbol{a_{in}}$ および Scratch-Pad からの観測情報 e が入力として与えられると，NPI の学習器は出力として現在実行中のサブプログラムを終了するかどうかを表す r，次に実行するサブプログラム p_{out} およびその引数 $\boldsymbol{a_{out}}$ を出力する．図 3.11 の 4 番目のステップと図 3.12 のステップを比べると，学習器への入力 p_{in}，$\boldsymbol{a_{in}}$，および e が同一であるにも関わらず，学習器からは異なる出力が得られている．これは学習器のレイヤーのうちシーケンシャルモデルがもつ時系列情報によるものである．シーケンシャルモデルの時系列情報はサブプログラムの呼び出しの深さや現在の呼び出しにおいて，どんなプログラムを呼び出したかといった情報をもっている．これにより，出力するべき r，p_{out}，$\boldsymbol{a_{out}}$ を適切に推論することができる．

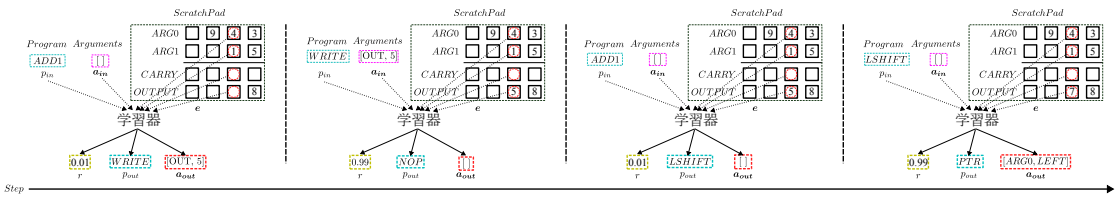

図 3.11: 加算のタスクを訓練したモデルを用いて 1 桁の加算を行っている様子 (1/3).

表 3.5: NPI のモデルに対する入出力

種別	名称	
入力	e	Scratch-Pad からの観測情報
	p_{in}	現在実行中のサブプログラム
	$\boldsymbol{a_{in}}$	プログラムに与えられた引数
出力	r	現在実行中のプログラムを終了するかどうか
	p_{out}	次に実行するプログラム
	$\boldsymbol{a_{out}}$	次に実行するプロウラムに与える引数

アルゴリズム 1 Neural Programmer-Interpreters の動作

Input: e: Scratch-Pad からの観測情報，p_{in}: 実行中のサブプログラム，$\boldsymbol{a_{in}}$: 引数，α: プログラム停止用の閾値，$\boldsymbol{h}$: 学習器の内部状態

function RUN($e, p_{in}, \boldsymbol{a_{in}}$)
  $r \Leftarrow 0$
  while $r < \alpha$ **do**
    $s \Leftarrow KeyValueMemory(e, \boldsymbol{a_{in}})$
    $r, p_{out}, \boldsymbol{a_{out}}, \boldsymbol{h} \Leftarrow Model(s, p_{in}, \boldsymbol{h})$
    if p_{in} が即時プログラム **then**
      $e \leftarrow Env(e, p_{in}, a_{in})$
    else
      $Run(e, p_{out}, a_{in})$
    end if
  end while
end function

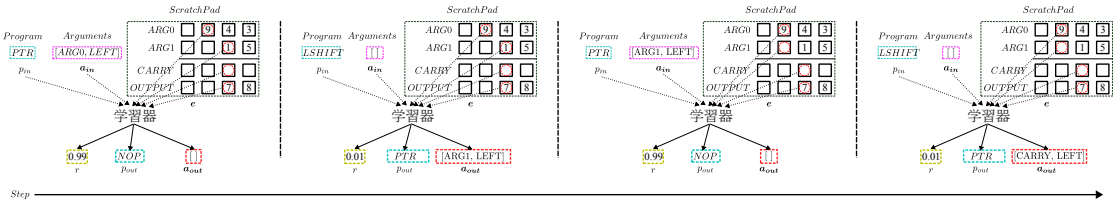

図 3.12: 加算のタスクを訓練したモデルを用いて 1 桁の加算を行っている様子 (2/3).

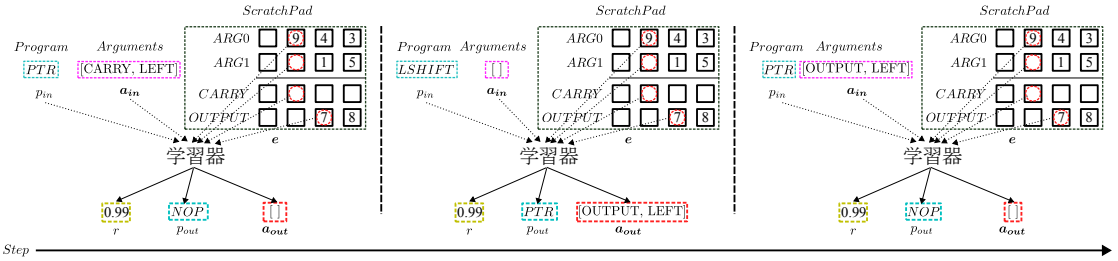

図 3.13: 加算のタスクを訓練したモデルを用いて 1 桁の加算を行っている様子 (3/3).

3.4.5　訓練

本節では NPI の訓練手法について述べる．

3.4.2 で述べたように，NPI の学習器はシーケンシャルモデルおよびキー値メモリの 2 つのレイヤーから構成されているが，シーケンシャルモデルの推論にはキー値メモリの出力が用いられる．そこで，これら 2 つのレイヤーのうちキー値メモリの訓練が行われ，最後にシーケンシャルモデルの訓練が行われる．

キー値メモリは Scratch-Pad からの観測情報をエンコードすることで，シーケンシャルモデルの訓練コストを抑制する．そこで，この訓練では図 3.14 に示すようにキー値メモリの出力に Dense 層と Activation 層を加え，その出力がタスクに必要なものになるようにする．例えば，加算のタスクでは 1 桁の加算の結果，ソーティングのタスクでは数値の大小関係といった具合である．キー値メモリの訓練が終わるとその重みは固定され，シーケンシャルモデルの訓練が行われる．

シーケンシャルモデルは時系列情報を扱うことができるモデルのため，入力と正解データを時系列順に与えなくてはならない．そこで，シーケンシャルモデルは実行中のサブプログラムおよびその引数，Scratch-Pad からの観測情報を入力し，その出力と新しい観測情報を入力するということと時系列順を繰り返すことで学習を行う．またその出力値の誤差は学習器が出力を行う毎に加算され，出力が終わった後に一度に逆伝搬される．また，適切に訓練されたモデルは図中の破線で示された入力と過去に自身が行った出力を基に推論を行う．

シーケンシャルモデルに入力される情報はタスクのステップ毎に次々変化していくため，人手で 1 ステップずつ生成するのは困難である．そこで NPI では訓練データ生成器を用いてタスクのステップ毎の観測情報とそれに対する理想的な出力を作成する．

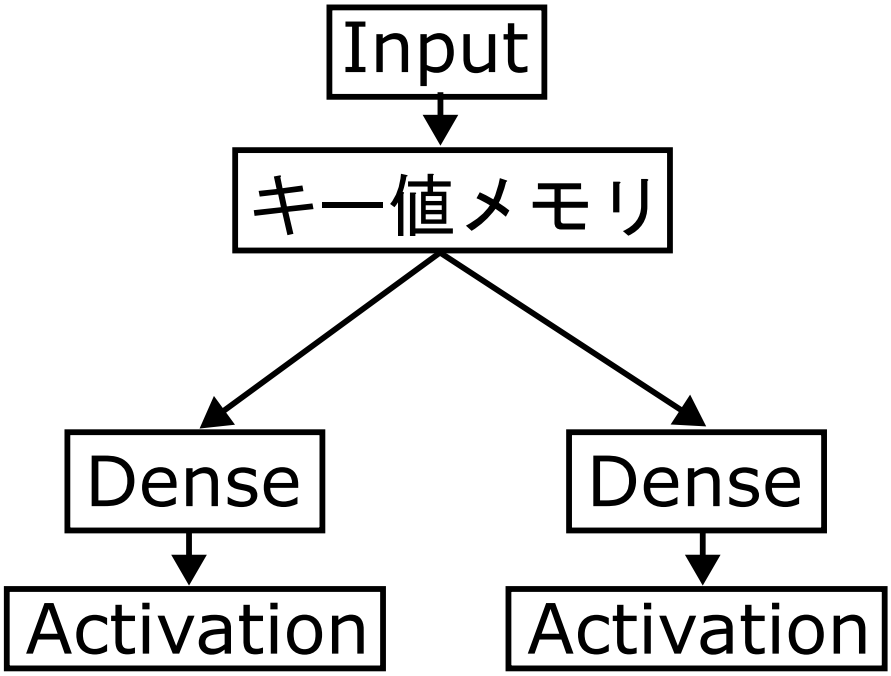

図 3.14: キー値メモリの学習の様子

3.4.5.1　生成器

訓練データ生成器は NPI にけるタスクの各ステップにおける観測情報とそれに対する理想的な出力を再現するため，生成器もアルゴリズム 1 に従って動作する．

生成器では NPI の学習器のうちシーケンシャルモデルに着目して訓練データを生成する．ここで，シーケンシャルモデルの推論時に入力されるデータはキー値メモリによるエンコーディング結果と実行中のサブプログラムの情報である．しかし訓練データの生成器は人手で設計するため，生成器の設計にキー値メモリからのエンコーディング結果を用いると，Scratch-Pad からの出力 e や前回のシーケンシャルモデルからの出力 $\boldsymbol{a}$ に基づいた生成器の設計を行うことができなくなってしまう．そこでデータセットの生成器は NPI の学習器そのものに入力される情報を基に，各ステップで出力するべきサブプログラムとそれに渡す引数，次のステップで現在実行中のサブプログラムを終了するかといった出力情報を決定し，訓練データの生成を行う．

3.4.5.2　訓練データ

訓練データには図 3.11，図 3.12，および図 3.13 に示した推論時に NPI の学習器に入力された情報 p_{in}，$\boldsymbol{a_{in}}$，e とそれに対する学習器の出力 r，p_{out}，$\boldsymbol{a_{out}}$ がステップ毎に保存されている．

また，訓練データ内のサブプログラム p には学習対象である通常のサブプログラムと Scratch-Pad に直接作用する即時サブプログラムの 2 種類存在しており，訓練データを構成するサブプログラムの呼び出しを追いかけていくと，呼び出しの一番深い部分では必ず即時プログラムが呼び出されて Scratch-Pad の更新を行っている．

第4章　色情報のみを用いた人検出器の NVIDIA Jetson Xavier上への実装

本章ではNVIDIA CUDA [10] を用いて並列実装された，Informed-Filtersに基づく人検出器のNVIDIA Jetson Xavier [8] 上への実装について述べる．

4.1　はじめに

著者が実現しようとしているIARを用いたセンサネットワークでは，図4.1に示すようにUnmanned Aerial Vehicle(UAV) や定点カメラから得られた画像に対して実時間でセンサを装着した人物を正確に検出する必要がある．

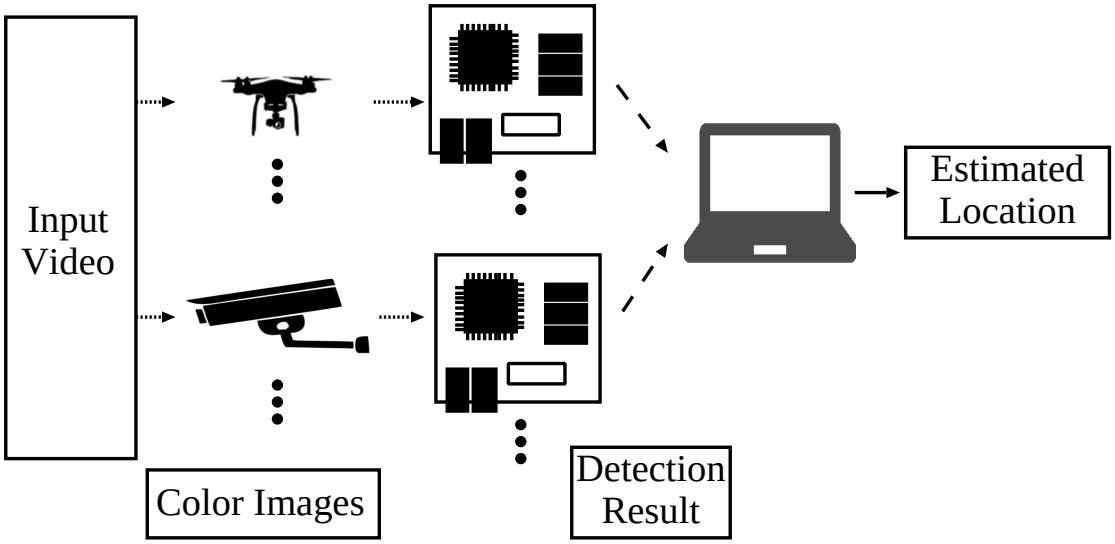

図 4.1: IARのネットワーク

第2章で述べたように，リアルタイムな人検出にはInformed-Filtersとその並列実装が有望であるという研究があり [3]，先行研究ではNVIDIA CUDAを用いてInformed-Filtersの並列実装を行っている．しかし [3] そのままではUAVから画像を直接取得し，それに対して人物の検出を行うことはできない．

この問題を解決するため，本章ではNVIDIA CUDAを用いて実装されたプログラムが動作するシングルコンピュータであるNVIDIA Jetsonを用いてUAVに搭載可能かつリアルタイムに動作

する Informed-Filters の並列実装の提案を行う．

4.2　NVIDIA Jetson

提案手法について説明する前に NVIDIA Jetson について説明を行う．NVIDIA Jetson はエッジコンピューティングで機械学習アプリケーションを実行するために設計されており，CPU として組込みシステムで用いられる ARM アーキテクチャのプロセッサを，GPU として NVIDIA 製の GPU を搭載していることが特徴である．これにより，小さなボード上での画像認識や深層学習の推論を高速かつ低消費電力で実現している．

Jetson には以下のようなバリエーションが存在する．

1. TK1
2. TX1
3. TX2
4. Xavier
5. Xavier NX
6. Nano

これらは発表時期や用途の違いから，GPU の世代やスペックに差異がある．さらに，中にはシングルコンピュータとしてではなく，GPU クラスタに搭載して用いる計算モジュールとしてのみ入手可能なモデルもある．提案手法ではこれらの条件や実現するシステムにおける人検出器をリアルタイムで動作させるために必要なシステム要件を考慮し，Jetson Xavier を用いてシステム実装を行う．以降では Jetson Xavier と，同じ GPU アーキテクチャの Xavier NX について述べる．

図 4.2: Jetson AGX Xavier

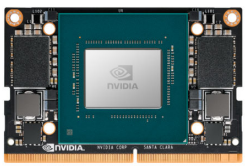

図 4.3: Jetson Xavier NX

表 4.1 および，表 4.2 にそれぞれ Xavier と Xavier NX の仕様を示す．表に示した仕様の他にも，Xavier は図 4.2 のような開発者キットが提供されておりそれを用いることで単体で動作させるこ

表 4.1: NVIDIA Jetson Xavier の仕様

Board	Jetson AGX Xavier
CPU	8-core NVIDIA Carmel ARM v8.2 64-bit CPU, 8MB L2 + 4MB L3
GPU	64 基の Tensor コア搭載 512 コア NVIDIA Volta アーキテクチャ GPU
Memory	16GB 256-Bit LPDDR4x — 137GB/s
Storage	32GB eMMC 5.1

表 4.2: NVIDIA Jetson Xavier NX の仕様

Board	Jetson Xavier NX
CPU	6-core NVIDIA Carmel ARM v8.2 64-bit CPU, 6MB L2 + 4MB L3
GPU	48 基の Tensor コア搭載 384 コア NVIDIA Volta アーキテクチャ GPU
Memory	16GB 128-Bit LPDDR4x — 59.7GB/s
Storage	16GB eMMC 5.1

とが可能なのに対し，Xavier NX は図 4.3 に示すような GPU クラスタに組込むためのボードしか提供されておらず，Xavier NX の他に組み込み用のケースやボードが必要であるという違いがある．また，Xavier NX は Xavier に比べて電力が限られた環境での運用が想定されており，計算に用いることのできるリソースやメモリの帯域幅に制限がある．

提案手法では以上の違いをふまえて Jetson の選定および，システムの実装を行う．

4.3　提案手法

NVIDIA Jetson シリーズのシングルボードコンピュータは CUDA コアが搭載された GPU を備えているため，CUDA を用いて設計されたプログラムが動作するのが特徴である．そのため，先行研究の並列実装をほぼそのまま移植することができるという利点がある．そこで，本節では NVIDIA Jetson を用いて [3] において CUDA で実装された人検出プログラムをエッジコンピューティング向けに実装する．まずは Jetson の選定を行う．

図 4.4 に，本章で実装するシテテムの構成を示す． システムではカメラから取り込まれた画像に対し人検出を行い，選手の位置情報を推定する．本システムが対象とする画像は図 4.5 に示すような 3820×2160 の解像度の画像におよそ 52×74 の大きさの人物が含まれるものであり，Maxwell 世代以前の GPU アーキテクチャの Jetson Nano や TX2 では少し計算リソースが不足してしまう．また今回は実装の簡略化のため，Jetson 本体の他にケースやボードが必要な Xavier NX ではなく，Xavier を用いる．

次に，人検出を行う手法の検討を行う．人検出を行う手法としては様々な手法が考えられるが [21, 57–59] ，本章で実装するシステムでは図 4.5 に示すような 3820×2160 の解像度の画像からおおよそ 52×74 の大きさの人物をリアルタイムで検出しなければならない．さらに，Jetson Xavier に搭載されている GPU はデスクトップ用のものに比べると計算リソースが限られる．これらを考慮し本システムでは計算量が少なく，さらに高精度に人検出を行うことのできる，Informed-Filters の GPU 実装 [3] を用いる．

図 4.4: 実装するシステム

システムでは Jetson に画像を直接取り込みそれに対して検出を行えるように，Jetson にカメラインタフェースを接続しているが，Jetson に対し 3820×2160 の解像度の画像を入力するためには，一般的な Web カメラではなく 4K 撮影が可能なビデオカメラを用意し，その HDMI 出力を USB Video Camera(UVC) として OS に認識させる必要がある．そこで，提案するシステムでは変換用インタフェースとして図 4.4 の中央に示すような AV.io 4K [60] を用いることにした．このインタフェースはビデオカメラだけでなく，ゲームや PC といったあらゆる HDMI 出力に対応したデバイスの映像出力を UVC に変換することができ，さまざまな OS の標準 UVC ドライバで動作することが特徴である．

以上より，システムの入力される画像の撮影には 4K 撮影が可能なビデオカメラを，その HDMI 出力を UVC に変換するために AV.io 4K を，そして検出処理を行うための計算機として NVIDIA Jetson AGX Xavier を用いる．これらを用いて実装したシステムは以下の手順で動作する．

1. カメラから画像を取得
2. スライディングウィンドウによる全探索
3. 識別器を用いたサブウィンドウの分類
4. NMS [61] を用いた，1 つの対象に対する複数の検出結果の統合
5. 検出結果のリストを作成，送信
6. (1) に戻り，次のフレームの処理

手順 5 において，システムは外部に検出した選手の位置情報を送信するが，現在はデータ収集ノードが存在しないため，選手の検出情報は外部に送信するかわりに，テキストファイルに書き出して Jetson 本体上の eMMC に保存する．

図 4.5: 検出対象が含まれた画像

4.4　評価

4.4.1　訓練

サブウィンドウを分類するための識別器は先行研究 [62] と同様の手順で訓練されている．弱識別器の深さは 1，弱識別器の個数は 200 個，強識別器は Adaboost で構築されている．

4.4.2　検出精度

図 4.6 に検出結果の一例を示す．画像中の赤枠が検出された対象を示している．図 4.6 内では誤検出，見逃しのいずれもないことがわかる．図 4.7 に本稿の検出器による DET カーブを示す．DET カーブ内では左下にあるグラフの精度が高い精度である．この図からもこの検出器の精度が高いことがわかる．

4.4.3　検出速度

実装したシステムの処理速度を評価するため，3840 × 2160 の解像度の実画像データセットを用いて画像一枚あたりの処理時間を計測する．

データセットの画像 1000 枚を用いて計測したところ，処理速度の平均値で 44.93ms であった．これは実時間処理において十分な速度であるといえる．

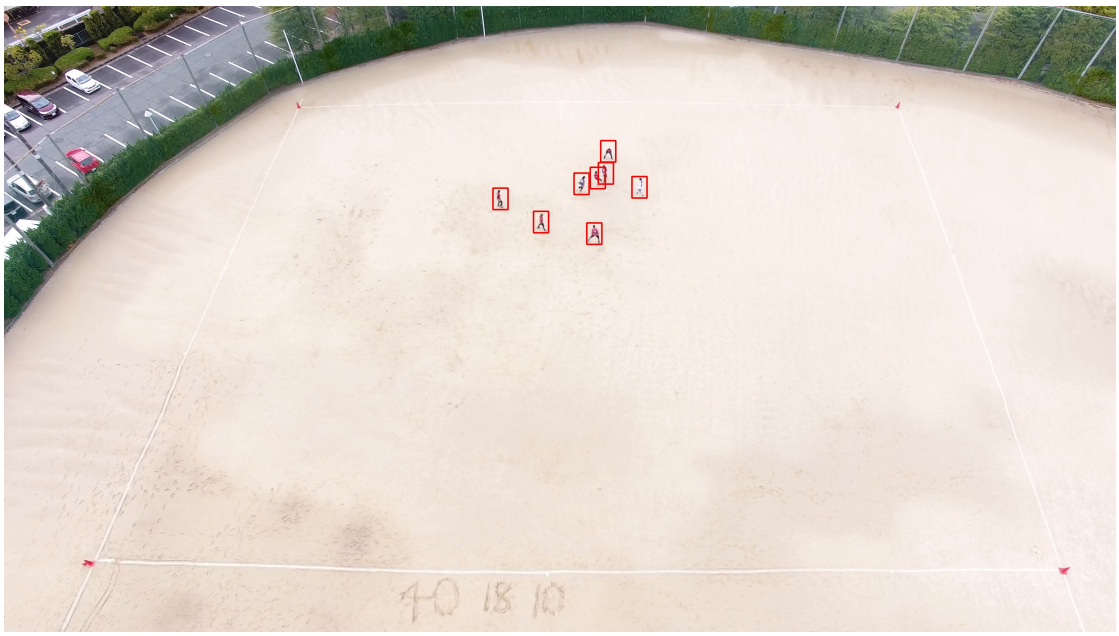

図 4.6: 検出結果の一例

4.5　まとめ

本章では UAV 上のカメラから画像を取得し，得られた画像からリアルタイムに人検出を行うシステムを実現するため，UAV に搭載可能かつリアルタイムに動作する人検出システムの提案を行った．そこで著者は，Informed-Filters の並列実装をそのまま UAV に搭載するため，CUDA が動作するシングルボードコンピュータである Jetson Xavier に対して検出器の移植を行うことでシステムの実装を行った．4K カメラから空撮画像を取得し，Jetson Xavier 上で検出処理を行うシステムの構築を行い，検出速度と検出精度の評価を行った．精度評価の結果 UAV で撮影された映像から高い精度で対象の人物を検出することができた．また検出速度の評価を行った結果，入力画像の解像度が 3840×2160 であるにも関わらず，およそ 23 fps で動作することが確認された．

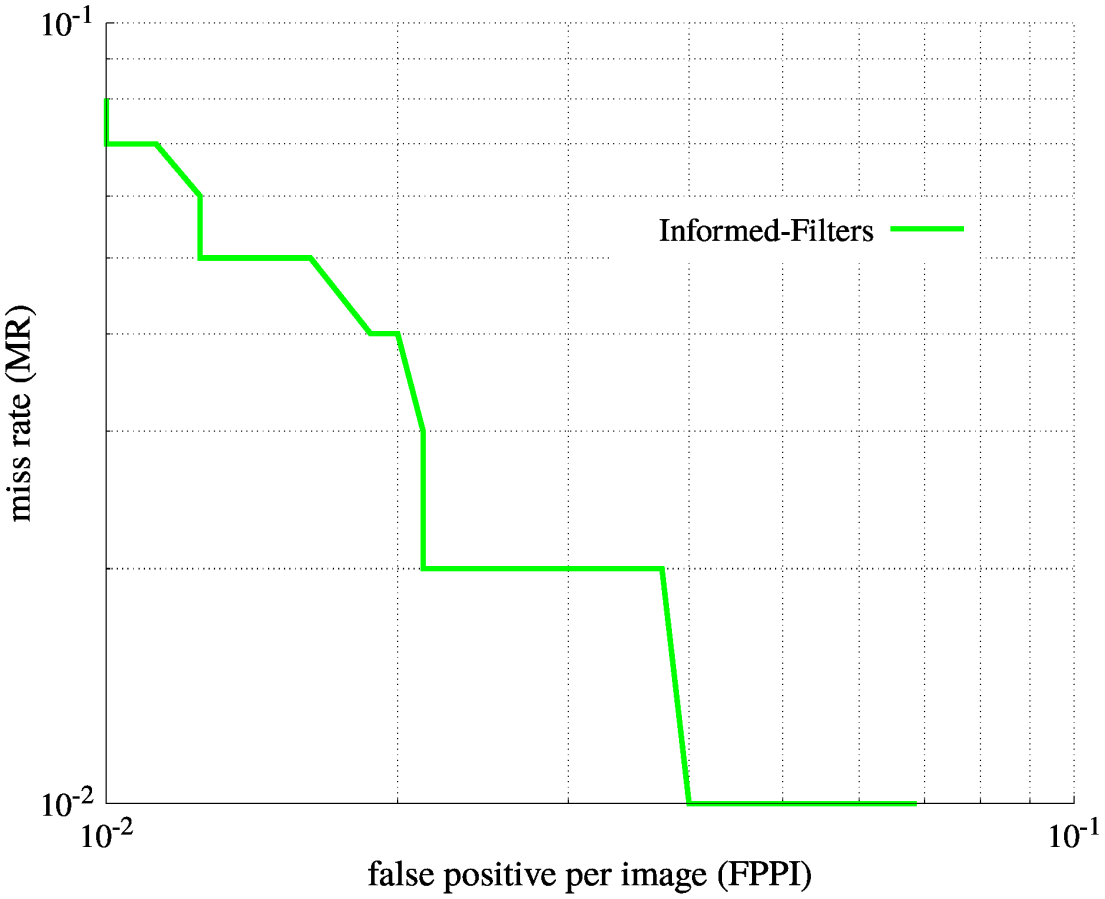
10-1
10-2
10-2
10-1
miss rate (MR)
false positive per image (FPPI)
Informed-Filters

図 4.7: DET カーブ

第5章　C++ AMPを用いたInformed-Filtersの並列実装

5.1　はじめに

4章ではUnmanned Aerial Vehicle(UAV)から直接画像を取得し，その画像に対して人検出処理を行うためのシステムとしてUAVに搭載可能かつリアルタイムに動作可能な実装の提案を行った．本章ではJetsonに搭載されているような省電力·省スペースなGPUではなく，先行研究[3]で用いられているようなデスクトップPC向けのGPUを用いて高速な人検出を実現するためにUAVから画像を無線経由で受け取り，それに対して人検出を行うシステムを構築する．

著者は空撮画像を撮影するためのUAVとしてDJI Phantom 4 Pro V2.0 [11]を用いることにしたが，Phantom 4 Pro V2.0から画像を取得するためには，ユニバーサルWindowsプラットフォーム(UWP) [12]を対象としたDJI Windows SDKを用いる必要がある．しかし，UWP上のアプリケーションはサンドボックス環境で実行されるため，CUDA [10]環境が利用できないという問題がある[63]．この問題を解決するためのプログラム実装方法として，UAVから画像を受け取るプログラムと人検出システムを分割しそれらの間でソケット通信を用いて画像の送受信を行う実装が考えられる．本章での実装に先立ちUnix上でソケット通信を用いてAndroidアプリを経由してデータの受け渡しを行うシステムの実装を行ったが，通信速度が安定しないという問題があった．そこで，本研究ではUAVからの画像の受け取りから人検出システムまでを1つのプログラムで行うことにした．

UWP上でリアルタイムに人検出を行うためには，CUDAの代替となる並列計算手法が必要となる．そこで，本章ではUWPにおいて外部GPUに代表される並列計算デバイス上で処理を行う枠組みである，C++ AMP [9]を用いることとした．これを用いることでCUDAを用いることのできないUWP環境においてもNVIDIA GPUを用いた並列計算が可能となる．このライブラリを用いてInformed-Filtersを並列実装することによって，DJI Phantom 4 Pro V2.0とリアルタイムに連携して動作する人検出ソフトウェアの実時間実装を得ることができる．

5.2　C++ AMP

C++ AMP [9]はMicrosoftが中心となって開発していた並列プログラミングを行うためのC++を対象とした拡張ライブラリである．Informed-Filtersの並列実装[3]ではCUDAが用いられているが，CUDAがNVIDIA GPU専用に設計されているのに対し，C++ AMPはそれに加えてAMD GPUやIntel CPU内蔵のGPUを用いて並列計算を行うことができる．また，GPUのアーキテクチャを考慮せずに逐次処理のループに近い形式で記述できることも大きな特徴である．

NVIDIA GPU では図 2.24 のように，グリッド内にブロックが，ブロック内にスレッドが生成される [64, 65]．CUDA においてはプログラマが GPU の構造を考慮してブロック数とグリッド数を決定し，ホスト-GPU 間のデータのやりとりや GPU メモリの確保のタイミングを適切に指定することで最適な粒度でプログラムの並列化を行うことが可能となっている．それに対し，C++ AMP ではループの繰り返し回数を独自の表記を用いて parallel_for_each 関数に渡すことで処理を並列化することができ，スレッドの生成数やデータの受け渡しのタイミングはライブラリが自動で決定する．

スレッド間の同期方法にも違いがある．CUDA では NVIDIA GPU のハードウェア仕様によりスレッド内部から実行中の全部のスレッドに対する同期をとることができず，ホスト側から cudaDeviceSynchronize を呼ぶことでこれを実現している．一方，C++ AMP ではスレッド内でのメモリの参照を行うための識別子である array_view に対してホスト側から synchronize を呼ぶことで，全スレッドの同期をとることができる．

5.3　提案手法

本章では，計算装置やその上のメモリの扱いの違いや制限事項を考慮した，C++ AMP を用いた Informed-Filters の並列実装について CUDA 版の実装と比較しつつ述べる．

Informed-Filters を実装する上での違いを述べる前に C++ AMP における並列実装の様子を図 5.2 に，CUDA と C++ AMP におけるメモリ確保から並列処理，スレッドの同期までの流れをそれぞれ図 5.3 および図 5.4 に示す．ここで 2.3.2.2 で述べたように，Informed-Filters の並列実装の方式にはサブウィンドウ毎の並列化と弱識別器毎の並列化の 2 通り考えられる．先行研究 [3] の CUDA 版の実装では各弱識別器におけるワープダイバージェンスの発生頻度とソフトカスケード構造による計算量の削減効果を考慮してサブウィンドウ毎の並列化を行っているため，本章の実装でもサブウィンドウ毎の並列化を行う．

ソースコード 5.1: filter_info_amp.hpp

```
1  concurrency::array<float, 1>* d_thrd;
2
3  void load_params(...) {
4    std::vector<float> h_thrd;
5    ...
6    while(...) {
7      h_thrd.push_back(value);
8    }
9    ...
10   d_thrd = new concurrency::array<float, 1>(h_thrd.size(), std::begin(h_thrd));
11 }
```

5.3.1　検出器の実装

CUDA を用いた並列実装 [3] においては，全てのブロックから高速にアクセスすることのできるコンスタントメモリに特徴量計算のフィルタや，弱識別器の葉ノードのスコア，二分木やカスケードの閾値を配置することによって，高速な処理を実現していた．一方で C++ AMP は様々なアーキテクチャの並列演算器に用いることができるように，CUDA にみられるようなメモリ階層

の概念はない．そこで，本実装ではソースコード 5.1 に示すような Concurrency Array と呼ばれる C++ AMP において並列計算に用いられるメモリ領域にホスト側で読み込んだ値に対する参照を作り，必要に応じて GPU に転送するようにしている．

5.3.2　デバイスの初期化 · メモリ割り当て

5.2 節で述べたように，C++ AMP では一般に並列計算に用いられる外部 GPU から CPU の内蔵グラフィックまで，さまざまな計算装置を用いて並列計算を行うことができる．そのため，C++ AMP の初期化においてはライブラリが検出した計算装置のリストを取得し，その中から使いたいものを選択する必要がある．本実装では CUDA 版の実装との比較を行うため，ソースコード 5.3 先頭の amp_initialize に示すように NVIDIA の GPU を用いるように実装した．

また，CUDA にはデバイスの初期化はないものの，ソースコード 5.2 の 15 行目や 29 行目，図 5.3 の①〜③にあるようなメモリの確保 · 解放やメモリのコピーといった処理が必要となる．C++ AMP ではメモリの確保 · 解放，コピーを行うかわりに，ソースコード 5.3 の 16〜17 行目や図 5.4 中の①から②に示すように，デバイスからホストのメモリへの参照を作成しており，メモリの確保 · 解放およびデータのコピーはライブラリが行う．

ソースコード 5.2: cuda_kernel.c

```
1  // 計算カーネル
2  __global__ void kernel (...)
3  {
4      const int x = blockDim.x * blockIdx.x + threadIdx.x;
5      const int y = blockDim.y * blockIdx.y + threadIdx.y;
6      int id = (cols-w_size)*y/step+x;
7      ...
8      result[id] = allsum;
9  }
10
11 // 呼び出し部
12 void detect_gpu(...)
13 {
14     // data transfer
15     cv::cuda::GpuMat d_img(img);
16     static cv::cuda::GpuMat d_aux;
17     static cv::cuda::GpuMat d_luv[3];
18     ...
19     // preprocess
20     cv::cuda::cvtColor(d_img, d_aux, cv::COLOR_BGR2Luv);
21     cv::cuda::GpuMat d_luv_cat( cv::Mat(cv::Size(d_aux.cols*3, d_aux.rows), CV_8UC1, cv::Scalar
           (0)) );
22     ...
23     // synchronize threads
24     CV_CUDEV_SAFE_CALL(cudaDeviceSynchronize());
25     kernel<<<grid,block>>>(pImg, d_result, d_flt, d_img.cols, d_img.rows, step, w_size, h_size);
26     // synchronize threads
27     CV_CUDEV_SAFE_CALL(cudaDeviceSynchronize());
28     // copy to host
29     cudaMemcpy(&h_result[0], d_result, range_result*sizeof(float), cudaMemcpyDeviceToHost);
30     ...
31
32     cudaFree(&d_result);
33     return;
34 }
```

ソースコード 5.3: amp_kernel.cpp

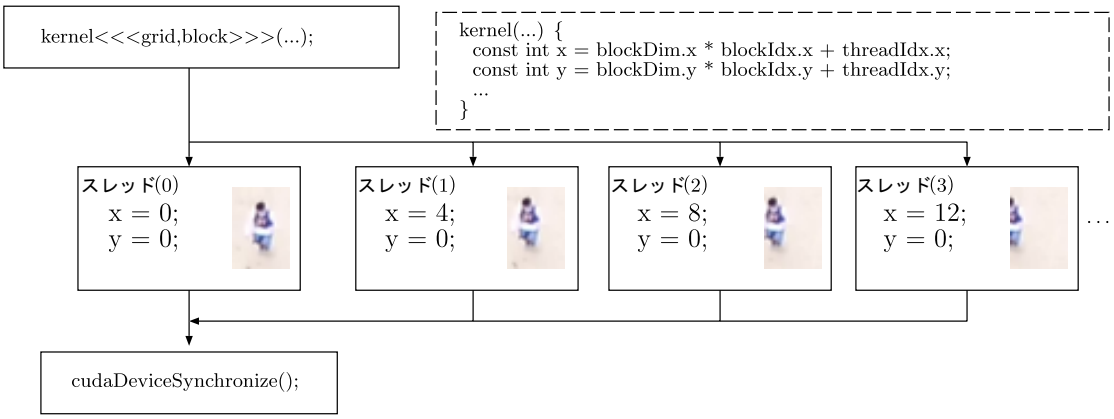

図 5.1: CUDA による並列化

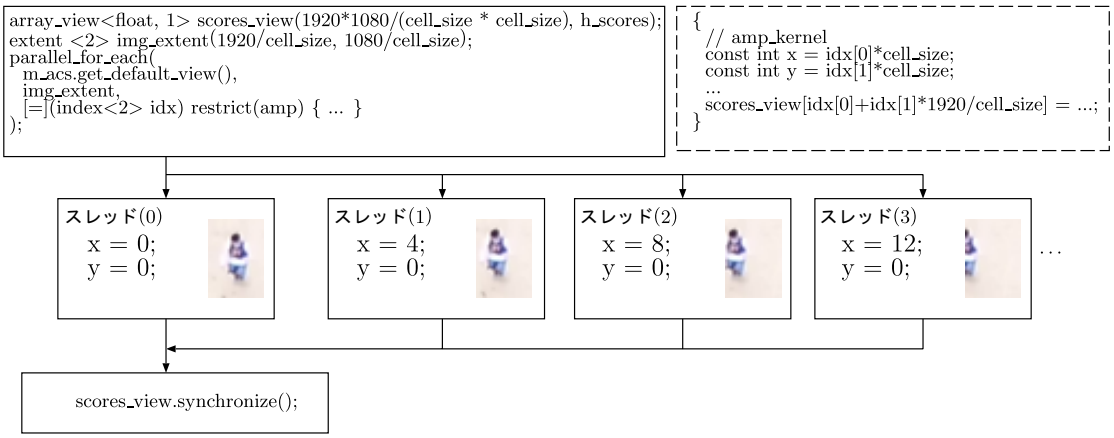

図 5.2: C++ AMP による並列化

```
1  void amp_initialize() {
2    // set accelarator
3      std::vector<accelerator> m_acs_lst = accelerator::get_all();
4      for (int i = 0; i < m_acs_lst.size(); i++)
5      {
6          if (m_acs_lst[i].get_description().find(L"GeForce")) {
7              m_acs = m_acs_lst[i];
8          }
9      }
10 }
11
12
13 void AMPDetector::AMPFeatureCalculation(const cv::Mat& img, float* result)
14 {
15   // make array_view of the detection parameters
16   array_view<float, 1> val_view = *val_a;
17   array_view<float, 1> thrd_view = *thrd_a;
18     ...
19   // clear current content to prepare for feature calculation
20   extent<2> img_flat_extent(img.rows, img.cols);
21   concurrency::array<unsigned int, 2> img_array(...);
22
23   // img_array for accelerator
```

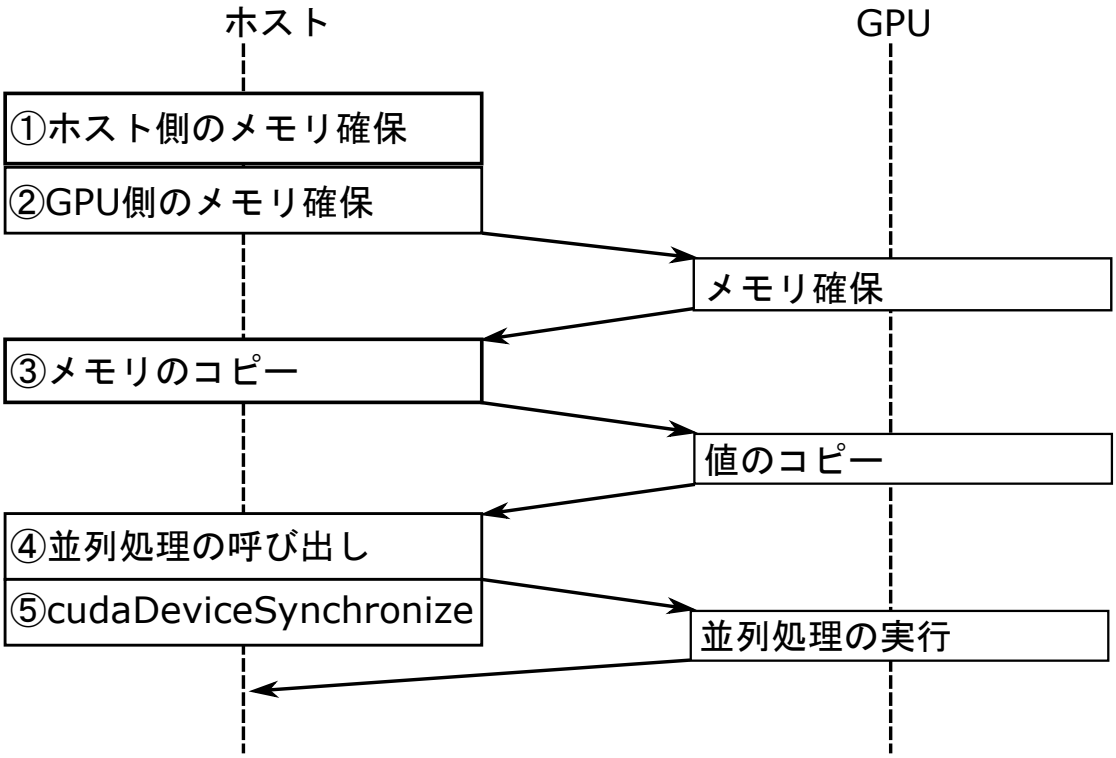

図 5.3: CUDA における並列処理の様子

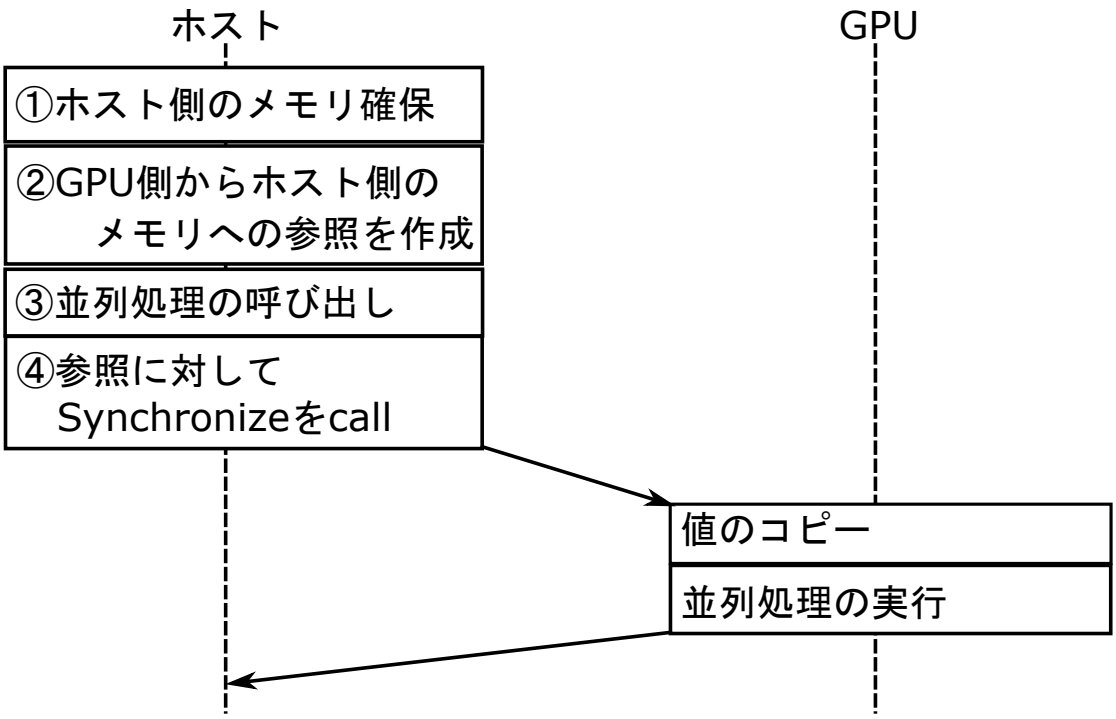

図 5.4: C++ AMP における並列処理の様子

```
24    array_view<unsigned int, 2> data_view = img_array;
25
26    // the dimension of kernel
27    extent<2> img_extent(rows_, cols_);
28      ...
29    // kernel
30          parallel_for_each(
31                ...img_extent,
32                [=](index<2> idx) restrict(amp) {
33        int u = idx[1] * p_step_p_;
34        int v = idx[0] * p_step_p_;
35
36                const int id = (cols_p_ - w_size_p_) * idx[0] / p_step_p_ + idx[1];
37                index<1> idx_id(id);
38                ...
39                result_view[idx_id] = sumall;
40      }
41    );
42
43    result_view.synchronize();
44  }
```

5.3.3　前処理

先行研究 [3] では，検出処理を行うための前処理として以下の処理を行っている．

- cv::cuda::cvtColor を用いた色空間の変更
- cv::cuda::integral を用いた積分画像の作成

これらの処理は OpenCV [66] の CUDA 向けの API をそれぞれ用いて行われるが，UWP で動作するプログラムからは CUDA が利用できないという制限がある [63]．そこで，本章では OpenCV の CPU 向けの SIMD [67] 実装である cv::cvtColor，cv::integral をそれぞれ用いてこれらの処理を実現する．

5.3.4　特徴量計算の実装

画像に対する前処理が終わると，特徴量計算が行われる．CUDA 版では前処理を GPU 側で行っているため，画像データの転送は前処理の時点で完了している．一方 C++ AMP 版は前処理をホスト側で行うため，ソースコード 5.3 に示すように前処理が終わったデータに対して参照を作成し並列処理を呼ぶことで，ライブラリが並列処理を行う際に自動的にホスト側から GPU にデータのコピーを行う．

Informed-Filters の特徴量計算はサブウィンドウ単位で処理の並列化されているため，スレッド内の処理には現在計算しているサブウィンドウの座標が必要である．5.2 で述べたように，C++ AMP では extent を用いて処理の繰り返し回数を宣言することにより，ライブラリが自動的にスレッドを生成し，各スレッドにインデックスを割り当てる．これにより，CUDA 版ではスレッドやブロックの ID とブロックの大きさからサブウィンドウの座標を計算する必要があるのに対し，本実装では図 5.2 のように extent として画像の解像度をセルの大きさで割った値を，サブウィンドウの座標としてインデックスにセルの大きさを掛けた値を用いることにより，CUDA 版よりも簡単な計算でサブウィンドウの座標を求めることができる．

5.3.5　後処理

サブウィンドウ毎のスコアの計算が終わるとホスト側でその集計を行う．そのためには GPU での結果の書き込みが終わっている必要がある．ソースコード 5.2 の 27 行目や図 5.3 の⑤のように，CUDA 版ではスコア計算処理の同期をとるために，GPU 上の全スレッドが停止するまで待機する，cudaDeviceSynchronize を用いていたが，C++ AMP 版の実装ではソースコード 5.3 の 41 行目や図 5.4 の④にあるように，スレッド内でスコアを書き込む領域である result_view に対して synchronize を呼ぶことによって同期を取っている．

5.4　評価

本章では Informed-Filters の C++ AMP 版実装の検出精度と実行時間について述べる．

5.4.1　評価環境

表 5.1 および表 5.2 に本検出器の評価環境と検出器のパラメータを示す．データセットに用いた画像は DJI Phantom 4 Pro V2.0 から無線で取得できるフレームの解像度に合わせて，1920×1080 とした．また，図 5.5 に識別器の訓練および評価に用いた画像の一例を，図 5.6 にその画像に対す

表 5.1: 評価環境

OS	Windows 10
CPU	AMD Ryzen 7 3700X
GPU	NVIDIA GeForce RTX 3090
Memory	DDR4 64GB

表 5.2: データセット・パラメータ

サブウィンドウのサイズ	26×37
弱識別器の個数	200
弱識別器の深さ	1
枚数 (訓練用)	1000 枚
枚数 (評価用)	1000 枚
フレームの解像度	1920×1080
特徴テンプレート	819 種類

る検出結果を示す．これらの画像は，サッカーのミニゲームの様子を，上空から UAV を飛行させて実際に撮影した動画から抽出されたフレームである．この一連の画像には常に 8 人の検出対象が写っており，フィールドをやや斜め上から撮影している関係で UAV から見て手前の対象は大きく，奥側の対象は小さく写っている．

Informed-Filters を用いるにあたって強識別器を構成する弱識別器の個数や深さ，分岐ノードの特徴テンプレートが必要となる．本実装の識別器の学習に使用した特徴テンプレートは 819 種類で，強識別器は 200 個の深さ 1 の木構造の弱識別器から構成されている．特徴テンプレートは先行研究 [3] と同様の方法で設計されている．

図 5.5: 入力画像

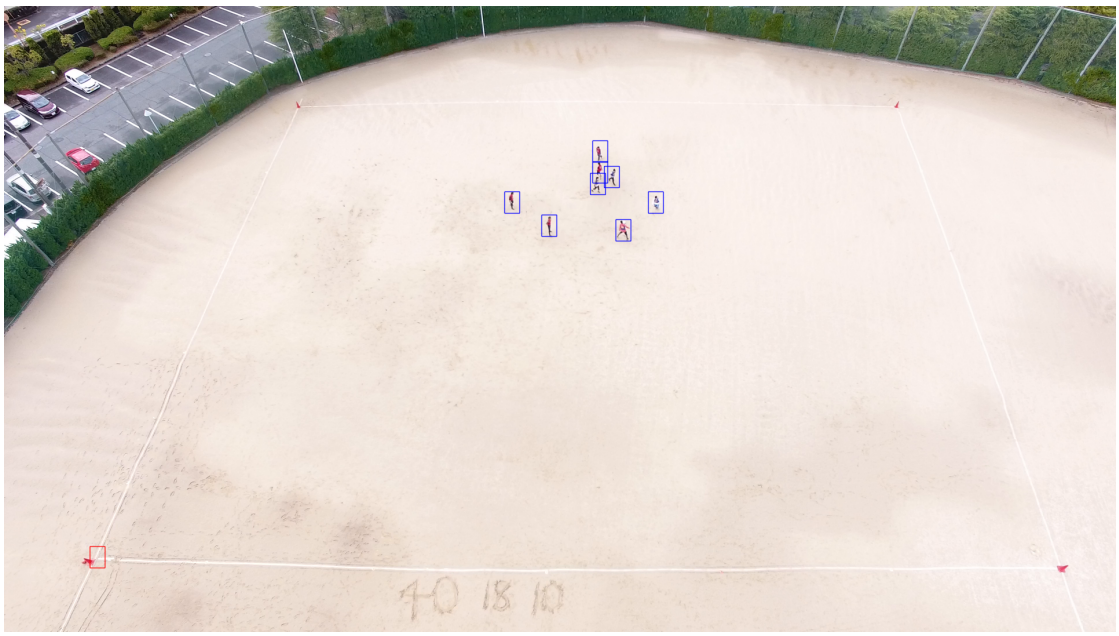

図 5.6: 検出結果

5.4.2　検出精度

本章で実装した検出器の検出精度の評価指標として，Detection Error Trade-off(DET) 曲線を用いる．これは図 5.7 のように，縦軸が見逃し率を示す Miss Rate(MR) を，横軸がテスト画像一枚あたりの誤検出数を表す False Positive Per Image(FPPI) を表すグラフで，検出器が出力した矩形を切り捨てるための閾値を変化させてプロットしたものである．DET 曲線がグラフの原点に近づくほど高精度な検出器であることを示している．また，MR と FPPI は以下のような式で表される．

$$FPPI = \frac{False\_Positive}{Number\_of\_Images} \tag{5.1}$$

$$MR = \frac{False\_Negative}{False\_Negative + True\_Positive} \tag{5.2}$$

ここで，$False\_Positive$ は検出器が対象の存在しない場所に出力した矩形の個数，$False\_Negative$ は画像内に存在する対象のうち見逃したものの個数，$True\_Positive$ は正しく見つけられた対象の個数を表している．検出器が出力した推定矩形と正解矩形の IOU が 0.6 以上の場合を検出成功とし，$True\_Positive$ としてカウントしている．

また本実装の有効性を確かめるため，近年精度の向上が著しい深層学習を用いた手法との比較を行う．今回のタスクで検出する対象は人物ではあるが，その大きさはおよそ 20×37 と非常に小さい．このようなタスクは小物体検出と呼ばれており，特別な深層学習モデルの利用が不可欠である．

そこで本研究では，先行研究 [68, 69] を参考に，EfficientDet-D0 [70] と RetinaNet [71] を比較対象とし，どちらも小物体用の特別な対策を行う．EfficientDet-D0 に対しては，モデルの学習時に ForceMatch という手法を適用する．これはモデルの学習時に正解矩形との IoU が十分に高いアンカーボックスが存在しない場合には，推論された中で最も IoU が高いアンカーボックスを強制

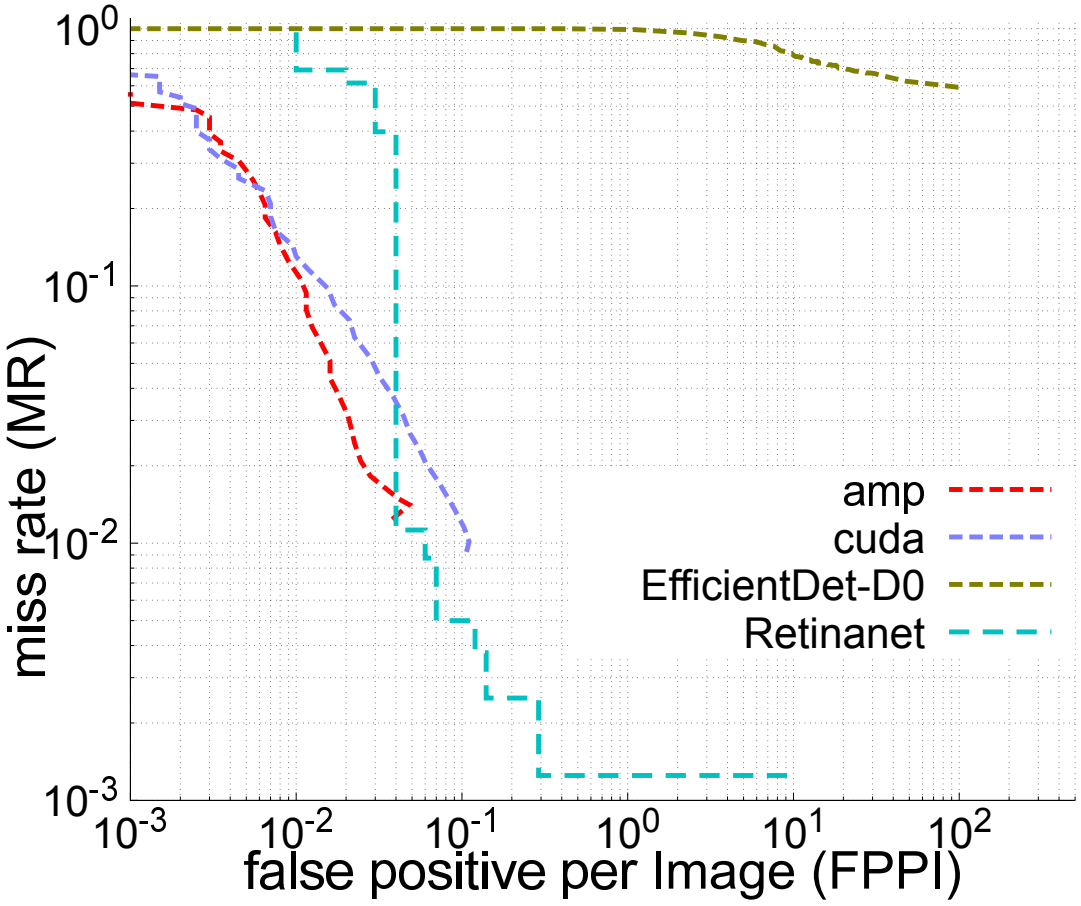

図 5.7: 検出精度

的に正解矩形にマッチさせることである．これにより，モデルが出力したアンカーボックスの全てが負例として扱われることを防いでいる．また，RetinaNet では層の浅い ResNet9 をバックボーンとして用い，さらに入力に近い P3 層の特徴量を使用することにより，ネットワークの通過による小物体の特徴量の消失を防止している．

ここで，2 つの深層学習をベースとした検出器は Informed-Filters とは異なり，サブウィンドウを抽出する手法ではなく画像全体を入力としてそれに対して検出窓の位置と大きさを推定する手法である．そのため精度評価においてはテスト画像全体を入力し，これらの手法の性能が適切に発揮されるようにした．

さらに，Informed-Filters が C++ AMP に正しく移植されていることを示すため，CUDA 版の実装と精度を比較した．精度比較の結果，EfficientDet-D0 はどの閾値でもほとんどの対象を検出することができず，画像一枚あたりの誤検出数も多くなった．Retinanet を用いた場合は，閾値が低い条件では今回比較した手法の中で最も多くの対象を検出することができているが，誤検出を削減するために閾値を高くしていったところ，見逃し率が高くなった．また，Informed-Filters を用いた検出器の CUDA 版と C++ AMP 版では同じパラメータを用いて実験を行ったのにも関わら

ず 2 つのグラフには若干の差異が見られる．この原因として，前処理に用いる計算装置の違いが考えられる．CUDA 版では GPU を用いて前処理を行っていたのに対し，C++ AMP 版では CPU を用いてそれを行っている．これに関して，OpenCV 4.5.0 の CPU 版と CUDA 版の色空間の変換をそれぞれ同じ画像に対して適用し処理結果を比較したところ，結果画像の画素値に一部差異が見られた．この差異はライブラリの実装に起因するため，本評価ではそのまま用いることとした．

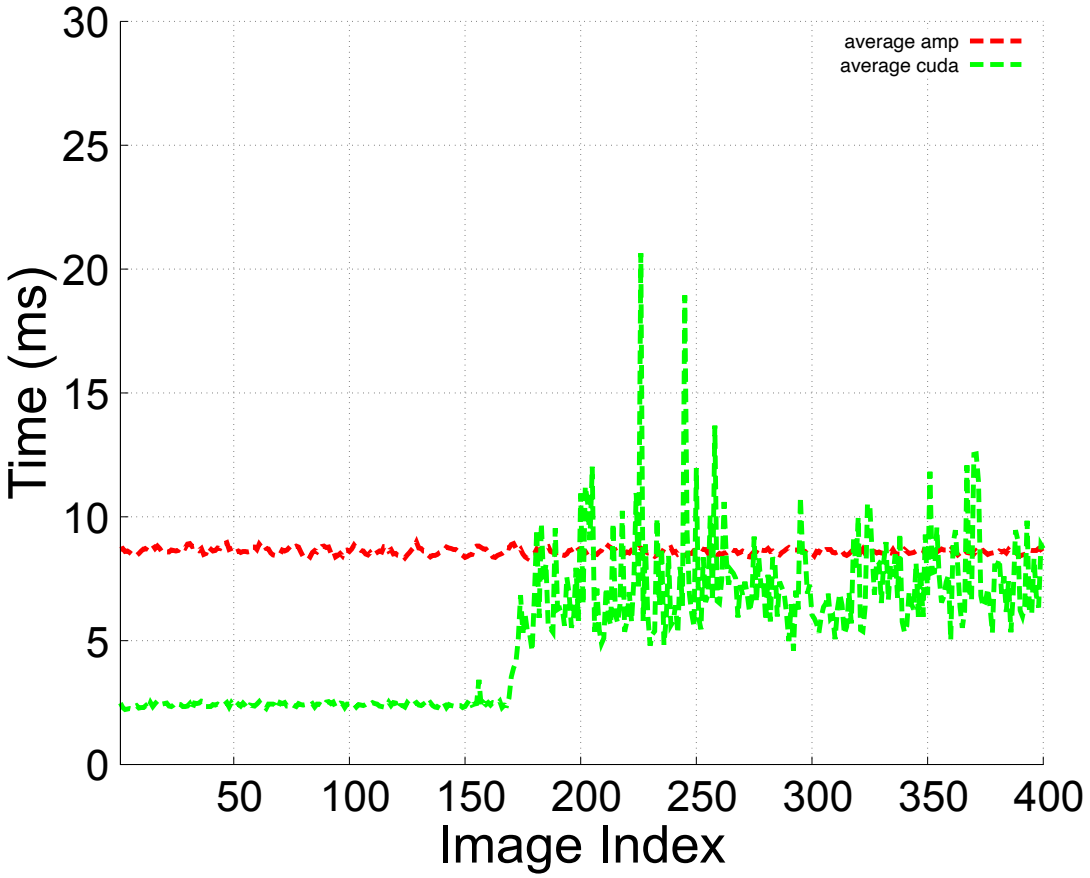

図 5.8: 前処理の実行時間

5.4.3　実行時間

本章で実装した検出器はソフトカスケード構造をとっており，検出対象が含まれていないサブウィンドウに対しては早期に検出を打ち切るため，検出対象や分類が困難な領域の少ない画像に対しては高速に検出を行えるが，そうでない画像に対しては多少検出速度が低下する．また，並列化に GPU を用いている関係でその初期化やメモリ確保 · 解放などが検出速度に影響することも考えられ，単純に算出された平均速度を指標とすることは適切ではない．そこで，一連のテスト画像を

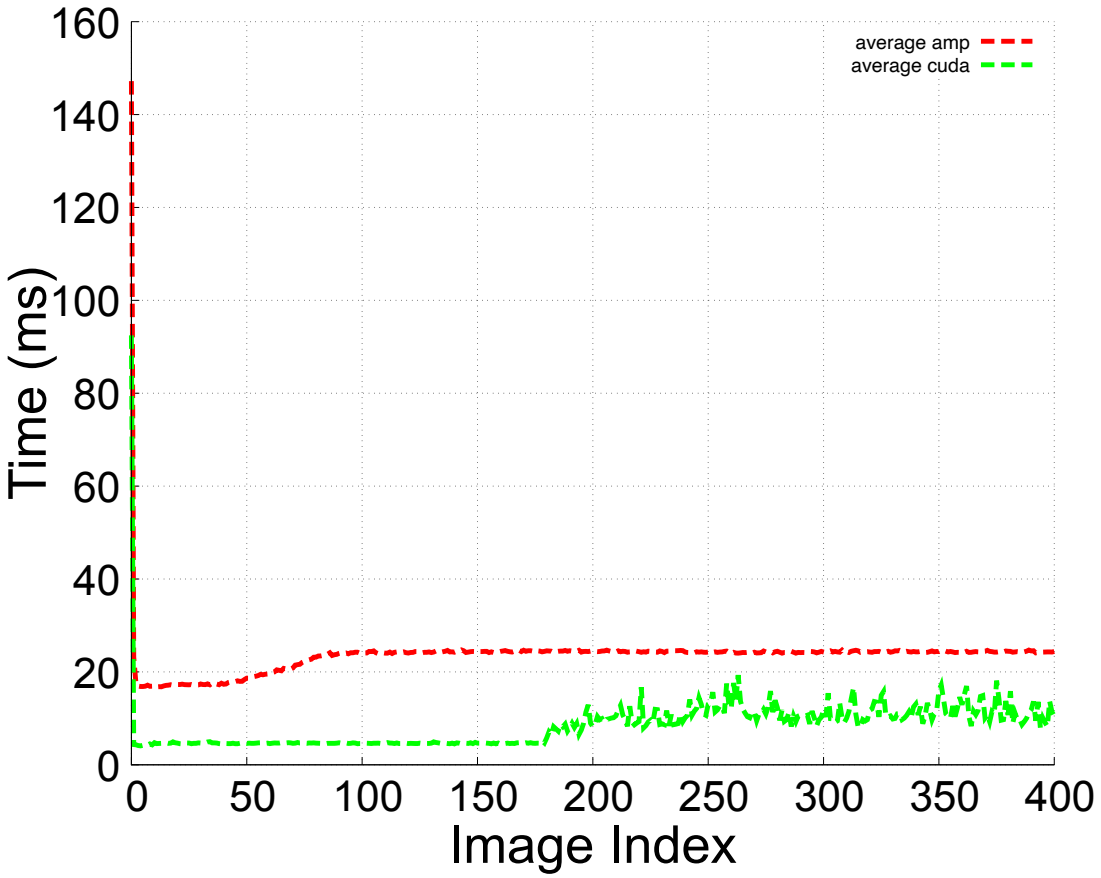

図 5.9: 検出処理全体の実行時間

ランダムに並び替えたリストを 50 通り作成し，それぞれに対して連続処理したときの実行時間の平均を図 5.8 および図 5.9 のようにプロットすることにした．

まず，検出処理の前処理に対する評価を行う．図 5.8 に CUDA 版および C++ AMP 版の検出器の前処理における実行時間をそれぞれ示す．CUDA 版の前処理の実行時間は後半になるにつれて変動している．一方で C++ AMP 版の前処理は全フレームを通してほぼ一定である．CUDA 版で処理時間が変動してしまった原因として，前処理と特徴量計算が共に GPU 上で行われていることが考えられる．

つぎに，前処理と強識別器による特徴量計算を合わせた実行時間を図 5.9 に示す．ただし，プログラムの処理時間は一般に部分的な時間測定と処理全体の時間測定とで異なるため，前処理の計測とは独立に行った．デバイスの初期化が入る最初のフレームを除けば時間を要した検出サンプルでも 40 ms 以下となっている．また，最初のフレームを除いたテスト画像全体では 41.8 fps の処理速度となっている．この結果は，本研究において目的としている Image-Assisted Routing を実時間で行うのに十分な速度である．Windows 用にビルドした CUDA 版の実装と比較すると，C++ AMP 版は 15 から 20ms 程度遅い傾向にあるが，これは C++ AMP がメモリ管理やスレッドの生

表 5.3: 各検出手法の平均検出速度

手法	検出速度 [ms]
Informed-Filters on C++ AMP	23.92
EfficientDet-D0	27.21
RetinaNet	160.43

成を自動で行っていることや，検出器のパラメータが高速にアクセスできるコンスタントメモリにないことが原因と考えられる．

CUDA 版，C++ AMP 版ともに，フレームがある程度進むと処理時間が少し伸びる傾向にある．それぞれの実装をデバッグ実行し計算リソースの消費量の確認を行ったが，原因の特定には至らなかった．また，本章の実装に際して，メモリの解放を手動で行う実装，ライブラリに任せる実装の 2 通り試したが前者では実行時間が安定せず，フレームレートも低くなってしまった．そこで，どちらの実装においてもメモリ上のメモリ解放をライブラリに任せている．後半のフレームでは計算リソースの消費に伴い，特徴量計算と同時にメモリの自動解放が発生することによりこのような処理時間になっていると考えられる．

最後に，5.4.2 節で比較した深層学習を用いた手法の検出速度の平均を表 5.3 に示す．この結果は，Informed-Filters が対象をほとんど検出できなかった EfficientDet-D0 や，Informed-Filters と近い精度を示した RetinaNet よりも高速に動作することを示している．

5.5　まとめ

運動中の人物の生体情報をリアルタイムに収集可能なセンサネットワークの実現を目指した研究が行われているが，センサノードの密度がある程度高く，その移動速度もある程度高速であるため，一般にマルチホップネットワークの動的な構成変更の手掛かりとして用いられる RSSI が利用できないという問題がある．この問題を解決するために，センサノードを装着した人物の位置に基づきネットワークを構築する Image-Assisted Routing が提案されているが，その実現には，ドローンから送信される画像を入力としてリアルタイムに人の位置を検出するシステムの構築が不可欠である．本章では，空撮画像を用いたリアルタイム人検出を実現するために，DJI Phantom 4 Pro V2.0 を撮影デバイスとし NVIDIA GPU を計算デバイスとするシステムを対象とし，Informed-Filters に基づく手法によりリアルタイム人検出を実現するシステムの構築を行った．

DJI Phantom 4 Pro V2.0 から画像を取得するためには，ユニバーサル Windows プラットフォーム (UWP) を対象とした DJI Windows SDK を用いる必要があるが，UWP には並列実装において広く利用されている NVIDIA CUDA 環境が利用できないという大きな制約があるため，本章では UWP から利用可能な並列プログラミング環境である C++ AMP を用いて Informed-Filters の並列化を行った．

1920×1080 の解像度の空撮画像を用いて速度と精度の評価を行った結果，Informed-Filters は小物体の検出に特化した RetinaNet と同等かそれ以上の精度精度を達成した．検出速度については，CUDA 版の実装と比較してフレームあたり 15 から 20ms 程遅い傾向にあるものの，最も時間のかかったフレームでも 40ms 以下であった．テスト画像全体の平均フレームレートでは 41.8 fps と

なっており実時間処理が可能であることが確認でき，深層学習を用いた小物体検出手法と比較しても高速に動作することが確認できた．

以上より，Image-Assisted Routing に不可欠な空撮画像からのセンサノードの位置推定を UWP 上で C++ AMP を用いて Informed-Filters を実装することにより，DJI Phantom 4 Pro V2.0 から得られる画像を入力としてリアルタイムに動作する高精度な人検出システムが実現できることを確認した．

本章で実装した検出器の実行時間の測定において，フレームが進むと処理時間が少し伸びる傾向が確認されたが，その原因のさらなる究明は今後の課題の 1 つである．また，本実装で用いている C++ AMP は開発停止となっているが，今後 Informed-Filters を他のフレームワークに移植する必要が出てきたとしても，本実装や CUDA 版の実装から得られた知見の活用が期待できる．

今後は，ネットワーキング処理および追跡処理との統合を行い，Image-Assisted Routing に基づくリアルタイムバイタルセンシングシステムの有効性を示したい．

第6章　Neural Programmer-Interpreters を用いたRISC-V命令セット向けのマシンコード生成

6.1　はじめに

近年，畳み込みニューラルネットワークの多層化により，従来は困難であったタスクにも機械学習が応用可能になりつつある．OpenAI の GPT-3 は与えられた情報から文章を作成することのできる機械学習の手法である．文章の生成だけでなく，この手法は自然言語により，仕様が与えられると，それを満たすようなプログラムのソースコードを生成するタスクにも応用されている．しかし，GPT-3 を用いて実行形式を直接生成する手法はまだ提案されていない．

一方で，アルゴリズムを模倣学習で訓練する手法が提案されている [72,73] ．中でも NeuralProgrammer-Interpreters [50](NPI) という手法は，タスクをプログラムで表現することを学習する．NPI においては学習器の外部記憶であるスクラッチパッドと，サブプログラムの並びで表されたアルゴリズムの実行例を与えることで，サブプログラムの中でも即時サブプログラムと呼ばれる構成単位はスクラッチパッドを操作するためのプログラムである．これらは NPI において，CPU の命令セットのような役割を果たしている．

著者は，NPI が実行単位を組み合わせてアルゴリズムを解くためのプログラムを構築するということが，コンパイラが CPU の命令を適切に組み合わせてプログラムを構築する様子と似ていることに着目した．即時サブプログラムとしてあるアーキテクチャの CPU の命令セットを用いて NPI を学習することにより，すでに構築されていて，あるマシンで走っているプログラムを元に，別のアーキテクチャのマシン向けの実行形式を直接生成できるようなプラットフォームを構築できるのではないかと考えた．

NPI の能力は、見本のプログラム列を与えると、そのタスクの解き方を学習できることである。この能力を適切に活用すると、あるアーキテクチャのプロセッサで動作するプログラムがあった時に、異なるターゲットアーキテクチャで動作する machine code を生成できる可能性がある。これが可能となれば、あるアーキテクチャ動作するプログラムがありさえすれば、簡単に異なるアーキテクチャにプログラムを移植することが可能となる。

本稿ではそのアイデアを確かめるため，即時サブプログラムとして RISC-V の命令セットの一部を用い，乗算をソフトウェア的に実現する，ソフト乗算を対象として，NPI の学習を行う．即時サブプログラムが対応する命令と同等の動作を行うために，本稿ではスクラッチパッドの再実装を行った．評価実験においては，NPI 構成要素として RISC-V [74, 75] アーキクチャベースの命令を用いた実験環境を作成し，ソフト乗算の学習がどの程度まで可能であるかの検証を行う．本稿では、このようなバイナリコードの移植を可能とするシステムの実現を目指し、あるプログラム列か

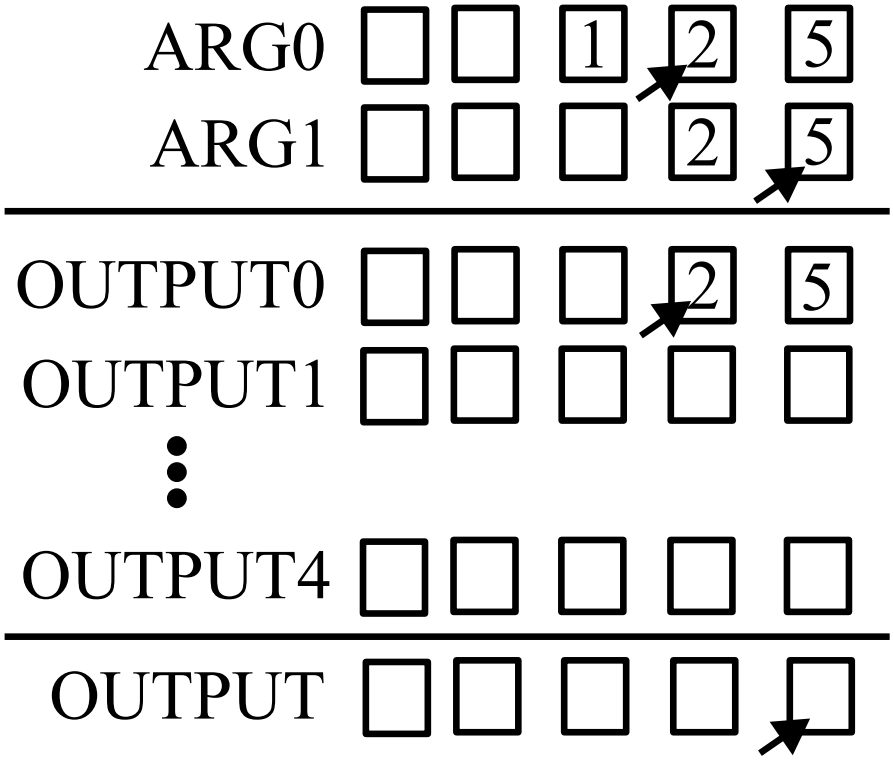

図 6.1: 予備実験における Scratch-Pad

らマシンコードの生成が可能であるか否かの検証を行った。ただし、本稿における実験では、入力出力ともに RISC-V アーキテクチャとし、主たる目的は、プログラム列の入力からマシンコードを生成可能であるかを確認することとした。

6.2　予備実験

本節では，本章の提案手法に先立って行った予備実験について述べる．

6.2.1　学習の対象

先行研究 [50] では，NPI の学習器に対して加算，ソーティング，3D モデルの回転を学習させている．そこで，著者は加算より少し複雑なタスクとしてかけ算をソフトウェアで表現する，10 進のソフト乗算をターゲットとして NPI の学習器の訓練を行った．

6.2.2　Scratch-Pad

予備実験を行うに際し，図 6.1 のようなスクラッチパッドを実装した．先行研究の加算の筆算のタスクでは CARRY と OUTPUT があればタスクを行うのに十分であるが，ソフト乗算では最初

表 6.1: 予備実験におけるサブプログラム

サブプログラム	動作
MUL	ソフト乗算の開始サブプログラム
MUL1	10 進 1 桁のソフト乗算を行うサブプログラム
ADD	ソフト乗算後段の加算を行うサブプログラム
ADD1	10 進の半加算を行うサブプログラム
CARRY	10 進の半加算において桁上がり処理を行うサブプログラム
RESET	ARG0，OUTPUT0 4 の注目座標を全て右端に移動する即時サブプログラム
PTR	Scratch-Pad における注目座標を移動する即時サブプログラム
WRITE	注目座標の位置にデータを書き込む即時サブプログラム

に第一引数 ARG0 と第二引数 ARG1 のそれぞれの桁同士での乗算を行い，その結果を合計する必要があるため，ARG1 の桁数分の OUTPUT レジスタが必要となる．そこで，図 6.1 のスクラッチパッドでは OUTPUT レジスタの個数を増やすことで加算の筆算のスクラッチパッドをソフト乗算用に拡張している．

ここで Scratch-Pad の OUTPUT レジスタ 0 4 から出力する観測情報としては以下の 2 通りが考えられる．

1. OUTPUT0 4 の各注目座標
2. OUTPUT0 4 のうちの 1 つの各注目座標

前者の方が後者よりも先行研究のスクラッチパッドに近く，Scratch-Pad の注目座標を移動させるためのサブプログラム PTR の動作も先行研究のものと同等になるため，NPI の学習器のサブプログラムにより適しているように思えるかもしれない．しかし，前者の実装方法は ARG1 の桁数により Scratch-Pad からの観測情報の個数が変化してしまうため，後者よりも拡張性に欠ける．さらに学習器への入力が増加すると，Scratch-Pad の学習器の訓練コストの削減の効果が薄れてしまう．このことを考慮し，予備実験では後者の方法で観測情報を出力している．

6.2.3　サブプログラム

サブプログラムは先行研究の加算の筆算のものを踏襲しつつ，拡張している．表 6.1 にそれらを示す．予備実験における Scratch-Pad の出力段は ARG1 の桁数だけ存在するため，PTR サブプログラムは注目座標を横だけでなく上下にも移動できなければならない．しかし加算の筆算のスクラッチパッドにおいて，PTR サブプログラムは左右方向に注目座標を移動することしか想定されていない．そこで，予備実験では左右に加えて上下方向にも対応できるように PTR サブプログラムの機能拡張を行った．

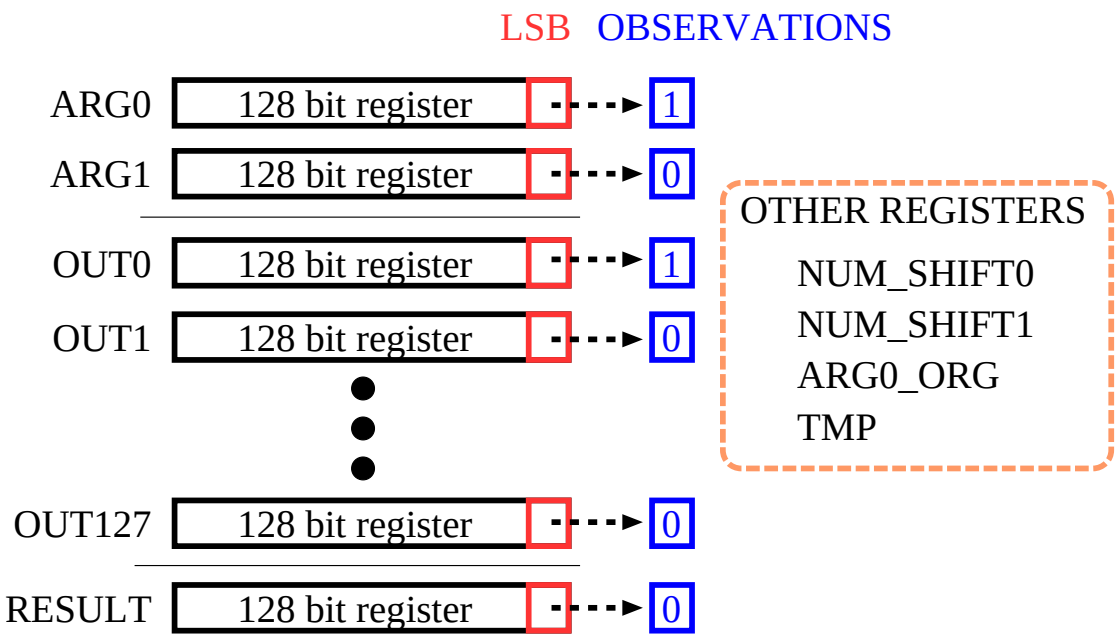

図 6.2: 提案手法におけるスクラッチパッド.

6.2.4　実験結果

上記の Scratch-Pad とサブプログラムを用いてソフト乗算の学習を行ったところ，学習が安定せず最もうまくいった場合でも訓練データのみを用いた Validation の精度が 70% という結果となった．著者は 10 進の乗算でキー値メモリが学習するパターン数が加算と比べて多くなり，シーケンシャルメモリに入力される特徴の変化が大きくなってしまったのがこの結果の原因ではないかと考えた．

6.3　提案手法

本節では 3 章で説明した Neural Programmer-Interpreters の学習器に対して，RISC-V アーキテクチャの命令セットを即時サブプログラムとして用いて訓練を行うことで，マシンコードを生成できるかどうかの検証を行う．もし，学習器が与えられた命令を用いて新しい命令を学習することができれば，このことが確認できる．

6.3.1　学習の対象

予備実験より，10 進数を用いたソフト乗算は NPI の学習器にとって難しすぎるという結論になっている．そこで，本節では Scratch-Pad の内部表現を 10 進数から 2 進数にしてソフト乗算を NPI の学習器を実現することを試みる．

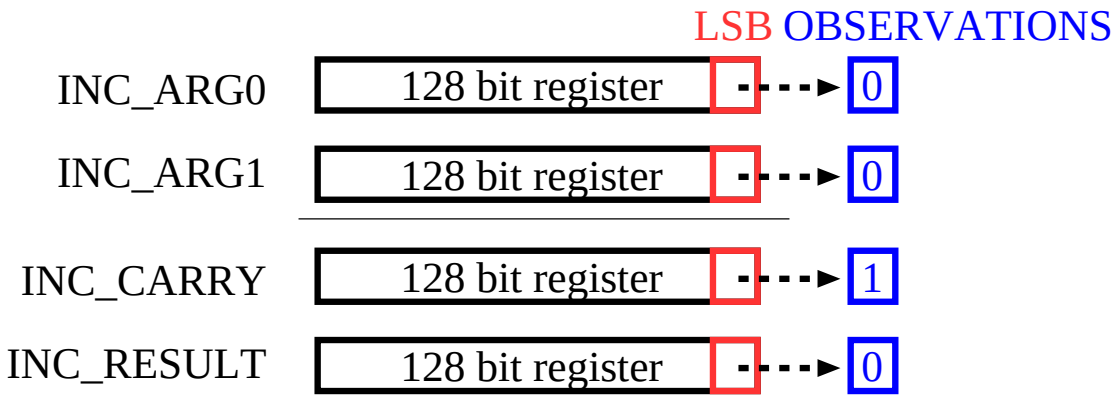

図 6.3: INCREMENT と DECREMENT のタスクで用いるレジスタ

NPI の学習器を用いて RISC-V アーキテクチャ向けのマシンコードを生成するためには，即時サブプログラムが対応する命令と同様にふるまわなければならない．それに加え，Scratch-Pad は RISC-V 命令セットを考慮したものでなくてはならない．そこで，提案手法では Scratch-Pad の再設計を行う．

6.3.2　Scratch-Pad

Neural Programmer-Interpreters の元の実装では加算やソーティングのタスクを行うために 10 進表現のスクラッチパッドを用いている [50, 51]．古いアーキテクチャの計算機では内部で 10 進数を直接扱うこともあったが [76]，今日において一般的なプロセッサでは内部的にバイナリ表現を用いている．また，予備実験より 10 進数を内部のデータを表現した Scratch-Pad を用いた学習は失敗に終わることがわかっている．そのため本稿のタスクには，内部的に 10 進表現を用いたスクラッチパッドは不向きである．また，スクラッチパッドにおける注目座標は PTR サブプログラムを注目座標における情報は WRITE サブプログラムを用いて更新されるが，一般的なプロセッサはレジスタやメモリを計算に用いるため，これらのサブプログラムもマシンコードの生成に用いることはできない．

マシンコードの生成を行うため，スクラッチパッドはレジスタやメモリがワードやバイト単位で参照されることを考慮し，設計される必要がある．そこで，提案手法では図 6.2，6.3 に示すようなスクラッチパッドを用いる．これらのスクラッチパッドの各行は 128bit のレジスタとしてふるまう．

このスクラッチパッドにおける即時サブプログラムはレジスタのアドレスや即値を引数として与えられると，スクラッチパッド内のレジスタの情報の更新を行う．そして，スクラッチパッドからは各スクラッチパッドの最下位ビットの情報とゼロフラグの出力を行う．これらの情報はレジスタから自然に取れるものとなっている．

そして，提案手法ではソフト乗算を 1 つの学習器のみで訓練 · 推論すると，ビット単位の演算を行っていることが原因でタスクの実行に必要なステップ数が従来の手法よりも増大してしまいうま

く訓練が収束しないことを確認している．そこで提案手法では訓練コスト削減のため，ソフト乗算を以下の 4 つのサブタスクに分割している．

1. MULTIPLICATION
2. ADDITION
3. INCREMENT
4. DECREMENT

MULTIPLICATION と ADDITION は図 6.2 のレジスタを用いて，INCREMENT と DECREMENT は図 6.3 のレジスタを用いて行われる．これらのタスクでは各レジスタは表 6.2 に示すような役割をする．

表 6.2: 各レジスタの用途

レジスタ名	用途
ARG0, ARG1	タスクの引数
ARG0_ORG	ARG0 の元の値
NUM_SHIFT0	ARG0 がどれだけ右にシフトしているか
NUM_SHIFT1	ARG1 がどれだけ右にシフトしているか
OUT0~127	MULTIPLICATION タスクの出力値
RESULT	ソフト乗算の最終的な結果が格納される
TMP	計算時に一時的利用されるレジスタ
INC_ARG0	INCREMENT，DECREMENT タスクの引数
INC_ARG1	1 ビットだけ HIGH になっており，最下位ビットが 1 のときは INCREMENT，DECREMENT タスクの開始もしくは終了を表す.
INC_CARRY	INCREMENT，DECREMENT タスクにおける桁上がりを表す
INC_RESULT	INCREMENT，DECREMENT タスクの結果が格納される

6.3.3　即時サブプログラム

即時サブプログラムは Neural Programmer-Interpreters の学習モデルが出力するプログラムの最小単位であり，スクラッチパッドを更新するために用いられる．シーケンシャルモデルはこれらのサブプログラムを適切に選択し，正しい順番で並べることでタスクを解くためのプログラムを構築する．提案手法のソフト乗算のタスクでは表 6.3 のようなサブプログラムが与えられる．

6.3.4　サブタスク

本手法ではソフト乗算を以下の 4 つのサブタスクに分割している．

a. MULTIPLICATION

表 6.3: 即時サブプログラムとその動作

即時サブプログラム	動作
OR	レジスタ値や即値を引数とした論理和演算
ST	レジスタ値や即値をレジスタに格納
SLL	レジスタ値を左にシフト
SRL	レジスタ値を右にシフト
ROR	レジスタ値を右に回転シフト

b. ADDITION

c. INCREMENT·DECREMENT

以降ではこれらのタスクを訓練する際に与えた実行例とこれらを構成するサブプログラムについて述べる．

MULTIPLICATION はソフト乗算の最初に実行されるタスクであり，全タスクのエントリーポイントにあたるタスクである．本手法の実装ではキー値メモリの訓練コストを抑制するために，ソフト乗算をビット演算で実現している．このタスクでは ARG0，ARG1 レジスタの各位のビットの論理積の結果を図 6.2 内の OUT レジスタに格納する．このタスクは以下の手順で実行される

a. 1 ビットの論理積演算を行う MUL1 を呼び出し

b. ARG0 レジスタを右にシフト

c. ARG0 レジスタが 0 なら ARG1 レジスタを右にシフト

d. ゼロフラグが 1 でなければ a に戻る

MUL1 では ARG0 と ARG1 の最下位ビットとそれに対するキー値メモリの出力を基に，論理積の演算を行う．論理積の結果は ARG0 レジスタのシフト数を基に左にシフトされた状態で OUT レジスタに格納されるが，その最に ST 命令を用いてしまうと OUT レジスタの値が MUL1 命令の出力値で置き換わってしまう．そのため，ここでの出力値は OUT レジスタの前に TMP レジスタに格納し，それを左にシフトしてから OUT レジスタとの論理和をとることによって OUT レジスタを 1 桁毎に更新している．MUL1 の処理が終わると，b〜d の通りに処理が進んでいく．

ADDITION はソフト乗算の後段の加算を行う処理であり，MULTIPLICATION タスクで書き込まれた OUT レジスタを 2 つずつ足していく．このタスクは以下の手順で実行される．

a. NUM_SHIFT1 レジスタの値を基に OUT レジスタの中で最も下にあるレジスタとその一つ上のレジスタの値を ARG0，ARG1 レジスタに格納

b. ARG0，ARG1 レジスタに値を格納した OUT レジスタを 0 で初期化する

c. ADD1 命令で ARG0，ARG1 レジスタの加算を 1 ビット毎に行い，a で参照した OUT レジスタのうち，上段にあるレジスタに結果を格納していく

表 6.4: INCREMENT タスクにおけるキー値メモリの理想的な入出力

INC_ARG0	INC_ARG1	INC_CARRY	carry_o	out
0	0	0	0	0
0	0	1	0	1
0	1	0	0	1
0	1	1	1	0
1	0	0	0	1
1	0	1	1	0
1	1	0	1	0
1	1	1	1	1

表 6.5: DECREMENT タスクにおけるキー値メモリの理想的な入出力

INC_ARG0	INC_CARRY	carry_o	out
0	0	0	0
0	1	1	1
1	0	0	1
1	1	0	0

d. ARG0，ARG1 レジスタの値と ADD1 における桁上がりが 0 になったら，NUM_SHIFT1 レジスタの値をデクリメント

e. NUM_SHIFT1 レジスタの値が 0 でなければ a に戻る

ADD1 では 1 ビット毎の加算つまり半加算を行い，その結果を MUL1 と同様に TMP レジスタに格納する．こちらも ARG0，ARG1 レジスタのシフト数だけ上位のビットの演算結果となるため，それに応じて TMP レジスタの値を左にシフトする．そして，OUT レジスタと TMP レジスタとで論理和演算を行い，結果を格納する．さらに ARG0，ARG1 レジスタを右にシフトし，NUM_SHIFT0 レジスタをインクリメントすることで，2 つのレジスタのシフト数を記録する．

e まで処理を行い，NUM_SHIFT1 が 0 になると，加算の結果が OUT レジスタのひとつ手前のアドレスに位置する，RESULT レジスタに加算の結果が書き込まれている．するとこのタスクも終了し，ソフト乗算全体も終了する．

INCREMENT，DECREMENT タスクは MULTIPLICATION，ADDITION タスクにおけるレジスタのシフト数を管理するためのサブタスクであり，それぞれ表 6.4 および表 6.5 の入出力を学習したキー値メモリを用いることで半加算と半減算を行っている．これらのタスクでは図 6.4 にあるように，ROR 命令を使ってレジスタ値を 1 ビットずつ読み込むことで加算と減算を実現しているが，ここでタスクの終了条件が問題となる．先行研究 [50] の 10 進数の加算のタスクではカーソルが指している位置の文字すなわちスクラッチパッドからの出力が空白であるというのを終了条件としていた．しかし，今回のタスクにおけるスクラッチパッドは CPU を再現したものであり，空白文字を扱うことはできない．そこで，INC_ARG1 レジスタを 1 ホットにして ROR で回転させ，INC_ARG1 の最下位ビットが 1 のときはタスクの開始状態もしくは終了状態と定義し，さら

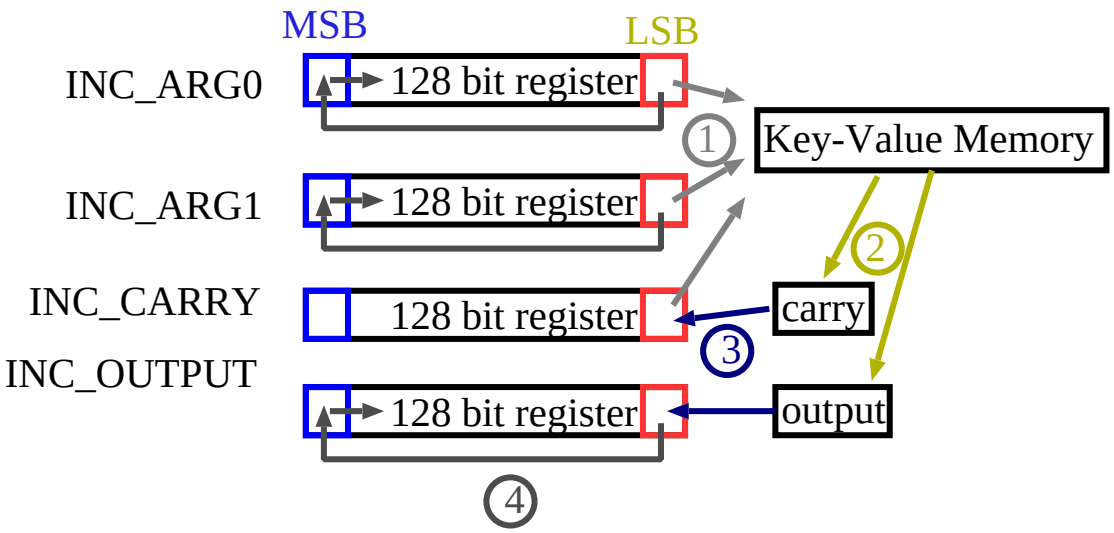

図 6.4: INCREMENT，DECREMENT タスクの流れ

に INC_CARRY レジスタの中身が 0 であれば演算終了状態であると定義した．

これらのタスクにおける流れはほぼ同じであるため，INCREMENT タスクの流れのみを示す．まず，INC_ARG0 レジスタに加算対象のレジスタの値を格納し，INC_ARG1 と INC_CARRY を 1 に，INC_OUTPUT を 0 に初期化する．さらに，サブプログラム INCREMENT1 を用いて 1 ビットの半加算を行う．ここで，表 6.4 に示したキー値メモリの出力のうち carry_o を INC_CARRY に，out を INC_OUTPUT に格納する．INCREMENT1 が終了すると，INC_ARG0，INC_ARG1，INC_OUTPUT レジスタが右に回転シフトされ，INC_ARG1 レジスタの最下位ビットが 1 かつ INC_CARRY レジスタの値が 0 ならタスクを終了し，そうでなければ INCREMENT1 が再び実行される．

6.3.5　評価

本節ではモデルの訓練および評価に用いたデータセットと評価に用いた環境について述べる．最後に，学習したモデルの評価を行う．

6.3.5.1　訓練データセット

訓練データセットは例外的なパターンを考慮して作られる必要がある．例えば，MULTIPLICATION や ADDITION で答えた 0 になる問題では 1bit の論理積や論理和をとるサブプログラムが実行例に含まれない．例外的な問題にも対応できるようなモデルを構築するために，訓練セットには 40 問をこのような例外的なパターンを含む，5bit 以内の数値のかけ算 120 問の学習用データを用いる．

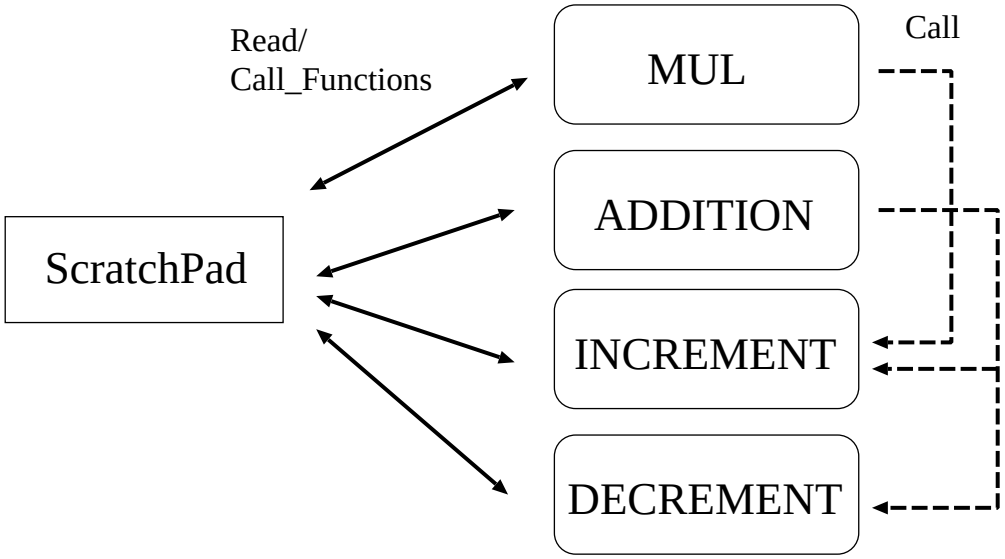

図 6.5: 評価環境

一方で，INCREMENT および DECREMENT タスクには例外的なパターンは存在せず，サブプログラムも引数を除けば同一である．そこで訓練データとしてそれぞれ，0 から 9 のインクリメントと 1 から 10 のデクリメントの学習用データを用いる．

6.3.5.2　訓練環境

本手法において，学習モデルは 4 種類し，それぞれ MULTIPLICATION，ADDITION，INCREMENT および DECREMENT のサブタスクを訓練し，お互いに連携してソフト乗算のプログラムを生成する．そのため，それぞれのタスクを同時に学習するのは困難である．そこで，提案手法では訓練時に学習対象を除いて理想的に動作するプログラムを用いる．

6.3.5.3　学習結果

計算結果のオーバーフローを防ぐため，評価は 64 ビット以内の引数同士の計算で行う．ここで，今回実装したソフト乗算はレジスタをビット毎に計算するため，レジスタに格納されうるあらゆるデータを用いて評価するのが理想である．しかし，NPI はサブプログラムを推論する度に学習器を用いるため，全パターンを用いた評価は時間的に現実的ではない．そこで，著者は NPI が長いプログラムを短いプログラムの延長として生成することに着目し，評価には図 6.6 のようにさまざまなビットパターンを含んだテストデータセットを用いることにした．図 6.6 では今回テストするパターンの最大長である 64bit のデータを 11 分割し，それぞれの区間を同時に加算することで生成している．

MULTIPLICATION，ADDITION タスクを学習したモデルは自身の実行中に INCREMENT，DECREMENT タスクを学習したモデルを何度も呼び出す．そのため評価用データセットを減らし

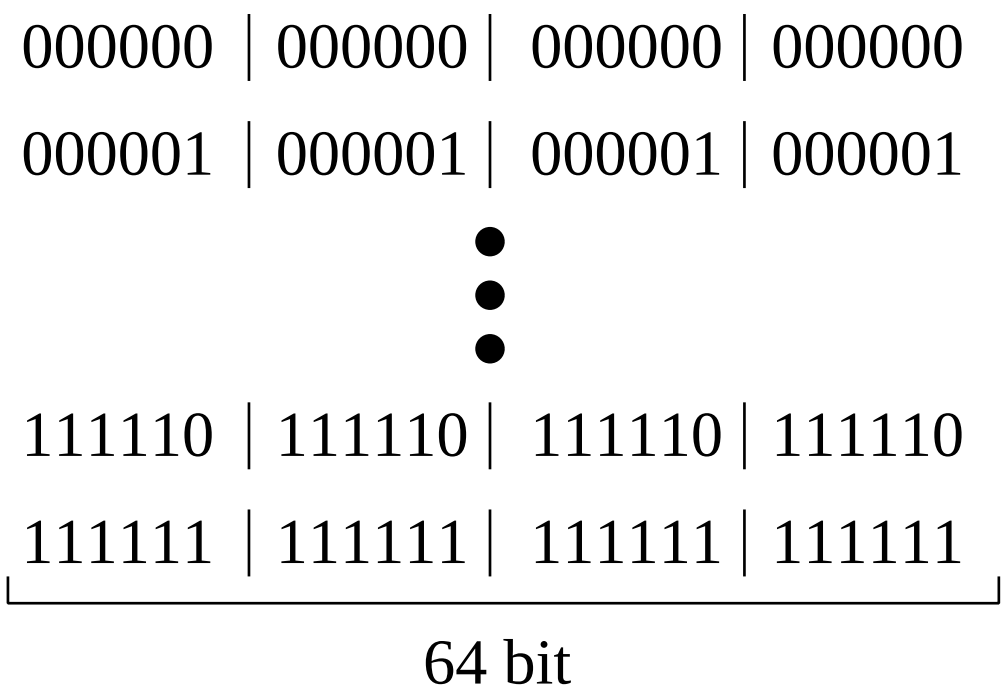

図 6.6: 評価データセットの作成方法

たとしても，4 つのモデルを同時に評価すると莫大な実行時間がかかってしまう．そこで，INCREMENT と DECREMENT は別で評価を行うことにした．ここで，MULTIPLICATION，ADDITION タスクに与えられる引数は 64bit 以内であるため，MULTIPLICATION および ADDITION タスクを実行するモデルが正常に動作している限り，INCREMENT，DECREMENT のタスクに与えられる引数はそれぞれ 0 から 63 と 1 から 64 である．そこで，これら 2 つのタスクを行う際の引数はそれぞれ 0 から 99 と 1 から 100 とした．

上記の条件で評価を行った結果，MULTIPLICATION，ADDITION の評価，INCREMENT，DECREMENT の評価それぞれで正解率が 100%であった．MULTIPLICATION，ADDITION の訓練データが 5bit 以内，INCREMENT，DECREMENT の訓練データが 4bit 以内であったことから，かなりの汎化性能を示しているといえる．

6.4　まとめ

本章では RISC-V で実行可能なマシンコードの生成を目的とし，RISC-V アーキテクチャの命令セットをベースとした即時サブプログラムから構成される訓練データとそれらを解釈可能な Scratch-Pad を用意し，NPI の学習器がそれらを用いて乗算命令を用いない，ソフト乗算を行うプログラムを生成するように訓練を行った．

提案手法に先立ち 10 進数でソフト乗算を行うための Scratch-Pad とサブプログラムを用意し学習器に対して訓練を行ったが，学習が安定せず精度も訓練で用いたデータセットに対して最大で

70% 止まりとなった．

提案手法ではこの結果を踏まえて学習器の訓練に用いられるサブプログラムのうち即時サブプログラムを RISC-V の命令セットで構成し，それらの命令セットが実際の CPU 上と同様に動作するように Scratch-Pad の実装を行った．また，ソフト乗算のタスク全体を 1 つの学習器に学習させようとしたところ学習器への入出力が長くなりすぎてしまい，うまく学習が収束しなかった．そこで，ソフト乗算のタスクを MULTIPLICATION，ADDITION，INCREMENT，DECREMENT の 4 つに分割することでこの問題に対処した．

提案手法を用いて NPI の学習器を訓練し評価を行なった．学習器の訓練に用いたデータセットは MULTIPLICATION および，ADDITION においては 5bit 以内のソフト乗算を開始プログラムとするデータ 120 個，INCREMENT および DECREMENT においてはそれぞれ 0 から 9 と 1 から 10 を引数とした加減算のデータを用いた．また，評価は実行時間の関係から MULTIPLICATION および ADDITION を一緒に行い，INCREMENT および DECREMENT は別々に評価を行った．64 bit 以内のソフト乗算の問題を用いて MULTIPLICATION および ADDITION の評価を行ったところ，精度は 100% であった．また INCREMENT および DECREMENT においては，MULTIPLICATION および ADDITION が正常に動いている範囲で渡される引数より少し余裕をもたせた 0 から 99 と 1 から 100 をそれぞれ引数として渡して評価を行ったところ，いずれも精度は 100 % であった．

以上のことから，MULTIPLICATION，ADDITION の訓練データが 5bit 以内，INCREMENT，DECREMENT の訓練データが 4bit 以内であったことから，かなりの汎化性能を示しているといえる．

第7章　Neural Programmer-Interpreters を用いたコード移植器の構築における問題の解決

7.1　はじめに

6 章では NPI を用いたマシンコード生成手法の提案を行ったが，これは NPI の実行単位として既存の CPU の命令セットを用いてタスクを学習することであるマシンで走っているプログラムを元に，別のアーキテクチャのマシン向けの実行形式を直接生成できるようなプラットフォームを構築できるのではないかというアイデアを確かめるためのものである．そのため 6 章ではマシンコードの生成を主たる目的とし，モデルの訓練サンプルおよび出力共に RISC-V の命令セットを用いて構成されている．

本章では 6 章の手法をベースに実際に学習時とは別の計算機向けの命令セットを生成する学習器を得る．このようなコード移植器を実現するためには，6 章において学習器の出力の中における即時サブプログラムのかわりに，目的のアーキテクチャに対応する即時サブプログラムを出力するように学習器を Fine-Tuning する必要がある．しかし，そこにはいくつかの課題があった．

7.2　コード移植器の実現における障壁

本節では NPI を用いて Transcoder を構築する際の問題を明かにし，それらに対する解決策を提案する．x86_64 アーキテクチャ向けのマシンコードを用いて RISC-V 向けのマシンコードの生成を行うためには，x86_64 と RISC-V の両方のアーキテクチャの命令を扱うことのできる Scratch-Pad が必要である．それに加えて，NPI の学習器が x86_64 アーキテクチャのかわりに RISC-V アーキテクチャのマシンコードを出力するように，訓練データを用いて Fine-Tuning する必要がある．x86_64 アーキテクチャ向けのマシンコードの生成を行う NPI の学習器が RISC-V 向けのマシンコードを生成できるようにしうるためには，RISC-V アーキテクチャの命令からなる訓練データが必要不可欠である．NPI における即時サブプログラムはタスク内のさまざまな箇所で用いられるため，さまざまなケースを網羅するような訓練データが必要となる．しかしながら，そのようなデータセットを用いて学習器を訓練するのであれば，はじめから RISC-V の命令セットで構成されたプログラムを出力するように学習を構築するのとかわらない．

そこで，以降ではコード移植器を実現するための新しい Scratch-Pad，訓練手法および訓練データの提案を行う．

7.3　Scratch-Pad

6 章の手法は RISC-V 命令のみに対応した Scratch-Pad を用いて実現されているが，本節で提案するコード移植器の実現には RISC-V·x86_64 の両方のアーキテクチャに対応した Scratch-Pad が必要となる．そこで，まずはコード移植器の実現に用いた Scratch-Pad について述べる．

コード移植器は 6 章の手法を拡張して行うため，ここで用いる Scratch-Pad は RISC-V 向けの Scratch-Pad と同様に，プロセッサの内部を再現したものが望ましい．そこで，新しい Scratch-Pad も同様に各行はそれぞれレジスタとしてふるまい，Scratch-Pad に対して即時サブプログラムが与えられると，即時プログラムと対応する命令と同様のレジスタ操作を行う．

今回新しく対応させる x86_64 命令は訓練および評価に用いる計算機のアーキテクチャと同一である．そこで即時サブプログラムを，インラインアセンブラを用いて実現することで，6 章の手法よりも厳密な評価を行う．インラインアセンブラを用いることの弊害として，命令の実用上の制約を考慮しなければならないことである．今回の評価を行うにあたっては 64bit のレジスタまでしか引数としてとれない命令があるため，スクラッチパッドの各行のレジスタは 64bit で構成することとする．

また，Scratch-Pad が出力する情報は 6 章のものと同様に特定のレジスタの最下位ビットや，計算過程のゼロフラグといった，レジスタから自然にとれる情報とする．

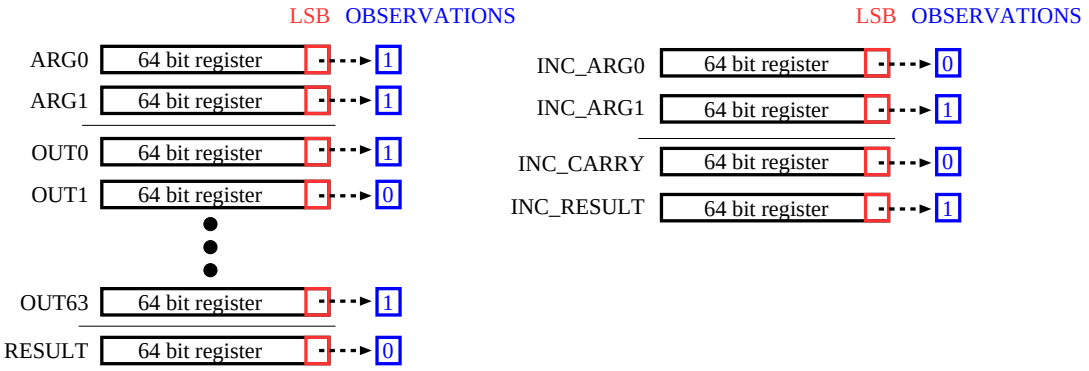

図 7.1: Scratch-Pad

7.4　訓練データ

NPI の学習器に対して Fine-Tuning を行うにあたり，どのような訓練データを与えるかが課題となる．例えば，6 章の提案手法と同様の方法で訓練したモデルに対して Fine-Tuning を行う際，プログラム中で置換対象の即時サブプログラムを呼び出している部分における入出力を抽出し，同じ入力に対して異なる出力を行うように訓練したとしてもうまくいかない．これは NPI が時系列情報を内部に保持しているからであり，仮のそのような方法で訓練した場合，学習器の時系列情報をリセットして最初に当該の入力を行った際は理想的な出力が得られるが，タスクを解くためのプログラムを得るためにタスクの開始命令から順に学習器の出力を得た場合は再学習前の出力が得ら

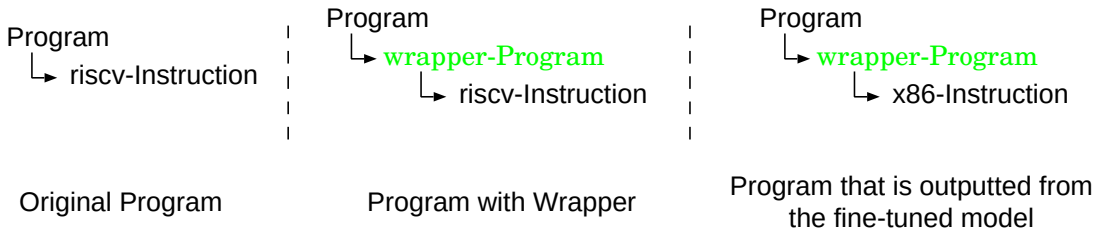

図 7.2: Wrapper サブプログラム

れてしまう．そのため，NPI の学習器に対して Fine-Tuning を行う際には完全なタスクの実行例が必要となる．このような訓練データを構築するためには，別のアーキテクチャ向けの訓練データを構築し直すのとほぼ同等の労力が必要である．

また，この問題を解決したとしても，即時プログラムを別のアーキテクチャの命令に対応するものに置き換える際にも問題が発生する．例えば，x86_64 命令セットでレジスタ値を用いて別のレジスタに対してシフト命令を実行する場合，シフト命令の制約からレジスタ値を一度カウンタレジスタに格納しなければならない．そのため，レジスタ値を用いたシフトは RISC-V アーキテクチャでは 1 命令で実行できるものの，x86_64 アーキテクチャでは 2 命令かかってしまうため，この命令を単純に置き換えることは困難である．

これらの問題に対処するため，本節では図 7.2 に示すように訓練データ内で即時命令を呼ぶ際に即時命令の Wrapper サブプログラムを追加する．これにより，即時命令が呼び出される際の入力や時系列情報をいつでもほぼ同じになり，再学習の際に即時命令を置き換えることが可能となる．

7.5　学習手法

以降では新たに実装した Scratch-Pad を用いて学習器の訓練を行う．はじめに提案手法に先立って行った予備実験について述べる．

7.5.1　予備実験

予備実験では以下のステップで学習器の訓練を行なった．

a. x86_64 命令セットで構成された訓練データを用いた学習

b. Wrapper サブプログラムを開始命令とする訓練データを用いた学習

最初の学習では，新たに実装した Scratch-Pad と x86_64 命令セットで構成された訓練データを用いて学習を行う．この際，訓練データにおける即時命令は Wrapper サブプログラムを通して呼び出している．最初の学習が終わると学習器の出力を RISC-V 命令で構成されたプログラムにするため，Wrapper サブプログラムを開始命令として RISC-V 命令を呼び出す訓練データを入力として与え，Fine-Tuning を行った．ここで Wrapper サブプログラムを開始プログラムのみを学習

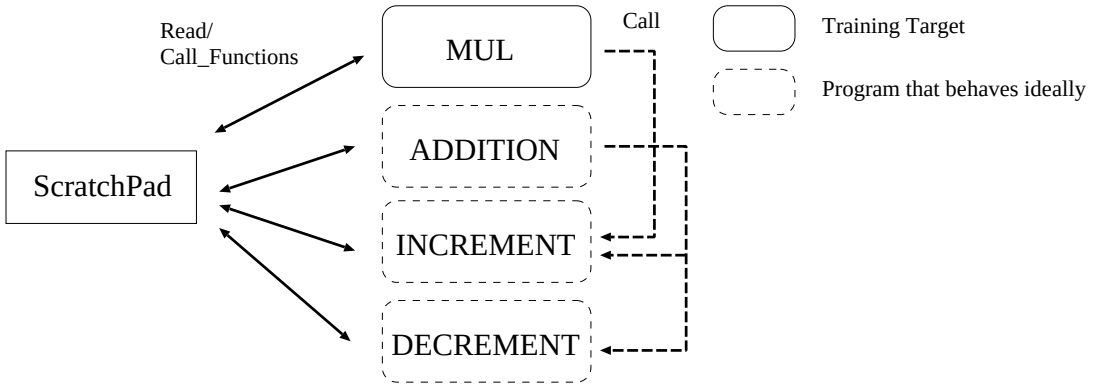

図 7.3: 評価環境

器与えてしまうと，学習器が本来の学習対象であるソフト乗算の方法を忘却してしまうことがあった．そこで，Fine-Tuning では最初の学習で与えたデータも混ぜて与えた．

このような方法で学習器を訓練した結果，開始プログラムが Wrapper サブプログラムである場合は RISC-V の命令を出力するプログラムが得られたものの，開始プログラムがソフト乗算の開始命令である場合は x86_64 の命令を出力するプログラムが得られた．著者は，この問題の原因が学習器のレイヤーのひとつである，シーケンシャルモデルの内部状態にあると考えた．なぜなら，シーケンシャルモデルは学習器に入力された情報だけでなく，これまでに観測された情報を考慮して，出力を決定することを可能にしているためである．

7.5.2　学習器の訓練

ここでは 7.5.1 で述べた予備実験の内容をふまえ，提案手法で行う学習器の訓練について述べる．

提案手法の訓練も予備実験と同様に 2 回行う．2 回のうち最初の訓練は予備実験と同様であるが，2 回目の訓練において Wrapper サブプログラムを開始プログラムとする訓練データは予備実験の結果から，NPI の学習器の内部状態を考慮して与えなければならない．一方で，Wrapper サブプログラムを開始プログラムとするデータにおいて即時プログラムに与えられる引数や呼び出される即時サブプログラムは Wrapper サブプログラムが呼ばれるまでのプログラムの呼び出しや観測情報に依らず Wrapper サブプログラムに与えられた引数のみで決まる．そこで，2 回目の Fine-Tuning においては Wrapper サブプログラムを開始プログラムとするデータはあらかじめ生成され，ソフト乗算用の訓練データを学習モデルに入力する際に Wrapper サブプログラムが呼び出された際に埋め込まれる．

7.6　評価

本節では x86_64 から RISC-V 向けのコードを移植する，コード移植器モデルの学習環境について述べ，最後に学習したモデルの評価を行う．

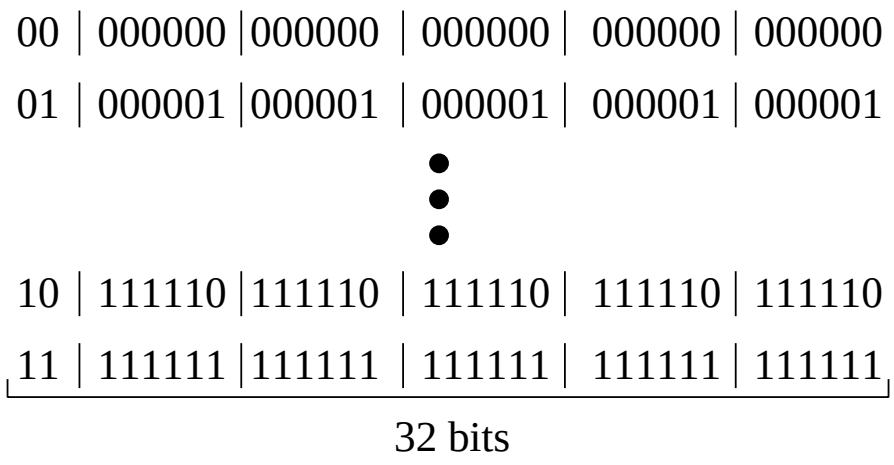

図 7.4: 評価データセットの作成方法

7.6.1　タスク

提案手法では 6 章で述べた手法と同様にソフト乗算を MULTIPLICATION，ADDITION，INCREMENT，DECREMENT の 4 つに分割して，7.5.2 節で述べたような方法で訓練を行った．しかし，ADDITION タスクの最初の学習が収束しなかった．そこで，本節では MULTIPLICATION タスクを学習したモデルに絞り，モデルが生成したプログラムを分析することで評価を行う．

7.6.2　訓練および評価環境

MULTIPLICATION タスクを学習したモデルは ADDITION，INCREMENT および DECREMENT を学習したモデルと連携してタスクを解くが，今回評価するのは MULTIPLICATION タスクを学習したモデルのみである．そこで，訓練および評価の両方で他の 3 つのモデルは理想的にふるまうものとして評価を行う．

7.6.3　評価結果

7.3 で述べたように，Scratch-Pad は x86_64 命令の制約から 64bit 幅で設計されている．そこで計算結果のオーバーフローを防ぐため，本章の評価は 32bit 以内の数値同士のソフト乗算で行う．

6 章の実装と同様に，本章の実装もソフト乗算をビット毎に行っている．そのため，図 7.4 の示すように，6 章における評価と同様の評価データセット生成手法を 32bit で行った．

生成したデータセットを用いて評価を行ったところ，最終的な値が 0 になる入力に対しては正しい結果を出力できるが，その他は誤った出力をするという結果になってしまった．そこで本節の手法の評価では，評価データセットの実行中に Wrapper サブプログラムに対するモデルからの出力を分析し，さらに Fine-Tuning 前のモデルによる出力と比較することで評価を行う．

7.6.3.1　Store Wrapper サブプログラム

図 7.5 は提案手法の訓練のうち最初の訓練を行った後に，図 7.6 は Fine-Tuning を行った後に学習器が STORE の Wrapper サブプログラムに対して即時サブプログラムを出力している様子である．STOREI 及び MOVI はレジスタに即値を格納する即時サブプログラム，STORE 及び MOV はレジスタに他のレジスタ値を格納する即時サブプログラムである．

これらの図から，Wrapper サブプログラムが呼び出す命令については Fine-Tuning することができているが，それらの引数はうまく行えていないことがわかる．

```
STOREI Wrapper,<IA: [1, 0, 2, 0]>
  ⇒MOVQI,<IA: [10, 1, 2, 0]>
STOREI Wrapper,<IA: [1, 0, 2, 0]>
  ⇒MOVQI,<IA: [11, 0, 2, 0]>
STORE Wrapper,<IA: [1, 0, 2, 0]>
  ⇒MOVQ,<IA: [8, 2, 2, 0]>
STORE Wrapper,<IA: [13, 1, 2, 0]>
  ⇒MOVQ,<IA: [2, 11, 2, 0]>
```

図 7.5: x86_64 向けに訓練された学習器の STORE Wrapper サブプログラムに対する出力

```
STOREI Wrapper,<IA: [5, 2, 1, 2]>
  ⇒STOREI,<IA: [1, 0, 1, 2]>
STOREI Wrapper,<IA: [5, 2, 1, 2]>
  ⇒STOREI,<IA: [9, 1, 1, 3]>
STOREI Wrapper,<IA: [13, 2, 1, 2]>
  ⇒STOREI,<IA: [12, 2, 0, 2]>
STORE Wrapper,<IA: [1, 11, 1, 3]>
  ⇒STORE,<IA: [2, 11, 1, 3]>
STORE Wrapper,<IA: [5, 2, 1, 2]>
  ⇒STORE,<IA: [5, 0, 1, 2]>
STORE Wrapper,<IA: [5, 2, 1, 2]>
  ⇒STORE,<IA: [5, 12, 2, 2]>
```

図 7.6: RISC-V 向けに Fine-Tuning された学習器の STORE Wrapper サブプログラムに対する出力

7.6.3.2　Shift Wrapper サブプログラム

図 7.7 は提案手法の訓練のうち最初の訓練を行った後に，図 7.8 は Fine-Tuning を行った後に学習器が SHIFT の Wrapper サブプログラムに対して即時サブプログラムを出力している様子である．SRLI，SARQI は即値を用いてレジスタ値を右にシフトする即時サブプログラム，SALQ，SLL はレジスタ値を用いてレジスタを左にシフトする即時サブプログラムである．また，図中の MOVQ プログラムはレジスタ値をカウンタレジスタに格納している．

SLL がシフト数を表すレジスタとしてあらゆるレジスタを用いることができるのに対し，SALQ はカウンタレジスタと呼ばれる特別なレジスタのみを用いることができる．そのため，x86_64 ではレジスタ値を用いた左シフトに 2 サイクルかかるのに対し，RISC-V ではこれを 1 サイクルで実行することができる．このことから，このサブプログラムは単純に置換ができないにも関わらず，呼び出す即時サブプログラムに関しては正しく置換できている．しかし，Store サブプログラムと同様にモデルが引数を十分に学習できておらず，再学習後のモデルで正しい答えが出力できない原因となっている．

```
SRLI Wrapper,<IA: [3, 1, 2, 0]>
  ⇒SARQI,<IA: [6, 1, 2, 0]>
SLL Wrapper,<IA: [1, 0, 2, 0]>
  ⇒MOVQ,<IA: [2, 0, 2, 0]>
    SALQ,<IA: [13, 2, 2, 0]>
```

図 7.7: x86_64 向けに訓練された学習器の SHIFT Wrapper サブプログラムに対する出力

```
SRLI Wrapper,<IA: [1, 11, 1, 3]>
  ⇒SRLI,<IA: [6, 6, 1, 2]>
SLL Wrapper,<IA: [5, 0, 1, 2]>
  ⇒SLL,<IA: [13, 2, 13, 2]>
```

図 7.8: RISC-V 向けに Fine-Tuning された学習器の SHIFT Wrapper サブプログラムに対する出力

7.6.3.3　OR Wrapper サブプログラム

図 7.9 は提案手法の訓練のうち最初の訓練を行った後に，図 7.10 は Fine-Tuning を行った後に学習器が OR の Wrapper サブプログラムに対して即時サブプログラムを出力している様子である．

これらの結果においては呼び出されている即時サブプログラムに与えられている引数については正しく再学習できているが，Wrapper サブプログラムに与えられている引数がどちらも正しくないという結果になった．

```
OR Wrapper,<IA: [1, 0, 2, 0]>
  ⇒ORQ,<IA: [13, 3, 2, 0]>
```

図 7.9: x86_64 向けに訓練された学習器の OR Wrapper サブプログラムに対する出力

```
OR Wrapper,<IA: [5, 2, 1, 3]>
  ⇒OR,<IA: [13, 2, 13, 3]>
```

図 7.10: RISC-V 向けに Fine-Tuning された学習器の OR Wrapper サブプログラムに対する出力

7.7　まとめ

本章では 6 章の手法をベースに NPI を用いてコード移植器を実現する際の問題点とそれを解決するための学習手法の提案を行った．

コード移植器を実現するにあたり NPI の学習器に対し，あるアーキテクチャのマシンコードで構成された訓練データを用いて学習を行い，さらに別のアーキテクチャのマシンコードで構成されたデータを用いて Fine-Tuning を行うという方法をとった．そこで，2 つのアーキテクチャ向けの命令を受け取り，適切に動作するように Scratch-Pad の拡張を行った．

Fine-Tuning を行うにあたり，訓練に用いるデータセットをどのように設計するかが問題となった．そこで，最初アーキテクチャのマシンコード生成に用いるデータに Wrapper サブプログラムを挿入することでこの問題に対処した．

本章の提案手法に先立ち最初の訓練に Wrapper サブプログラムを入れた x86_64 アーキテクチャ向けソフト乗算のプログラム全体を，Fine-Tuning に Wrapper サブプログラムを開始サブプログラムとする RISC-V アーキテクチャ向けプログラムを用いて学習器の訓練を行ったところ，ソフト乗算の開始サブプログラムを入力した場合は x86_64 アーキテクチャ向けのプログラムを，Wrapper サブプログラムを開始サブプログラムとした場合は RISC-V アーキテクチャ向けのプログラムを出力する学習器が得られた．

著者はシーケンシャルモデルが内部的にもつ時系列情報がこの結果の原因ではないかと考えた．そこで，提案手法では Fine-Tuning の際に Wrapper サブプログラムの訓練データをソフト乗算のプログラムの中に埋め込むようにして与えることにした．

提案手法を用いて学習器の訓練を行った結果，即時サブプログラムに関しては正しく学習しなおせたものの，その引数に関してはうまく学習できないという結果となった．また，6 章の 4 つのサブタスクのうち，ADDITION タスクの学習が収束しないという結果になった．

第8章　結論

本論文では，エッジコンピューティング向けに Informed-Filters の並列実装を移植するために Jetson AGX Xavier を用いた実装と，市販の UAV と Informed-Filters を連係して動作させるための実装の提案を行った．また，機械学習を用いてマシンコードを生成 · 移植する手法の提案を行った．以降ではこれらについてのまとめを行う．

著者が所属する研究室では，運動中の選手に装着されたセンサの間でネットワークを同席に構築し，スポーツシーンにおける生体情報取得のためのシステムを実現しようと試みている．そこで，研究室では画像情報に基いてセンサの位置関係を把握し，ネットワークを構築する Image-Assisted Routing(IAR) を提案しているが，そのためにはリアルタイムな人検出手法が必要であり，それには Informed-Filters とその並列実装が有望であるという研究がある．しかし，Informed-Filters の並列実装として提案されている手法ではプログラムを実行するためにデスクトップ用の強力な GPU を搭載したサーバマシンを用いているが，電力や場所の限られた屋外においてはその実装をそのまま用いることは困難である．また，ソフトウェアの最適化に関して，計算機のアーキテクチャに関する知識がなくても最適なプログラムを得られるようにするためにはコンパイラによるプログラムの最適化が重要であるが，今日のコンパイラで行われている典型的な最適化のみでは人手による最適化と比べて効果が薄いという問題もある．

これらの問題に対処するため，本論文では Informed-Filters の並列実装を NVIDIA Jetson Xavier，C++ AMP の 2 つのプラットフォームに移植し，深層学習でマシンコードを生成する手法および，コード移植器を実現する手法を提案した．

まず，4 章においては Informed-Filters の並列実装を UAV に搭載することができるように組込み環境に移植し，さらに実際に 4K カメラからシステムに画像を取り込むためにカメラの HDMI 出力を UVC に変換するアダプタの選定を行った．まず，Informed-Filters の並列実装を走らせる環境として，Image-Assisted Routing (IAR) のシステム要件を考慮して NVIDIA Jetson Xavier を用いることにし，HDMI を UVC に変換するアダプタとして AV.io 4K を用いることにした．提案したシステムを用いて実環境から得られた空撮画像に対して検出プログラムを動作させたところ，3840 × 2160 の解像度の画像に対して約 22 fps で動作可能であることが確認された．また，検出精度も閾値を調整することでおよそ見逃し率 1%で画像一枚あたりの誤検出数が 0.02 未満であることがわかった．この結果から UAV に搭載可能なサイズかつ省電力に動作するシングルボードコンピュータを用いることで IAR が動作するために十分な速度の人検出器の並列実装が実現可能であることがいえる．

さらに，5 章では UAV から画像を取得しそれに対してデスクトップ PC 向けの GPU を用いて検出処理を行うため，UAV から画像を無線経由で受け取りそれに対してリアルタイムに人検出を行うための実装を行った．著者は IAR に用いる UAV として DJI Phantom 4 Pro V2.0 を用いることにしたが，Phantom 4 Pro V2.0 から画像を受け取るためにはユニバーサル Windows プラット

フォーム (UWP) 向けに提供されている DJI Windows SDK を用いる必要がある．しかし，UWP ではセキュリティ上の関係からアプリケーションはすべてサンドボックス環境で実行されるため，CUDA 環境を用いることができないという問題があった．この問題を解決するための手段としては，UAV から画像を受信するプログラムとそれに対して人検出を行うプログラムを分割し，それらの間でソケット通信を用いて画像の送受信を行う実装が考えられる．この実装に先立ち，Unix 上でソケット通信を用いて Android アプリを経由してデータの受け渡しを行うシステムの実装を行ったが，通信速度が安定しないという問題があった．そこで本研究では UAV からの画像の受け取りから人検出システムまでを 1 つのプログラムで行うことにした．

著者は UWP 上で CUDA を用いたプログラムが動作しないという問題を解決するために，UWP 上で並列計算を行う手段である，C++ AMP を用いて Informed-Filters の並列実装の移植を行うことにした．検出器の実装を行い，UAV から無線経由で取得できる 1920 × 1080 の解像度の実環境から得られた空撮画像を用いて評価を行ったところ，検出精度は CUDA 版と比べて遜色ない精度が得られ，比較手法として用いた EfficientDet-D0 や RetinaNet と比べても高い精度が得られた．また，検出速度は Windows 上で動作させたときの CUDA 版と比べて 15 から 20ms 程遅い傾向にあるものの，最も時間のかかったフレームでも 40ms 以下となっており，テスト画像全体のフレームレートでは 41.8 fps であった．また，比較手法の EfficientDet-D0 や RetinaNet よりも高速に動作するという結果が得られた．このことから，C++ AMP を用いることで UAV と連携してリアルタイムに動作する人検出手法の並列実装が得られることがわかった．

そして，6 章において，著者は模倣学習の手法のひとつである Neural Programmer-Interpreters (NPI) がタスクをステップ毎に分割し，プログラムを構成するように解くことに着目した．そこで，著者は NPI におけるタスクのステップを CPU の命令として表現し，生成するプログラムの動作をトレースするようにコードを生成することで，マシンコードが生成可能ではないかと考えた．この着想が実現可能であることを確かめるために，NPI を用いてマシンコードの生成を行った．

マシンコードの生成を行うタスクのターゲットとして，乗算命令や加算命令を用いないソフト乗算を対象として実装を行うことにした．はじめは NPI の先行研究を参考に NPI の学習器の外部メモリである Scratch-Pad の内部表現を 10 進数にして実装することを試みたが，学習器に入力される数値の組合せが膨大になるためか学習が安定せず，訓練サンプルを用いた Validation でも精度が最大で 70% という結果になった．そこで Scratch-Pad の内部表現を 2 進数にすることで学習器に入力される組合せを削減し，さらに CPU の命令が実際の環境と同様に動作するように Scratch-Pad を改良することにした．すると今度はタスクで呼ばれる命令数が膨大すぎて，学習器 1 つでは訓練がうまくいかなかった．そこで 6 章の提案手法ではソフト乗算のタスクを 4 つに分割しそれぞれ別の学習器を用いて訓練を行い，タスク全体はそれらの学習器が連携して行うようにした．また，評価を網羅的に行うと現実的な時間で評価が終わらないため，NPI の学習器の性質を考慮しつつ評価用データセットになるべく多くのビットパターンが入るようなパターン生成手法を提案し，それを用いて評価を行うことにした．その結果，64 bit 以内のソフト乗算の問題を用いて MULTIPLICATION および ADDITION の評価を行ったところ，精度は 100% であった．また INCREMENT および DECREMENT においては，MULTIPLICATION および ADDITION が正常に動いている範囲で渡される引数より少し余裕をもたせた 0 から 99 と 1 から 100 をそれぞれ引数として渡して評価を行ったところ，いずれも精度は 100 % であった．これらのことから，MULTIPLICATION，ADDITION の訓練データが 5bit 以内，INCREMENT，DECREMENT の

訓練データが 4bit 以内であったことから，かなりの汎化性能を示しているという結論に至った．

最後に 7 章では 6 章で提案した NPI の学習器の訓練手法の拡張を行い，NPI を用いてコード移植器を実現する上での問題点を明かにし，それらの問題を解決するための手法の提案を行った．コード移植器を実現するにあたり NPI の学習器に対し，あるアーキテクチャのマシンコードで構成された訓練データを用いて学習を行い，さらに別のアーキテクチャのマシンコードで構成されたデータを用いて Fine-Tuning を行うという方法をとった．そこで，2 つのアーキテクチャ向けの命令を受け取り，適切に動作するように Scratch-Pad の拡張を行った．

Fine-Tuning を行うにあたり，訓練に用いるデータセットをどのように設計するかが問題となった．そこで，最初アーキテクチャのマシンコード生成に用いるデータに Wrapper サブプログラムを挿入することでこの問題に対処した．7 章で提案した手法先立ち最初の訓練に Wrapper サブプログラムを入れた x86_64 アーキテクチャ向けソフト乗算のプログラム全体を，Fine-Tuning に Wrapper サブプログラムを開始サブプログラムとする RISC-V アーキテクチャ向けプログラムを用いて学習器の訓練を行ったところ，ソフト乗算の開始サブプログラムを入力した場合は x86_64 アーキテクチャ向けのプログラムを，Wrapper サブプログラムを開始サブプログラムとした場合は RISC-V アーキテクチャ向けのプログラムを出力する学習器が得られた．著者はシーケンシャルモデルが内部的にもつ時系列情報がこの結果の原因ではないかと考えた．そこで，提案手法では Fine-Tuning の際に Wrapper サブプログラムの訓練データをソフト乗算のプログラムの中に埋め込むようにして与えることにした．

提案手法を用いて学習器の訓練を行った結果，即時サブプログラムに関しては正しく学習しなおせたものの，その引数に関してはうまく学習できないという結果となった．また，6 章の 4 つのサブタスクのうち，ADDITION タスクの学習が収束しないという結果になった．

ここまでの研究成果より，CUDA が実行可能な GPU が搭載されたシングルボードコンピュータや UWP といったプラットフォーム向けに人検出の並列実装の移植を行うことによって，人検出器を電力や場所の限られた屋外環境で実行し IAR に用いることが可能であることが示されている．また，これらの手法を用いた移植は Informed-Filters に限らず他の物体検出手法や意味論的領域分割手法にも応用可能であるため，画像認識手法をアプリケーションに応用する際に有用である．

また，NPI を用いたマシンコードの生成およびコード移植器実現手法における結果をそのまま実世界のアプリケーションに活かすことは難しい．特に，本稿で提案した手法で生成されるプログラムでは乗算を 1 ビット毎に計算しているため，実行に通常の計算に比べて何倍ものサイクル数を要す．そして，マシンコードの生成手法が 1 モデルで実装できていないこと，コード移植器の実装において ADDITION のタスクを行う学習器の訓練が完了しなかったこと，コード移植器の訓練手法で引数までは訓練できなかったことも解決しなければならない課題である．今後は提案手法で未解決な問題に対処するため，モデルの見直しや学習器の構造の調査，そして Scratch-Pad の改良を行っていく所存である．さらに，近年急速に発展している自然言語生成ベースの手法である GPT-3 や AlphaCode [77] と連携してそれらの手法で生成されたコードを最適化することができれば，ようやく実世界のアプリケーションで応用できるといえるだろう．

謝辞

本論文は著者が，明治大学大学院理工学研究科博士後期課程において，画像応用システム研究室にて行った研究成果をまとめたものになります．

本論文をまとめるにあたり，研究の機会を与えて頂き，その遂行が円滑に行われるために，終始あたたかくも適切なご指導とご鞭撻を頂きました宮本龍介准教授に心より感謝致します．

明治大学理工学部情報科学科の先生方には学部生時代より常に変わらぬあたたかいご指導を頂きました．大変感謝しております．ここに深く感謝を示します．また，お忙しい中，副査をお引き受け頂いた堤利幸教授，井口幸洋教授には論文をご精読頂き，大変有益なご助言を頂きました．厚く御礼申し上げます．また，飯塚秀明教授には，著者の本研究に対する取り組みにおいて，多くの有益かつ重要なご指摘，ご提案を頂きました．深く感謝致します．

さまざまな面でご協力，ご支援を頂きました画像応用システム研究室の皆様に厚く御礼申し上げます．中でも長い研究室生活を一緒に過ごしてくださった安達美穂さんや，深層学習に関してお手伝いいただいた森岡隼也さん，人検出手法のアプリケーション側を頑張ってくださった薛俊峰さん，修士までお世話になった大木琢郎氏には感謝しても仕切れません．

加えて，検証に使用するデータセット作成のための手動でのアノテーション作業にご助力を頂きました皆様には，大変な労力を要する作業でありながら根気強く取り組んで頂きました．

本研究成果の一部は，国立研究開発法人情報通信研究機構（NICT）の委託研究「未来を創る新たなネットワーク基盤技術に関する研究開発」(採択番号 19103)，JSPS 科研費 JP22K12082，および，明治大学科学技術研究所の若手研究の助成により得られたものであります．ここに記すと共に感謝の意を表します．

最後に，長い学生生活を辛抱強く支えて頂いた父母，姉に深く感謝致します．

参考文献

[1] S. Zhang, C. Bauckhage, and A.B. Cremers. Informed haar-like features improve pedestrian detection. In *Proc. IEEE Conf. Comput. Vis. Pattern Recognit.*, pp. 947–954, 2014.

[2] R. Miyamoto and T. Oki. Soccer player detection with only color features selected using informed haar-like features. In *Proc. Advanced Concepts for Intelligent Vision Systems*, Vol. 10016, pp. 238–249, 2016.

[3] T. Oki and R. Miyamoto. Efficient GPU implementation of informed-filters for fast computation. In *Proc. IEEE Pacific-Rim Symposium on Image and Video Technology*, pp. 302–313, 2017.

[4] Jakub Kurzak, Hartwig Anzt, Mark Gates, and Jack Dongarra. Implementation and tuning of batched cholesky factorization and solve for nvidia gpus. *Proc. IEEE Transactions on Parallel and Distributed Systems.*, Vol. 27, pp. 1–1, 01 2015.

[5] Yaohung M. Tsai, Piotr Luszczek, Jakub Kurzak, and Jack Dongarra. Performance-portable autotuning of opencl kernels for convolutional layers of deep neural networks. In *Proceedings of the Workshop on Machine Learning in High Performance Computing Environments*, MLHPC '16, pp. 9–18, Piscataway, NJ, USA, 2016. IEEE Press.

[6] Free Software Foundation, Inc. GCC, the GNU Compiler Collection. `https://gcc.gnu.org`.

[7] Chris Lattner and Vikram Adve. LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation. In *Proceedings of the 2004 International Symposium on Code Generation and Optimization (CGO'04)*, Palo Alto, California, Mar 2004.

[8] NVIDIA. AI Platform for Autonomous Machines — NVIDIA Jetson AGX Xavier. https://www.nvidia.com/en-us/autonomous-machines/jetson-agx-xavier/, 2018.

[9] Microsoft. Reference (C++ AMP). `https://docs.microsoft.com/ja-jp/cpp/parallel/amp/cpp-amp-overview`, 2022.

[10] NVIDIA Corporation. Cuda. `http://docs.nvidia.com/cuda/`.

[11] DJI. Phantom 4 - DJI. `https://www.dji.com/jp/phantom-4`, 2016.

[12] Microsoft. What's a Universal Windows Platform (UWP) app? - UWP applications — Microsoft Docs. `https://docs.microsoft.com/en-us/windows/uwp/get-started/universal-application-platform-guide`.

[13] Chunfang Deng, Mengmeng Wang, Liang Liu, Yong Liu, and Yunliang Jiang. Extended feature pyramid network for small object detection. *IEEE Transactions on Multimedia*, Vol. 24, pp. 1968–1979, 2022.

[14] J. Redmon, S. Divvala, R. Girshick, and A. Farhadi. You only look once: Unified, real-time object detection. In *Proc. IEEE Conf. Comput. Vis. Pattern Recognit.*, pp. 779–788, 2016.

[15] J. Redmon and A. Farhadi. Yolo9000: Better, faster, stronger. In *Proc. IEEE Conf. Comput. Vis. Pattern Recognit.*, pp. 6517–6525, 2017.

[16] J. Redmon and A. Farhadi. Yolov3: An incremental improvement. *arXiv*, 2018.

[17] Ultralytics. YOLOv5 by Ultralytics. `https://github.com/ultralytics/yolov5`, 2022.

[18] Robotstart inc. ストラドビジョンが公道で自動運転車向け AI 物体認識システムの試乗デモを実施日本市場に本格展開へ「2032 年には 5600 万台に導入したい」 - ロボスタ. `https://robotstart.info/2022/10/25/stradvision-road-demo.html`, 2022.

[19] S. Hara, H. Yomo, R. Miyamoto, Y. Kawamoto, H. Okuhata, T. Kawabata, and H. Nakamura. Challenges in Real-Time Vital Signs Monitoring for Persons during Exercises. *International Journal of Wireless Information Networks*, Vol. 24, pp. 91–108, 2017.

[20] R. Miyamoto, H. Yokokawa, T. Oki, H. Yomo, and S. Hara. Human detection in top-view images using only color features. *Journal of the Institute of Image Electronics Engineers of Japan*, Vol. 46, No. 4, pp. 559–567, 2017.

[21] S. Hanashiro, J. Xue, J. Morioka, R. Miyamoto, T. Hamagami, K. Yanagihara, Y. Kawamoto, H. Okuhata, H. Yomo, and T. Takubo. Testing environment for developing a wireless networking system based on image-assisted routing for sports applications. *Proc. International Conference on Sports Sciences Research and Technology Support*, pp. 138–143, 10 2021.

[22] R. Girshick, J. Donahue, T. Darrell, and J. Malik. Rich feature hierarchies for accurate object detection and semantic segmentation. In *Proc. IEEE Conf. Comput. Vis. Pattern Recognit.*, pp. 580–587, 2014.

[23] Koen EA Van de Sande, Jasper RR Uijlings, Theo Gevers, and Arnold WM Smeulders. Segmentation as selective search for object recognition. In *Proc. IEEE Int. Conf. Comput. Vis.*, pp. 1879–1886. IEEE, 2011.

[24] W. Liu, D. Anguelov, D. Erhan, C. Szegedy, S. Reed, C. Fu, and A. Berg. SSD: single shot multibox detector. In *Proc. European Conference on Computer Vision*, pp. 21–37, 2016.

[25] Karen Simonyan and Andrew Zisserman. Very deep convolutional networks for large-scale image recognition. In Yoshua Bengio and Yann LeCun, editors, *ICLR*, 2015.

[26] M. Everingham, L. Van Gool, C. K. I. Williams, J. Winn, and A. Zisserman. The PASCAL Visual Object Classes Challenge 2007 (VOC2007) Results. http://www.pascal-network.org/challenges/VOC/voc2007/workshop/index.html.

[27] R. Girshick. Fast R-CNN. In *Proc. IEEE Int. Conf. Comput. Vis.*, December 2015.

[28] S. Ren, K. He, R. Girshick, and J. Sun. Faster R-CNN: towards real-time object detection with region proposal networks. *IEEE Trans. Pattern Anal. Mach. Intell.*, Vol. 39, No. 6, pp. 1137–1149, 2017.

[29] P. Viola and M. Jones. Robust real-time face detection. *Int. J. Comput. Vis.*, Vol. 57, No. 2, pp. 137–154, 2004.

[30] J. Canny. A computational approach to edge detection. *IEEE Trans. Pattern Anal. Mach. Intell.*, Vol. PAMI-8, No. 6, pp. 679–698, 1986.

[31] J.P. Lewis. Fast template matching. *Vis. Interface*, Vol. 95, , 11 1994.

[32] 辰夫大附, 佐藤政生, 昌良橘, 司郎鳥居. 複合長方形領域の最小分割. 情報処理学会論文誌, Vol. 24, No. 5, pp. 647–653, sep 1983.

[33] P.J. Werbos. Backpropagation through time: what it does and how to do it. *Proc. of the IEEE*, Vol. 78, No. 10, pp. 1550–1560, 1990.

[34] Sepp Hochreiter and Jürgen Schmidhuber. Long short-term memory. *Neural Comput.*, Vol. 9, No. 8, pp. 1735–1780, November 1997.

[35] Rui Zhao, Haider Ali, and Patrik van der Smagt. Tow-stream RNN/CNN for action recognition in 3d videos. pp. 24–28, 9 2017.

[36] Shuohang Wang and Jing Jiang. Learning natural language inference with LSTM. In *Proceedings of the 2016 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies*, pp. 1442–1451, San Diego, California, June 2016. Association for Computational Linguistics.

[37] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Ł ukasz Kaiser, and Illia Polosukhin. Attention is all you need. In I. Guyon, U. Von Luxburg, S. Bengio, H. Wallach, R. Fergus, S. Vishwanathan, and R. Garnett, editors, *Advances in Neural Information Processing Systems*, Vol. 30. Curran Associates, Inc., 2017.

[38] Alec Radford, Jeffrey Wu, Rewon Child, David Luan, Dario Amodei, and Ilya Sutskever. Language models are unsupervised multitask learners. 2018.

[39] Gobinda Chowdhury. Natural language processing. *ARIST*, Vol. 37, pp. 51–89, 01 2005.

[40] Tom B. Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, Sandhini Agarwal, Ariel Herbert-Voss, Gretchen Krueger, Tom Henighan, Rewon Child, Aditya Ramesh, Daniel M. Ziegler, Jeffrey Wu, Clemens Winter, Christopher Hesse, Mark Chen, Eric Sigler, Mateusz Litwin, Scott Gray, Benjamin Chess, Jack Clark, Christopher Berner, Sam McCandlish, Alec Radford, Ilya Sutskever, and Dario Amodei. Language models are few-shot learners. Vol. 33, pp. 1877–1901, 2020.

[41] ASReal Limited. AI Programmer. `https://aiprogrammer.hashlab.jp`.

[42] Lambda. OpenAI's GPT-3 Language Model: A Technical Overview. https://lambdalabs.com/blog/demystifying-gpt-3, 2020.

[43] Jay Alammar. How GPT3 Works - Visualizations and Animations. https://jalammar.github.io/how-gpt3-works-visualizations-animations/, 2020.

[44] J. Kolter, Pieter Abbeel, and Andrew Ng. Hierarchical apprenticeship learning with application to quadruped locomotion. In J. Platt, D. Koller, Y. Singer, and S. Roweis, editors, *Proc. Advances in Neural Information Processing Systems*, Vol. 20. Curran Associates, Inc., 2007.

[45] Constantin A Rothkopf and Dana H Ballard. Modular inverse reinforcement learning for visuomotor behavior. *Biological cybernetics*, Vol. 107, pp. 477–490, 2013.

[46] Alex Graves, Greg Wayne, and Ivo Danihelka. Neural turing machines. *arXiv preprint arXiv:1410.5401*, 2014.

[47] Oriol Vinyals, Meire Fortunato, and Navdeep Jaitly. Pointer networks. In C. Cortes, N. Lawrence, D. Lee, M. Sugiyama, and R. Garnett, editors, *Advances in Neural Information Processing Systems*, Vol. 28. Curran Associates, Inc., 2015.

[48] Dzmitry Bahdanau, Kyunghyun Cho, and Yoshua Bengio. Neural machine translation by jointly learning to align and translate. *arXiv preprint arXiv:1409.0473*, 2014.

[49] Ilya Sutskever, Oriol Vinyals, and Quoc V Le. Sequence to sequence learning with neural networks. *Proc. Advances in Neural Information Processing Systems*, Vol. 27, , 2014.

[50] Scott Reed and Nando de Freitas. Neural programmer-interpreters. In *International Conference on Learning Representations*, pp. 1–13, 2016.

[51] Jonathon Cai, Richard Shin, and Dawn Song. Making neural programming architectures generalize via recursion. In *5th International Conference on Learning Representations, 2017, Toulon, France, April 24-26, 2017, Conference Track Proceedings*, pp. 1–20. OpenReview.net, 2017.

[52] David E. Rumelhart and James L. McClelland. *A General Framework for Parallel Distributed Processing*, pp. 45–76. 1987.

[53] Francesco Donnarumma, Roberto Prevete, Fabian Chersi, and Giovanni Pezzulo. A programmer–interpreter neural network architecture for prefrontal cognitive control. *International journal of neural systems*, Vol. 25, No. 06, p. 1550017, 2015.

[54] Ilya Sutskever and Geoffrey E Hinton. Using matrices to model symbolic relationship. *Advances in neural information processing systems*, Vol. 21, , 2008.

[55] Jürgen Schmidhuber. Learning to Control Fast-Weight Memories: An Alternative to Dynamic Recurrent Networks. *Neural Computation*, Vol. 4, No. 1, pp. 131–139, 01 1992.

[56] Felix Gers, Jürgen Schmidhuber, and Fred Cummins. Learning to forget: Continual prediction with lstm. *Neural computation*, Vol. 12, pp. 2451–71, 10 2000.

[57] Zhe Cao, Gines Hidalgo, Tomas Simon, Shih-En Wei, and Yaser Sheikh. Openpose: Realtime multi-person 2d pose estimation using part affinity fields. *IEEE Trans. Pattern Anal. Mach. Intell.*, Vol. 43, No. 1, pp. 172–186, 2021.

[58] Zhe Cao, Tomas Simon, Shih-En Wei, and Yaser Sheikh. Realtime multi-person 2d pose estimation using part affinity fields. In *Proc. IEEE Conf. Comput. Vis. Pattern Recognit.*, pp. 7291–7299, 2017.

[59] S. Zhang, R. Benenson, and B. Schiele. Filtered channel features for pedestrian detection. In *Proc. IEEE Conf. Comput. Vis. Pattern Recognit.*, pp. 1751–1760, 2015.

[60] Epiphan video. AV.io 4K - USB / HDMI 4K capture card - Epiphan Video. https://www.epiphan.com/products/avio-4k/, 2023.

[61] Ryusuke Miyamoto, Shingo Kobayashi, Takuro Oki, Hiroyuki Yomo, and Shinsuke Hara. Improved pairwise max suppression considering total number of targets. In *IEEE Proc. of International Conference on Systems, Man, and Cybernetics, SMC 2018*, pp. 2091–2095, 2018.

[62] T. Oki, R. Aoki, S. Kobayashi, R. Miyamoto, H. Yomo, and S. Hara. Vision-based detection of humans on the ground from actual aerial images by informed filters using only color features. In *Proc. International Conference on Sport Sciences Research and Technology Support*, pp. 84–89, 2019.

[63] Microsoft. Win32 and COM APIs for UWP apps - Windows UWP applications — Microsoft Docs. `https://docs.microsoft.com/en-us/uwp/win32-and-com/win32-and-com-for-uwp-apps`.

[64] NVIDIA Corporation. Programming Guide :: CUDA Toolkit Documentation. `https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html`.

[65] 伊藤智義. GPU プログラミング入門. 講談社, 2013.

[66] G. Bradski. The OpenCV Library. *Dr. Dobb's Journal of Software Tools*, 2000. `https://opencv.org`.

[67] Intel. Intel® Instrinsics Guide. `https://www.intel.com/content/www/us/en/docs/intrinsics-guide/index.html`.

[68] J. Morioka and R. Miyamoto. A study on accuracy improvement of small object detection using deep neural networks. In *Proc. IEEE Asia Pacific Conference on Circuits and Systems*, pp. 73–76, 2021.

[69] 森岡隼也, 宮本龍介. 深層学習に基づく小物体検出の精度向上に関する一検討. スマートインフォメディアシステム研究会 (SIS). 電子情報通信学会, 12 2022.

[70] M. Tan, R. Pang, and Quoc V. Le. Efficientdet: Scalable and efficient object detection. In *Proc. IEEE Conf. Comput. Vis. Pattern Recognit.*, pp. 10781–10790, June 2020.

[71] T. Lin, P. Goyal, R. Girshick, K. He, and P. Dollar. Focal loss for dense object detection. In *Proc. IEEE Int. Conf. Comput. Vis.*, pp. 2980–2988, Oct 2017.

[72] Arvind Neelakantan, Quoc V. Le, and Ilya Sutskever. Neural programmer: Inducing latent programs with gradient descent. In Yoshua Bengio and Yann LeCun, editors, *4th International Conference on Learning Representations, 2016, San Juan, Puerto Rico, May 2-4, 2016, Conference Track Proceedings*, pp. 1–18, 2016.

[73] Alex Graves, Greg Wayne, and Ivo Danihelka. Neural turing machines. *arXiv preprint arXiv:1410.5401*, pp. 1–26, 2014.

[74] RISC-V Foundation. RISC-V Foundation — Instruction Set Architecture (ISA). https://riscv.org/.

[75] Andrew Waterman and Krste Asanovic. *The RISC-V Instruction Set Manual*. SiFive Inc. and CS Division, EECS Department, University of California, Berkeley, 2017.

[76] A.W. Burks. Electronic computing circuits of the eniac. *Proceedings of the IRE*, Vol. 35, No. 8, pp. 756–767, 1947.

[77] DeepMind. Competitive programming with AlphaCode. `https://www.deepmind.com/blog/competitive-programming-with-alphacode`, 2012.

第9章　業績

9.1　和文ジャーナル

1. 空撮画像を対象とした Informed-Filters に基づく人検出の C++ AMP を用いたリアルタイム実装, 津山 雅彦, 薛 俊峰, 森岡 隼也, 大木 琢郎, 宮本 龍介, 画像電子学会誌, 第 52 巻 第 2 号 (通巻 264 号), 2023

9.2　国際会議論文

1. Hardware Implementation of a Classifier Trained with Informed-Filters Using Only Color Features. M. Tsuyama, S. Aoki, T. Oki and Ryusuke Miyamoto. Proc. IEEE International Symposium on Intelligent Signal Processing and Communication Systems, pp. 319-324, 2018.

2. Embedded Implementation of Human Detection Using Only Color Features on the NVIDIA Xavier, Masahiko Tsuyama, Takuro Oki, Shingo Kobayashi, Risako Aoki, Ryusuke Miyamoto, Hiroyuki Yomo, Shinsuke Hara, Proc. IEEE International Symposium on Intelligent Signal Processing and Communication Systems, pp. 1-2, 2019.

3. Machine Code Generation for the RISC-V Instruction Set Using Neural Programmer-Interpreters, Masahiko Tsuyama, Ryusuke Miyamoto, IEICE Trans. International Workshop on Smart Info-Media System in Asia, pp. 83-88, 2021.

4. Addressing a Problem in Constructing a Transcoder using Neural Programmer-Interpreters, Masahiko Tsuyama, Ryusuke Miyamoto, IEICE Trans. International Workshop on Smart Info-Media System in Asia, pp. 57-62, 2022.

9.3　査読なし論文

1. 機械学習によるデータからのかけ算の学習に関する一検討. 津山 雅彦, 宮本 龍介. パルテノン研究会, 2018

2. 色特徴のみを用いた Informed-Filters による物体検出手法のハードウェア実装. 津山 雅彦, 青木 修平, 大木 琢郎, 宮本 龍介. 画像関連学会連合会 第 5 回秋季大会, 2018

3. 意味論に基づく探索範囲の制限による画像情報を用いた物体検出の高速化. 津山 雅彦, 安達 美穂, 本多 和史, 宮本 龍介. 画像電子学会 Media Computing Conference, 2022

4. 意味論的領域分割の計算コストと精度の関係に関する研究. 伊藤 陸斗, 森岡 準也, 安達 美穂, 津山 雅彦, 今川 隆司, 宮本 龍介. 深層学習に基づく小物体検出の精度向上に関する一検討, 2022

5. C++ プログラムからの JetBot のモーター制御. 上田 有里子, 津山 雅彦, 薛 俊峰, 宮本 龍介. パルテノン研究会, 2022

9.4　受賞歴

1. ISPACS2019 Best Student Paper Award - Embedded Implementation of Human Detection Using Only Color Features on the NVIDIA Xavier, Masahiko Tsuyama, Takuro Oki, Shingo Kobayashi, Risako Aoki, Ryusuke Miyamoto, Hiroyuki Yomo, Shinsuke Hara, Proc. IEEE International Symposium on Intelligent Signal Processing and Communication Systems, pp. 1-2, 2019.