博士論文

ヘテロジニアス計算環境における深層学習基盤の研究

日高　雅俊

Acknowledgements

本論文は，東京大学大学院情報理工学系研究科知能機械情報学専攻　原田達也教授の指導のもと執筆されました．

原田達也教授には私が学部 4 年生のときから 7 年以上に渡りご指導いただきました．深層学習技術が革新的な進歩をみせ，機械学習分野をとりまく環境がめまぐるしく変化する時代において，テーマ設定から発表資料の作成まで多岐に渡りご支援をいただくことができ，深く感謝申し上げます．

副査をご担当いただいた稲葉雅幸教授，國吉康夫教授，岡田慧教授，中山英樹准教授には，本論文に対する重要なご指摘を多数いただくことができ，より良い論文へと仕上げることができました．感謝申し上げます．

木倉悠一郎さんは，本論文の一部となる WebDNN の開発を共同で行なってくださいました．WebDNN の実現には深層学習に関する知識だけではなくプログラミングに関する広範な知識が必要であり，私の知識が及ばない部分を引き受けていただき，素晴らしいソフトウェアを開発することができました．Web 技術を扱うという研究室の中でも異質な研究テーマであり，知識豊富な木倉さんは非常に重要な相談相手になっていただきました．

同期である椋田悠介助教には，学生時代より数学に関する相談，論文の改良等数多くの場面でご協力いただきました．Tejero de Pablos Antonio 特任研究員，黒瀬優介特任研究員には研究環境の整備に多大なる貢献をいただきました．大西克典さん，早川顕生さん，金山哲平さん，上原康平さん，河合航さんをはじめとするみなさまには，研究室内で最も重要な装置である計算機群のセットアップ・メンテナンスを担当していただきました．山根宏彰特別研究員，森友亮さんをはじめとした研究室のメンバーには，ミーティングでのアドバイスのほか，様々な分野の情報を教えていただいたり，雑談や食事に付き合っていただいたりと研究室での生活を豊かにしてくださいました．研究室秘書の金子葵さん，加治佐美由希さん，丸山佐和子さんには奨学金・出張などにかかわる事務で大変お世話になりました．研究室のみなさまには改めて感謝いたします．

最後に，私の長きに渡る学生生活を支援してくださった家族に最大の感謝を捧げます．

Abstract

機械学習モデルの 1 つである Deep Neural Network (DNN) は，画像認識，自然言語処理，ゲーム AI など様々なパターン認識・生成タスクにおいて高い性能を達成し，応用範囲の広がりを見せている．深層学習や，学習された DNN モデルによる推論の計算量は大きく，深層学習を用いたシステムの実現には強力な計算資源が必要である．計算資源として最も一般的なのは強力な演算装置を搭載したクラウドサービス上のサーバであり，データを端末からインターネットを通じて転送・処理する．しかし，サービス提供コスト，回線の速度や遅延，データがサービス提供者側にわたることによるプライバシー上の懸念が存在する．上記のような課題の解決策として，消費者が保持するスマートフォンなどの情報端末や IoT デバイスの潜在的な計算資源を活用することが考えられる．クラウドに代表される，強力かつ均質なコンピュータを用いた深層学習基盤はハードウェア・ソフトウェア両面において活発に開発・利用がなされている一方，情報端末など異種のハードウェア・ソフトウェアが混在するヘテロジニアス環境における計算基盤は未だ十分に整備されていない状況である．

本研究の目的は，ヘテロジニアス環境における DNN モデルの学習・推論を効率的に行える基盤を構築することである．深層学習基盤の構成要素は大きく分けて 3 つある．DNN モデルは行列計算を繰り返すことにより計算を行う．行列計算を実行する機構と，行列計算の実行順序を制御する機構が必要である．さらに，コンピュータ一台で学習から推論までを完結することはまれであり，通信に関する機構が必要である．推論アプリケーションにおいては学習済みモデルの読み込み，学習システムにおいては分散計算での他の計算ノードとの協調が該当する．これらの要素をヘテロジニアス環境に適した手法を用いて実装する．

本研究では特に，従来科学技術計算に用いられることが少なかった一方，カメラの性能向上などにより DNN を用いたアプリケーション開発の需要が高まっているスマートフォン等の消費者向け端末をターゲットに具体的な要件検討，ソフトウェア実装を進める．ハードウェアの差異を吸収し，またソフトウェアのインストールの煩雑さを解決するプラットフォームとして Web ブラウザを用いることができる．Web ブラウザ上で科学技術計算を行うことをブラウザコンピューティングと呼び，特に，深層学習に関する計算を高速に行うための GPU の活用の観点からソフトウェアの実装法を論じる．

本研究では大きく分けて，複数デバイスを協調させて深層学習を行う課題と，単一デバイスで推論を実行する課題に取り組む．前者の課題では，デバイス間の通信による計算結果の同期が必要な深層学習において，デバイスの性能差に着目し計算時間を均等化する分散計算アルゴリズムの提案および，単一の行列計算アルゴリズム実装をプラットフォームごとに異なる GPU の利用方法にあわせて変換する抽象化機構を提案する．後者の課題では，まず学習済

DNN モデルを圧縮し，通信環境にかかわらずアプリケーションの動作を迅速に開始できるようにする手法を論じる．次に，様々な学習済み DNN モデルについて，単一の中間表現へと変換し，演算の性質を活用した最適化を施すことにより統一的な高速化が達成できることを示す．

これらの組み合わせにより，DNN モデルの学習から利用までを高性能・均質なクラウド環境ではなく，ヘテロジニアスな端末上の計算能力を用いて実現可能となる．

Table of contents

List of figures **xi**

List of tables **xiii**

1 序論 **1**

1.1 研究背景 1

1.2 研究目的 3

1.3 論文の構成 4

2 Deep Neural Network **5**

2.1 多次元テンソル 5

2.2 DNN で用いられる代表的なレイヤー 6

2.2.1 全結合層 6

2.2.2 畳み込み層 6

2.2.3 活性化層 7

2.2.4 プーリング層 8

2.3 DNN の最適化 9

2.3.1 バッチ正規化層 11

2.3.2 SGD の改良 11

2.3.3 ファインチューニング 12

2.4 計算グラフ 12

3 ヘテロジニアス計算環境における深層学習基盤 **15**

3.1 深層学習基盤とは 15

3.2 ヘテロジニアス計算環境とは 17

3.3 ブラウザコンピューティング 21

3.4 Web ブラウザ 21

3.5 JavaScript による科学技術計算 22

3.6 Web ブラウザにおけるハードウェアアクセラレーション 25

3.6.1 WebGL 26

3.6.2 WebMetal 27

3.6.3 WebGPU 27
3.6.4 WebAssembly 27
3.7 ソフトウェアスタック 28

4 Web ブラウザを計算ノードとした Deep Neural Network の分散計算 29
4.1 序論 29
4.2 関連研究 30
4.2.1 Web ブラウザにおける DNN の学習 30
4.2.2 Web ブラウザにおけるアクセラレータの活用 31
4.2.3 ヘテロジニアス計算環境における DNN の分散学習 31
4.3 手法 32
4.3.1 コード変換による複数アクセラレータの活用 33
4.3.2 分散計算アルゴリズムの選択 46
4.3.3 計算性能差を考慮した分散計算システム 49
4.4 実験 54
4.4.1 コード変換ベンチマーク 54
4.4.2 分散計算ベンチマーク 55

5 Web ブラウザのための Deep Neural Network の圧縮 61
5.1 序論 61
5.2 関連研究 63
5.3 手法 65
5.3.1 パッチ表現を考慮した反復的な圧縮 65
5.3.2 精度回復のためのスケールファインチューニング 68
5.3.3 可逆圧縮による効率的な符号化 69
5.4 実験 71
5.4.1 AlexNet on ImageNet 71
5.4.2 MobileNetV2 on ImageNet 75
5.5 画像生成への応用 77

6 Web ブラウザにおける Deep Neural Network の推論高速化 79
6.1 序論 79
6.2 関連研究 80
6.3 手法 81
6.3.1 中間表現による DNN モデルの共通化 82
6.3.2 中間表現上でのモデル最適化 85
6.3.3 複数アクセラレータを対象としたコード生成 88
6.3.4 パラメータの圧縮 92
6.3.5 ランタイムライブラリ 92
6.4 実験 93

6.5 オープンソースソフトウェアとしての WebDNN 97

7 結論と今後の展望 99

7.1 結論 99

7.1.1 DNN の分散学習 (第 4 章) 100

7.1.2 学習済み DNN の圧縮 (第 5 章) 100

7.1.3 学習済み DNN の推論高速化 (第 6 章) 100

7.2 今後の課題 102

References 105

List of figures

2.1 im2col 処理による入力行列の変換 7
2.2 im2col を用いた畳み込み計算 8
2.3 VGG16 モデルの計算グラフの一部 13

3.1 深層学習基盤の構成要素 15
3.2 Web アプリケーションの例の動作画面 24
3.3 Web ブラウザによる DNN 学習・推論システムのソフトウェアスタック 28

4.1 データ並列とモデル並列 47
4.2 バッチサイズに比例して学習率を変化させた場合の validation loss の推移 51
4.3 各デバイス・バックエンドにおける，バッチサイズに対する処理時間 52
4.4 分散計算システムの構成 54
4.5 固定バッチサイズにおけるスループットの推移 56
4.6 Group3 におけるスループット・各デバイスのバッチサイズの推移 59
4.7 動的なデバイス接続・切断におけるスループット・各デバイスのバッチサイズの推移 60

5.1 提案システムの概要 63
5.2 既存手法による AlexNet モデルの圧縮結果の例 66
5.3 手法の概要 66
5.4 パッチ生成の 2 つの手段 67
5.5 パッチの符号化の概要 71
5.6 反復的な圧縮過程における，モデルのサイズと学習データに対する精度の関係 . 72
5.7 ベースモデルおよびそれにパッチを適用した際のファイルサイズと validation 精度の関係 73
5.8 圧縮が進んだ際の各レイヤーのサイズ変化 75
5.9 事前圧縮を行わない場合の手法間の比較 76
5.10 提案手法により圧縮された Neural Style Transfer モデルによる画像変換の結果 . 78

6.1 WebDNN の構成 81
6.2 Average Pooling における深層学習フレームワーク間の差異 84

6.3 WebDNN に入力される計算グラフの例 (Chainer) 86
6.4 WebDNN の IR の例 87
6.5 WebDNN の最適化された IR の例 89
6.6 スマートフォン上で，カメラから取得された画像を Neural Style Transfer モデルでリアルタイムに加工するアプリケーションの実行例. 94
6.7 WebDNN のベンチマーク (MacBook) 96
6.8 WebDNN のベンチマーク (iPhone, ResNet50 モデル) 96

List of tables

4.1 CIFAR-10 を識別する DNN のモデル構造 50
4.2 512×512 の行列同士の行列積の処理時間 55
4.3 分散計算に用いるデバイス一覧 56
4.4 分散計算に用いるデバイスの組み合わせ 57
4.5 分散計算によるスループット測定結果 57

6.1 Github に寄せられた WebDNN に関するバグ・質問の集計 98

Chapter 1

序論

1.1 研究背景

機械学習モデルの 1 つである Deep Neural Network (DNN) は，画像認識，自然言語処理，ゲーム AI など様々なパターン認識・生成タスクにおいて高い性能を達成し，応用範囲の広がりを見せている．DNN が成功した背景として，インターネットを通じて大量のデータが得られるようになったこととともに，計算機の性能向上が大きな役割を果たしている．大量のデータを用いて DNN のパラメータを最適化する過程である深層学習や，学習された DNN モデルによる推論 (モデルを未知のデータに対して適用して利用する) の計算量は大きく，深層学習を用いたシステムの実現には強力な計算資源が必要である．深層学習で必要となる計算は Graphics Processing Unit (GPU) 上で効率よく行えることがよく知られており，高性能な GPU を搭載したサーバを購入するかクラウドサービス上でレンタルすることにより計算資源を確保することが一般的である．そして DNN モデルを用いたサービスの展開は，スマートフォンや IoT デバイスにおいて処理対象となる画像等を撮影し，サーバへアップロードして処理し，結果を返送することにより実現が可能となる．クラウドサービスを用いる場合，システムのセットアップは容易であるものの，高負荷な計算をサーバ上で引き受けるためユーザ数に応じた費用が掛かる．また，インターネットを経由することによる必然的な遅延や，データをサービス提供者側に送信することにより生じるプライバシー上の懸念などがサービス提供における課題となる．また，クラウド上の計算資源も無尽蔵に存在するわけではなく，機械学習の国際学会への論文投稿締め切り直前に GPU が枯渇したという事例 [1] も存在する．

上記のような課題の解決策として，消費者が保持するスマートフォンなどの情報端末や IoT デバイスの計算資源を活用することが考えられる．2017 年時点の日本における個人のスマートフォンの保有率は 60.9%とされている [1]．機種の内訳は不明であるが，代表的な機種である Pixel 2 に搭載されている GPU の性能理論値は約 500GFLOPS である．日本全体のスマートフォンの計算能力を総合すると 10^{19}FLOPS 程度と見積もることができる．これは代表的なスーパーコンピュータである京コンピュータの 10^{16}FLOPS の千倍であり，消費者が持っている端末の潜在的な計算能力は決して無視できない規模であるといえる．金銭的に見れば，クラウド

[1] https://www.theregister.co.uk/2017/05/22/cloud_providers_ai_researchers/

サービスの代表である Amazon Web Services において NVIDIA Tesla V100 GPU (14TFLOPS) 搭載のサーバが 1 台・1 時間あたり 3 ドルでレンタルされている．単純に計算するとスマートフォン全体の計算能力は 1 時間当たり 200 万ドルの経済的価値があると考えることができる．また，IoT デバイス向けに生産されている小型コンピュータ基板の 1 つであり，パソコン並みの計算能力を持つ Raspberry Pi シリーズは世界で 1000 万台以上が出荷されている [2].

カメラから取り込まれた画像からパターン認識する技術を用いて，皮膚の写真から皮膚疾患の早期発見を行うアプリケーション [3]，水族館等で展示されている動物の名前・解説を提示するアプリケーション [4]，様々な動植物を撮影し，コレクションするゲームアプリケーション [5]，電子機器のコネクタ名称を判定し，適切なケーブルを推薦してくれるアプリケーション [6] など，医療・ゲーム・実用共に企業・個人からアプリケーションが多数提供されている．具体的な技術について明らかにされている事例は少ないが，DNN を学習する技術・ソフトウェアが普及し，これらを活用することで実用化されているものも多いと考えられる．DNN の学習手法を知らずとも学習データとなる画像を用意するだけで簡単に画像分類モデルを学習できるクラウドサービス [7] も登場しており，専門家でなくても DNN を活用したアプリケーションを開発可能になってきている．しかし上記のサービスの開発者は，「ただ，現状ではモデルの学習に使っている GPU インスタンスの費用が高いため，マネタイズの問題がまだまだ残っています.」[8] と述べており，クラウドを用いる限り計算資源のコスト問題が避けられない．アプリケーション開発時のモデル学習，公開後の推論を端末側の計算資源を用いて行うことで，今までコスト面で実現されなかったサービスが実現可能となる余地があると考えられる．経済的な観点以外にも，プライバシー保護上の制約により施設外のクラウド等が利用できない環境での応用も考えられる．例えば医療機関において，病理画像から病変を認識する課題が考えられる．病理画像に対して DNN モデルが有効であることが知られている [2] が，検体の染色方法などに施設間の差異 [3] があり，すべての施設でそのまま利用できる DNN モデルの構築は難しい．この課題に対応するため，施設内で撮影したデータを用いて共通モデルをファインチューニング [4] またはドメイン適応 [5] により補正し施設に適したモデルを作成することが有用であると考えられる．このような計算を行う場合に，施設内に存在するサーバの余剰計算能力や，事務用のパソコンなどを計算資源として用いることにより，これらの課題を解決できる可能性がある．

クラウドに代表される，強力かつ均質なコンピュータを用いた深層学習基盤はハードウェア・ソフトウェア両面において活発に開発・利用がなされている．ハードウェアとしては NVIDIA 社の GPU が高いシェアを誇っており，ソフトウェアとしては Linux を OS (Operating System) とし，NVIDIA 社の GPU 専用の汎用並列計算プラットフォーム CUDA を用いて計算を実装した Caffe 等の深層学習フレームワークが広く用いられている．一方，異種のハードウェア・ソ

[2]https://www.raspberrypi.org/magpi/raspberry-pi-sales/
[3]https://www.skinvision.com/service
[4]https://lens.linne.ai/ja/
[5]https://biome.co.jp/news/avada_portfolio-7443/
[6]https://nlab.itmedia.co.jp/nl/articles/1902/18/news027.html
[7]https://aimaker.io/
[8]https://qiita.com/2zn01/items/0d3147c9d9d804ad9880

フトウェアが混在するヘテロジニアス計算環境における計算基盤は未だ十分に整備されていない状況である．

1.2 研究目的

本研究の目的は，ヘテロジニアス計算環境における DNN モデルの学習・推論を効率的に行える基盤を構築することである．本研究では大きく分けて，複数デバイスを協調させて深層学習を行う課題と，単一デバイスで推論を実行する課題に取り組む．深層学習においては計算量が膨大であり，複数デバイスの計算資源の統合によりスループットを向上させることが必要である．一方推論においてはデバイス一台の計算能力で扱える DNN モデルの範囲が広く，ネットワークへデータ送信を行わないことによる低遅延・プライバシー保護のメリットを享受できる設定が有用であると考えられる．

深層学習基盤の構成要素は大きく分けて 3 つある．DNN モデルは行列計算を繰り返すことにより計算を行う．行列計算を実行する機構と，行列計算の実行順序を制御する機構が必要である．さらに，コンピューター台で学習から推論までを完結することはまれであり，通信に関する機構が必要である．推論アプリケーションにおいては学習済みモデルの読み込み，学習システムにおいては分散計算での他の計算ノードとの協調が該当する．

本研究では特に，従来科学技術計算に用いられることが少なかった一方，カメラの性能向上などにより DNN を用いたアプリケーション開発の需要が高まっているスマートフォン等の消費者向け端末をターゲットに具体的な要件検討，ソフトウェア実装を進める．典型的な実装方法は，ハードウェア・OS ごとにネイティブアプリケーションを開発し，その中に DNN モデルの計算機能を組み込むことである．しかし現代ではデスクトップ・スマートフォンを合わせると多数の OS が存在しており，また OS ごとに主流のプログラミング言語や GPU へアクセスするための API が異なっている．そのため，主要な環境 (Windows，MacOS，Android，iOS) すべてをカバーするネイティブアプリケーションの実装はコストが大きい．また，アプリケーションのインストール手段も異なってくる．本研究では，インストールの問題やアプリケーション開発手法の違いを吸収するプラットフォームとして Web ブラウザに着目する．Web ブラウザはあらゆる端末に標準搭載されており，Web ページの静的な文書や画像を表示するだけでなく，JavaScript という言語で記述されたプログラムを動作させることができる．さらに，プログラム上で動的な通信も可能となっている．例えば Google Document はインターネット上の他のユーザと協調して文書を編集できるワードプロセッサアプリケーションである．この機能を科学技術計算に応用することで，複数の端末を用いた分散計算が実現できる．

ほぼすべてのブラウザで動作する JavaScript 言語はスクリプト言語で，実行速度が比較的低いという課題がある．Web ブラウザの種類を問わず同一の挙動が期待できるため，アプリケーションの骨格部分として非常に有用であるものの，科学技術計算部分では端末の性能を引き出すボトルネックとなってしまう．深層学習において有効な GPU の利用は JavaScript から呼び出せる Web ブラウザの API によって行うこととなる．Web ブラウザには画面表示，センサー情報取得，暗号化など多種多様な API が搭載されているが，特に計算の高速化に関する API を本研究ではアクセラレータ API と呼ぶことにする．しかしこれは Web ブラウザごとに

実装されているアクセラレータ API やその機能レベルに差異が大きく，この差分を吸収し各ハードウェアの性能を最大限引き出すことが重要な課題となる．複数のデバイスを協調させる分散計算のノードとして用いる場合，デバイス間の計算能力の差異が課題となる．同一機種の端末を多数保有していることを前提としたシステムは応用範囲が狭い．既存の深層学習フレームワークにおける分散計算機能では原則として同一性能のハードウェアを並べることが前提になっており，アドホックに複数デバイスを協調させて分散計算させるためにはこの点の考慮が必要となる．Web ブラウザを用いることでソフトウェアのインストールが不要となる反面，大きなデータ容量を占める DNN モデルのダウンロードにかかる時間は通信環境に大きく依存する．手軽に利用できるアプリケーションを提供するためには，通信による待機時間の短縮が重要である．通信にかかわる部分はハードウェアの性能に依存するものであり，Web ブラウザの仕組みによらず必要なものである．

本研究の主な貢献は以下の通りである．本研究では Web ブラウザを具体的なプラットフォームとしてシステムの実装を進めるが，開発したアルゴリズムは異なる機種が混在するサーバ環境・IoT デバイス等での DNN に関する処理の高速化にも寄与する．

- ヘテロジニアス計算環境において深層学習を効率的に行うために必要な技術を整理した．
- ハードウェアの計算能力を活用するため，高速化に関する API を抽象化する技術を提案した．
- 計算性能の異なる計算ノードを協調させて深層学習を行わせるための分散計算手法を提案した．
- Web ブラウザ上での DNN を用いたアプリケーション特有の課題に対応するモデル圧縮手法を提案した．

1.3 論文の構成

本論文の構成を述べる．第 1 章では，論文全体にわたる研究背景および目的を述べた．第 2 章では，本研究で学習・推論を行う Deep Neural Network モデルの特徴を解説する．第 3 章では，ヘテロジニアス計算環境における深層学習基盤を構成する要素について論じ，その具体的な実装対象となる Web ブラウザをプラットフォームとした科学技術計算について解説する．第 4 章では，性能の異なる複数の端末を協調させ，分散計算により深層学習を行うための手法を論じる．第 5 章では，学習済 DNN モデルを圧縮し，通信環境にかかわらずアプリケーションの動作を迅速に開始できるようにする手法を論じる．第 6 章では，様々な学習済み DNN モデルについて，単一の端末上での推論を効率化するための手法を論じる．第 7 章では，論文全体にわたる結論と今後の展望を述べる．

Chapter 2

Deep Neural Network

本章では，本研究で学習・推論する機械学習モデルである Deep Neural Network (DNN) について述べる．

DNN は，線形変換と非線形変換を多層に組み合わせ，入力データを複雑に非線形変換する．従来は対象データの性質に合わせた非線形変換を手動で設計し，得られた特徴量を線形識別機で識別する手法が一般的であった．DNN では誤差逆伝播法により，手動での特徴量設計の代わりに，学習データから全ての層のパラメータを学習することが可能である点が重要な特徴である．これは End-to-end な学習と呼ばれる．DNN の概念は 1990 年代に存在していた [6] ものの，画像分類コンペティション ILSVRC2012 において，SIFT [7] 等の手動で設計された特徴量と線形モデルの学習を組み合わせた従来手法を大きく引き離したスコアで優勝したことにより大きく注目された [8]．DNN が 2010 年代になって成功した背景として，データからの特徴量学習が可能になるほどの大規模なデータセットが開発された [9] ことのほか，計算上のボトルネックが行列積であり，GPU の高い並列計算能力が活用できたことが大きく貢献した．

本研究では主に画像分類をタスクとして扱うことを想定した DNN の特徴を述べる．DNN のレイヤー (層) としてほとんどのモデルで用いられるのは，全結合層，畳み込み層，活性化層である．

2.1 多次元テンソル

DNN のレイヤー間を流れるデータは実数値を要素とする多次元テンソルで表現される．チャンネル数 (C)，画像の高さ (H)，画像の幅 (W) からなる 3 次元のテンソル X が用いられる．各スカラー要素は $X_{c,h,w}$ のように 3 つのインデックスで指定する．行列積を行う全結合層へ入力を行う場合は，C，H，W をまとめて 1 つの次元とみなした 1 次元のテンソル X_c に変形する．このように，数学的な演算はないもののデータの変形を行うレイヤーが暗黙的に挿入される場合がある．

2.2 DNNで用いられる代表的なレイヤー

2.2.1 全結合層

全結合層 (Fully-connected layer) は，行列積により入力テンソルを線形変換する．C_{in} の1次元テンソル X(行ベクトル) を入力，$C_{out} \times C_{in}$ の2次元テンソル W を重み (学習可能なパラメータ) としたとき，出力 Y は

$$Y = XW^T \tag{2.1}$$

と計算される．重みパラメータ数は $C_{out}C_{in}$，計算量は $C_{in}C_{out}$ となる．通常，行列積後に C_{out} 次元のバイアス $\boldsymbol{b}$ を加算する．

2.2.2 畳み込み層

畳み込み層 (Convolutional layer) は，画像のピクセルのように空間的な並びを持った入力テンソルに対し，局所領域ごとに重み行列との内積を計算する処理である．数学的に正確な表現は相関演算であるが，慣習として畳み込みと呼ばれている．

$C_{in} \times H_{in} \times W_{in}$ の3次元テンソル X を入力，$C_{out} \times C_{in} \times K_h \times K_w$ の4次元テンソル W を重みとしたとき，出力 Y の各要素は

$$Y_{c,h,w} = \sum_{d=1}^{C_{in}} \sum_{y=1}^{K_h} \sum_{x=1}^{K_w} X_{d,h-p+y,w-p+x} W_{c,d,y,x} \tag{2.2}$$

となる．W はカーネルと呼ばれることもある．ただし，p は padding と呼ばれる値で，一般的にカーネル幅 K_h の半分 (切り捨て) とし，入出力の画像サイズが一致するようにする．X の範囲外の部分は0で埋められているとみなす．通常，行列積後に C_{out} 次元のバイアス $\boldsymbol{b}$ を加算する．

重みパラメータ数は $C_{out}C_{in}K_hK_w$ に対し，計算量は $C_{out}C_{in}K_hK_wH_{in}W_{in}$ となる．

畳み込み層は，画像からの特徴抽出のように，画像中の絶対位置にかかわらず同じ特徴量を抽出したい場合に有効である．DNNの中では全結合層と類似する役割を持つが，パラメータ数が大幅に少なくなる．これにより，過学習の問題が回避しやすくなるという利点がある．また，計算量をパラメータ数で割った比率は，全結合層の場合と比べ $H_{in}W_{in}$ だけ大きい．現代の計算機は演算コアの能力に対しメモリアクセスやネットワークの速度がボトルネックとなりやすい．機械学習の数学的な面とは別に，通信等のコストの面でも畳み込み層は有用である．

具体的な実装においては，式 (2.2) を直接的に実装するのではなく，im2col と呼ばれる形状の変換を行い行列積で演算可能とする．im2col の例を図 2.1 に示す．画像の幅・高さが3ピクセル，カーネルの幅・高さが2ピクセルである．バッチサイズは1とし表記を省略している．1点の出力ピクセルの計算に必要な X の要素を1列にまとめる．次に，図 2.2 のように W を出力チャンネルを1行に展開した行列との行列積を行い，最後に要素の順序を入れ替えることで Y が得られる．この方式のメリットは，メモリアクセスパターンが比較的よく，また多く

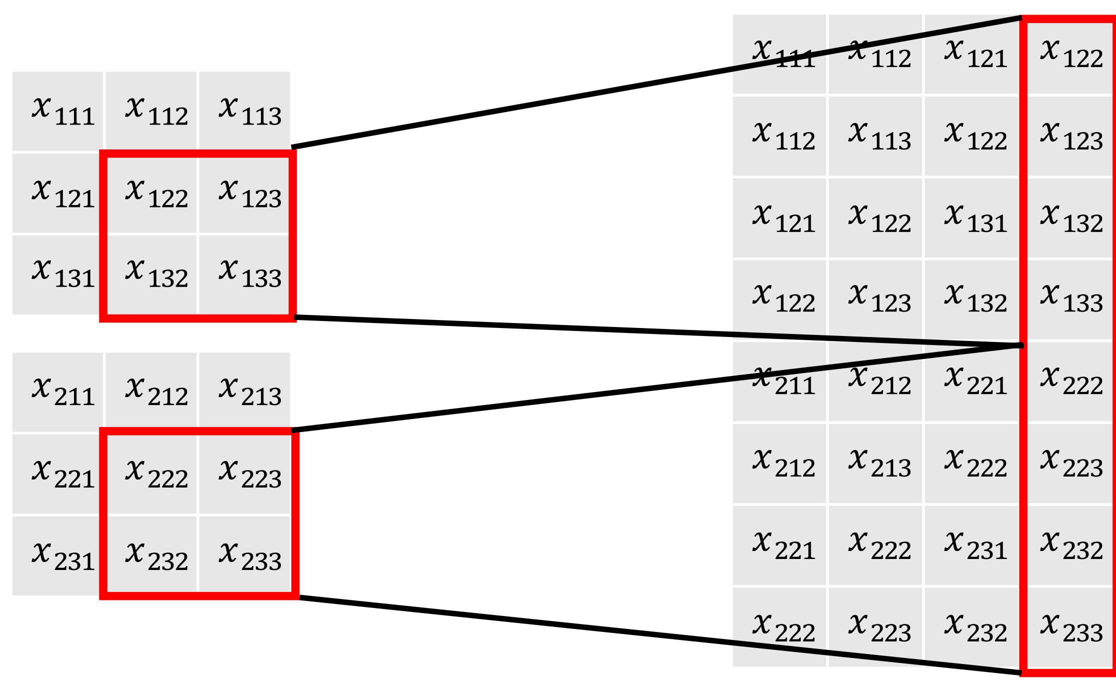


図 2.1 im2col 処理による入力行列の変換

のプラットフォームで最適化された実装が存在する行列積ルーチンをそのまま利用できる点である．

2.2.3 活性化層

全結合層，畳み込み層は線形な変換であり，これを複数回連続適用しても，1 回の線形変換と同等の表現力しか得られない．複雑な非線形変換を可能にするため，線形変換と線形変換の間に非線形変換を行う活性化層を挿入する．活性化層の形式として，$y = \max(x, 0)$ で与えられる ReLU がしばしば用いられる．

深層学習では，誤差逆伝播法により最適化を行うため，出力層の勾配を入力に近い層まで伝播させる必要がある．その際，活性化層において勾配にスケールが掛けられると，多数の活性化層を通過することにより勾配が発散又は消失してしまう問題が生じる．ReLU ではこの問題が最小限に抑えられると考えられている．ほかの活性化層として，sigmoid $\frac{1}{1+e^{-x}}$ や ELU [10] が用いられる場合もある．

活性化層は要素ごとの計算であるため，入力サイズ C_{in} に対し計算量は C_{in} であり，全結合層・畳み込み層と比較し相対的に小さい．

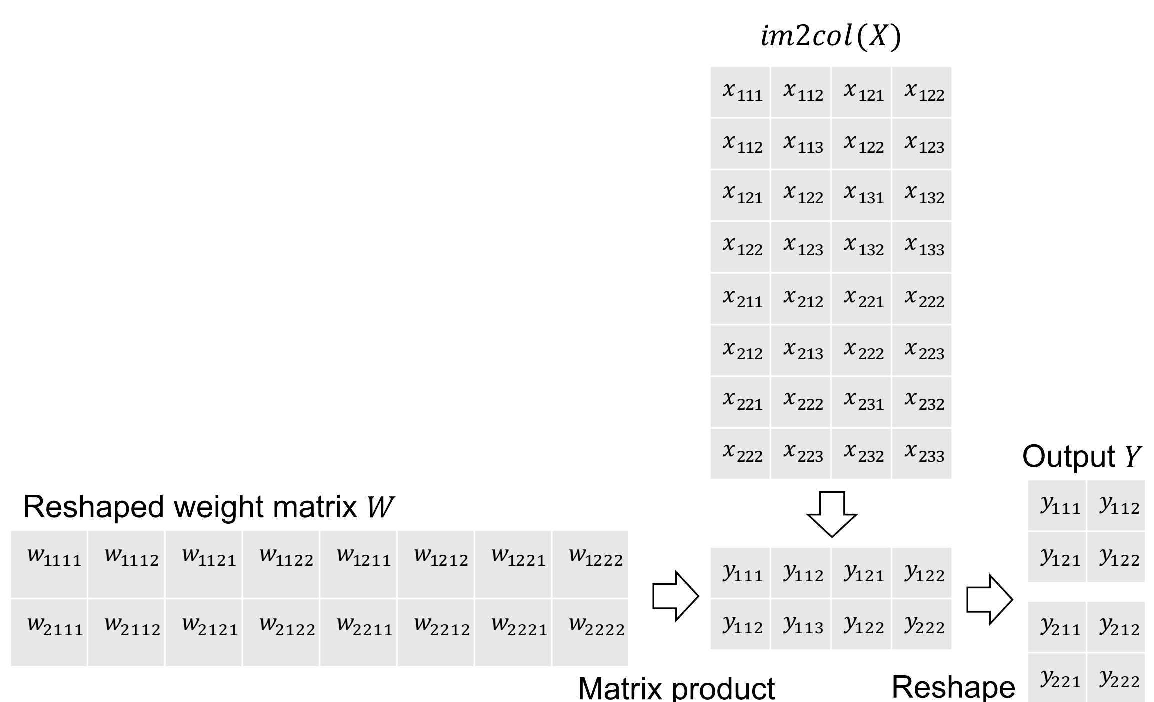


図 2.2 im2col を用いた畳み込み計算

2.2.4 プーリング層

画像中の物体を認識する課題において，画像中の物体の位置が多少変化しても出力は変化しないことが理想的である．これを実現するため，畳み込み層の後ろにプーリング層を導入し，画像のサイズを縮小しつつ位置不変性を導入する．

代表的な最大値プーリング (Max Pooling) では，カーネルサイズ K，出力要素の 1 ピクセル移動に対する入力の注目領域の移動量を表すストライド S に対し

$$Y_{c,h,w} = \max_{0 \leq y,x < K} X_{c,hS+y,wS+x} \tag{2.3}$$

と計算する．すなわち $K \times K$ サイズのウィンドウ内の最大値を取り出すこととなる．$K = S = 2$ と設定する場合が多い．$C_{in} \times H_{in} \times W_{in}$ の 3 次元テンソル X を入力とした場合，$H_{out} = \frac{H_{in}}{S}, W_{out} = \frac{W_{in}}{S}$ として $C_{in} \times H_{out} \times W_{out}$ の 3 次元テンソル Y を出力する．最大値プーリングのほか，ウィンドウ内の平均をとる平均プーリング (Average Pooling) も用いられる．また，プーリング層ではなく畳み込み層にストライドを適用し，画像サイズを縮小する場合もある．

プーリング層は学習すべきパラメータはなく，計算量は $CH_{out}W_{out}K^2$ となる．

2.3 DNN の最適化

本研究で扱う，機械学習の一種である教師あり学習では，パラメータ w(すべての層の学習可能な重みの集合) を持つモデルに入力 x を与えて得られた出力と正解データ t を比較し，その差を損失 $E(w;x,t)$ として定義する．そして損失が最小となるよう w を最適化する．画像分類であれば，画像 1 枚 1 枚に対し，写っている物体の名称 (ラベル) が正解データとして与えられる．画像をモデルに与えた際の出力として推定されたラベルが出力され，これが正解データと一致していれば損失が小さく，不一致であれば損失が大きくなるように損失を定義する．分類問題では Softmax cross entropy loss，回帰問題では二乗誤差が用いられる場合が多い．

$l \in \{1,2,...,L\}$ 番目のレイヤーの入力を X^l，重みを W^l とし，レイヤーごとに定められた関数 $F^l(X^l,W^l)$ によって出力 X^{l+1} を計算する．Softmax cross entropy loss を用いる K クラスの分類問題では，最終レイヤーの出力 X^{L+1} を K ベクトルの行列とする．Softmax 関数は

$$\hat{X}_c^{L+1} = \frac{\exp(X_c^{L+1})}{\sum_{j=1}^{K} \exp(X_j^{L+1})} \tag{2.4}$$

と計算され，これは入力サンプルが各クラスに属する確率に対応する．Cross entropy loss は，

$$E = \sum_{j=1}^{K} -t_j \log(\hat{X}_j^{L+1}) \tag{2.5}$$

と定義される．ここで，正解ラベルを表す $\boldsymbol{t}$ は K 次元のベクトルで，ラベルに対応する 1 つの次元の値のみが 1，それ以外が 0 となる one-hot ベクトルを用いる．

DNN を教師あり学習する場合，誤差逆伝播法 (Backpropagation) が一般的に用いられる．誤差逆伝播法により，損失に対する各パラメータの勾配 (偏微分値)$\frac{\partial E}{\partial W}$ が得られる．誤差逆伝播法では，出力に近い変数から逐次的に勾配を計算していく．

まず X^{L+1} に対する勾配 $\partial X^{L+1} \triangleq \frac{\partial E}{\partial X^{L+1}}$ を計算する．これは以下のように求まる．

$$\partial X^{L+1} = \hat{X}^{L+1} - \boldsymbol{t} \tag{2.6}$$

以下，連鎖則 $\frac{\partial E}{\partial X^l} = \frac{\partial E}{\partial X^{l+1}} \frac{\partial X^{l+1}}{\partial X^l}$ を再帰的に用いる．

$$\partial X_i^l = \sum_k \frac{\partial F^l(X^l,W^l)_k}{\partial X_i^l} \partial X_k^{l+1} \tag{2.7}$$

となり，また途中の重みに対して

$$\partial W_{i,j}^l = \sum_k \frac{\partial F^l(X^l,W^l)_k}{\partial W_{i,j}^l} \partial X_k^{l+1} \tag{2.8}$$

と勾配が求まる．この計算は 1 層あたり $\partial X^l, \partial W^l$ の 2 つの計算が必要となり，一般的に順方向計算の 2 倍の計算量となる．

例えば全結合層の場合は

$$\partial X^l = \partial X^{l+1} W^l \tag{2.9}$$
$$\partial W^l = (\partial X^{l+1})^T X^l \tag{2.10}$$

となり，ReLU 層では

$$\partial X^l = \partial X^{l+1} \circ I(X^l > 0) \tag{2.11}$$

となる．ただし，$\circ$ は要素ごとの積，$I(X^l > 0)$ は X^l の要素ごとに正の値であれば 1，そうでなければ 0 を与える関数である．

この手順で得られた勾配をもとに，各パラメータを更新する．最も基本的な手法は勾配降下法である．ある時点 t における重み w_t に対し学習データを用いて勾配を計算し，w_{t+1} へと更新する．

$$w_{t+1} = w_t - \lambda \frac{1}{|D|} \sum_{x \in D} \partial E(w_t; x) \tag{2.12}$$

$\partial E(w_t; x)$ は，学習データの 1 サンプル x を用いて計算した損失に対する w_t の勾配を表す．x により値は変化する．勾配降下法ではデータセット D 中の全学習データを用いて勾配を計算し，その平均を用いる．ここで，ハイパーパラメータとして学習率 (learning rate)λ が必要である．勾配の値は現在のパラメータに対する一次近似値であるため，非線形モデルを一度で最適化することはできない．小さな λ を用いてモデルを更新した後，再度勾配を求めて反復的に更新を行う必要がある．

実際には，数万サンプル以上に上る学習データに対する勾配を求めてモデルを少しだけ更新するのは計算効率が低い．そのため，学習データから 1 サンプルから数百サンプル程度の部分集合 (ミニバッチ)D_s を抽出し，ミニバッチ中の学習サンプルに対する勾配の平均計算し，更新を行う．更新ごとに異なる部分集合を取り出すことで，すべての学習サンプルを用いることができる．これを確率的勾配降下法 (stochastic gradient descent=SGD) と呼ぶ．ミニバッチに含まれるサンプル数 $|D_s|$ をバッチサイズと呼ぶ．

$$w_{t+1} = w_t - \lambda \frac{1}{|D_s|} \sum_{x \in D_s} \partial E(w_t; x) \tag{2.13}$$

ここで，1 回の更新をイテレーション (iteration) と呼び，学習データセット全体を一度ずつミニバッチに利用する過程を 1 エポック (epoch) と呼ぶ．ミニバッチの大きさにより，同じデータセットであってもエポック当たりのイテレーション数が異なる．GPU で計算したり，複数台の計算機で分散計算したりする場合，バッチサイズを大きくすることで計算効率を向上することが可能である．GPU は数千個の演算コアからなり，それらをすべて稼働させるためには十分大きな計算量が必要なためである．実装上，バッチサイズ (N) に対応する次元が先頭に

付加された 4 次元のテンソルが用いられ，各画像に対する計算を大きな行列計算として処理する．動画像や 3D ボクセルを扱う場合は 5 次元テンソルが用いられる場合もある．バッチサイズが増えれば計算時間がかかり一定時間当たりの更新回数は減少するが，勾配の安定性が増し一回の更新における収束への寄与が大きくなるトレードオフの関係にある．

2.3.1 バッチ正規化層

層の数が多い DNN は，各層への入力が他の層のパラメータの分布変化に影響されるため最適化が難しい．この問題に対処するため，バッチ正規化レイヤー (Batch Normalization) が提案されている [11]．バッチ正規化レイヤーは以下の式で表される．

$$Y_{n,c,h,w} = \gamma_c \frac{X_{n,c,h,w} - Mean(X_{*,c,*,*})}{\sqrt{Var(X_{*,c,*,*}) + \varepsilon}} + \beta_c \tag{2.14}$$

他のレイヤーと異なり，ミニバッチに属するサンプルが独立ではなく全体の統計量を用いる点が特徴的である．そのため入出力をバッチサイズを含めた 4 次元テンソルで表現している．$Mean(X_{*,c,*,*})$ は，チャンネル c に属するすべての要素の平均値，$Var(X_{*,c,*,*})$ は分散である．これらは各イテレーションにおいて計算し，チャンネルごとの統計量を平均 0，分散 1 へと正規化している．ε は分散が 0 となった場合のゼロ除算回避のための 10^{-5} 程度の小さな値である．γ_c, β_c はチャンネルに対応するスカラーのスケール値で，最適化の対象である．学習後のモデルをテストする場合，バッチサイズ 1 では統計量が計算できず，また他のテストサンプルによって出力が変化してしまうのは問題となる．その代わりに $Mean, Var$ は学習時の統計量の移動平均を定数として用いる．

2.3.2 SGD の改良

純粋な SGD は式 (2.13) に示した通りであるが，より収束を速めるために改良を施した最適化方式が存在する．もっとも代表的なものは Momentum SGD である．Momentum SGD は以下の数式で表される．

$$\nabla E_t \triangleq \frac{1}{|D_s|} \sum_{x \in D_s} \partial E(w_t; x) \tag{2.15}$$

とおき，

$$w_{t+1} = w_t - \lambda \nabla E_t + \mu(w_t - w_{t-1}) \tag{2.16}$$

と更新する．慣性項 $\mu(w_t - w_{t-1})$ により勾配方向の時系列的な平滑化を行うことにより，一定方向への更新を促進する．スカラー値 μ は 0.9 が用いられる場合が多い．他の最適化方式として，RMSProp [12] や Adam [13] 等も提案されているが，大規模な DNN の最適化では Momentum SGD の利用例が多い [14][15].

2.3.3 ファインチューニング

学習の開始前に，パラメータは正規分布に従う乱数で初期化することが一般的である．一方，学習データが少ない場合は乱数で初期化されたモデルを十分に学習させることができない場合がある．医療画像等，大量の学習データを用意するのが難しい場合，類似するタスクのために学習されたパラメータを初期値として，タスクの差異を吸収する最低限のパラメータ変更を行うファインチューニング (fine-tuning) [4] が行われる．画像分類におけるファインチューニングでは，AlexNet [8]，VGG16 [16] などのモデルを 100 万枚以上の画像を含む大規模画像データセット ImageNet [9] を用いて学習した状態のパラメータを初期値として用いる．最終の全結合層は ImageNet における 1,000 クラスに対応しており，新しいタスクには使えないため乱数で初期化した別の行列に置換する．この状態で新しいタスクのデータセットを用いて最適化を行うことにより，すべての層を乱数で初期化した場合より一般に良い精度が得られることが知られている．

2.4 計算グラフ

DNN モデルは，変数 (多次元テンソル) をエッジ，畳み込み層などのレイヤー (層) をノードとする有向非循環グラフで表現できる．このグラフを計算グラフと呼ぶ．図 2.3 に計算グラフの例を示す．ここでは，レイヤーは長方形で表され，変数とその形状が八角形で表現されている．左上の変数が入力画像であり，バッチサイズ 4，3 チャンネル (RGB)，幅・高さが 224 ピクセルであることを表している．`Convolution2DFunction` は畳み込み層を表しており，入力画像のほか重み W・バイアス b を受け取り，64 チャンネルの変数を出力する．その変数が ReLU 活性化関数に入り，同一形状の変数を出力する．以下同様にグラフのエッジをたどって計算が進行する．

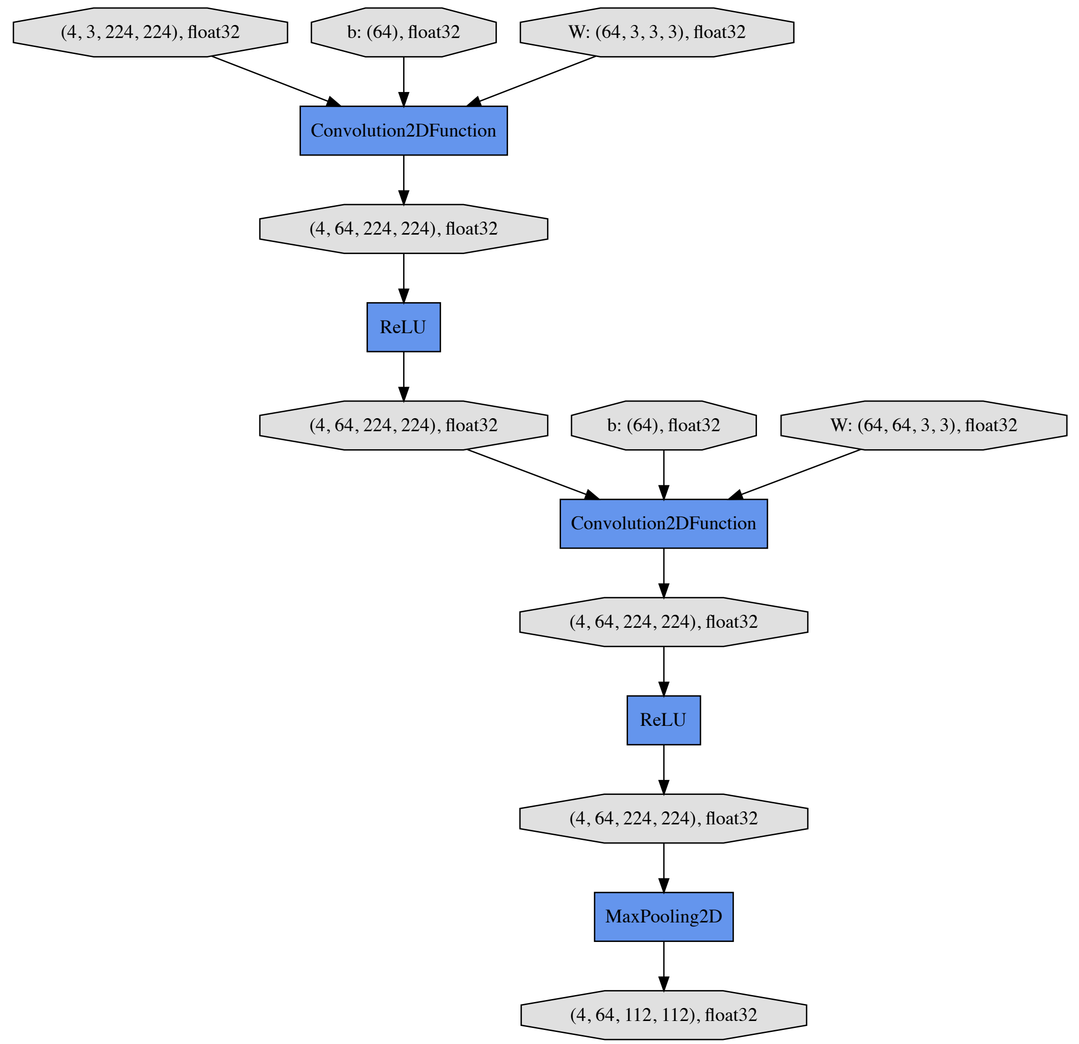


図 2.3 VGG16 モデルの計算グラフの一部．深層学習フレームワーク Chainer により可視化．

Chapter 3

ヘテロジニアス計算環境における深層学習基盤

3.1　深層学習基盤とは

深層学習基盤とは，DNN モデルの学習およびアプリケーションへの組み込みにおいて必要な機構の総称である．DNN のアルゴリズムは第 2 章で述べたように行列を処理単位とし，多数の行列計算を繰り返すことで計算結果を得る．ソフトウェア実装を想定した，深層学習基盤の構成要素を図 3.1 に示す．本研究で想定する環境における主な課題については，第 3.2 節以降で説明する．

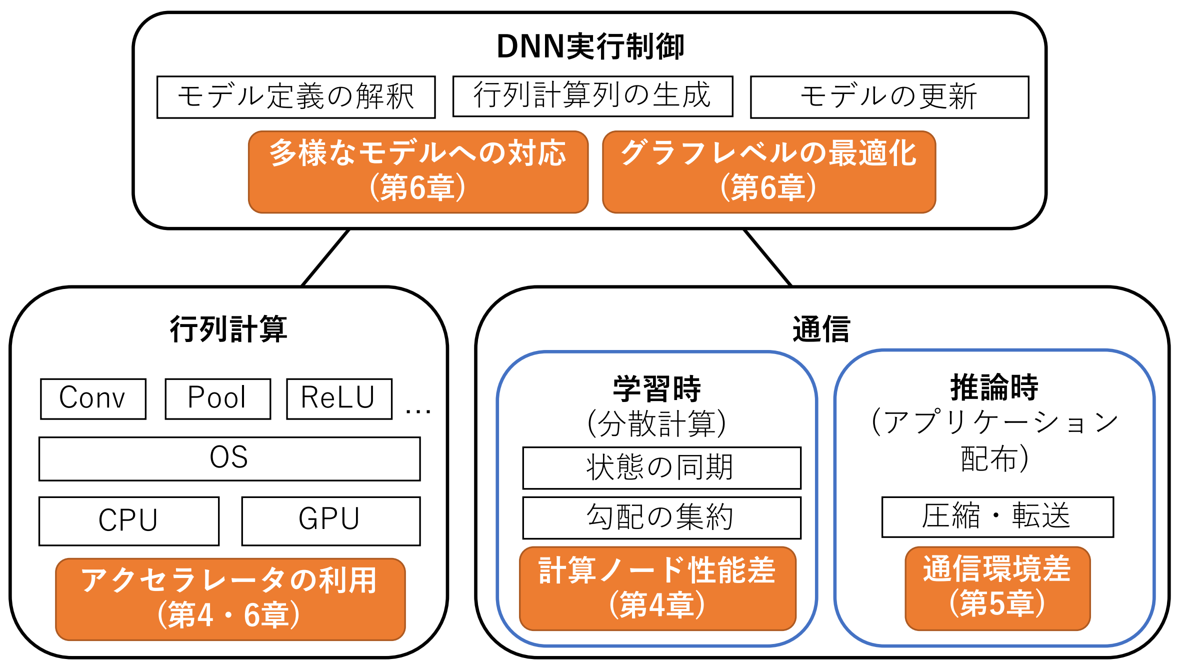


図 3.1 深層学習基盤の構成要素．オレンジ色の枠内は本研究で検討する主な課題．

DNN の最小単位である行列計算には畳み込み層，プーリング層，活性化層などが存在する．これらの具体的な実装は，OS を介して CPU または GPU 上で計算される．計算量の最も大きな箇所であり，GPU 等を用いることにより高速に行うことが必要となる．行列計算に特化したハードウェアとして Tensor Processing Unit 等も開発されている．行列計算に用いるハードウェアを総称してアクセラレータと呼び，その効率的な利用が焦点となる．

DNN の実行制御機構では，抽象的な DNN モデルを具体的な行列計算の列へと変換する．最初に必要なのはモデル定義の解釈である．DNN モデルは多数の層からなるが，層の種類，順序，動作パラメータ (プーリング層のストライドなど) に広い自由度があり，開発者がタスクに合わせて定義する必要がある．多様なモデル定義を可能とするため，もっとも単純な深層学習フレームワークではあらかじめ定義されたレイヤーを実行順に並べて記載した設定ファイルを用いる一方，最近のフレームワークにおいては畳み込み層・バッチ正規化層・活性化層のような 1 つの DNN 中で何度も利用される組み合わせをサブルーチンとして定義し，ループによって多数の層を見通しよく接続できるようになっているなど，モデル定義手段は様々な方法が考案されている．第 6.3.1 節に実例を掲載している．次に，実際の行列計算の順序への変換を行う．DNN モデルはモデル定義の段階で有向非循環グラフで表現されているが，トポロジカルソートなどの手段により各レイヤーの実行順序を決定する．DNN モデルの推論においては，入力データから出力データを計算する一連の処理を生成する一方，学習を行う場合は誤差逆伝播法により学習可能な行列それぞれの勾配を計算する処理を生成する必要がある．ここで，モデル定義時の層の種類と実際に呼び出される行列計算のコードは必ずしも一対一対応しない．例えば第 2.3.1 節で述べたように，バッチ正規化層は学習時と推論時で論理的な挙動が異なる．また，第 2.2.2 節で示したように，畳み込み層は直接的な実装ではなく，im2col 関数と行列積を順に実行するほうが性能が高い場合がある．そのため状況に応じてグラフ上で実際の行列計算の呼出し手順を割り当て・最適化する必要がある．最後に，DNN モデルの学習の場合は学習可能な行列それぞれについて，計算された勾配を用いて更新する．更新則は第 2.3 節のとおりであるが，学習させるモデルや分散計算の条件に合わせてハイパーパラメータを調整する必要がある．

DNN モデルの学習から推論までが単一のコンピュータで完結しない場合は，他のコンピュータとの通信が必要となる．学習時においては，分散計算を行う場合が該当する．本研究で扱う分散計算の方針においては，計算ノード間で DNN モデルの状態を同期し，それぞれの計算ノードで行列計算を行い勾配を計算する．全計算ノードにおける勾配を集約し，DNN モデルを更新する．推論時においては，学習済み DNN を組み込んだアプリケーションを配布する場合が該当する．学習された DNN モデルをそのまま推論に用いることも可能であるが，一般にファイルサイズが大きく通信コスト・ストレージ容量等の面で効率的でない．そのため，DNN モデルの推論結果の変化を小さく抑えつつ圧縮することが考えられ，圧縮された状態のモデルをネットワーク上で転送する．

3.2 ヘテロジニアス計算環境とは

深層学習基盤の実装において，クラウド環境においては，NVIDIA 社の GPU を搭載し，Linux OS がインストールされた計算機が広帯域のネットワークにより接続されている構成が一般的であり，これを前提に深層学習システムが構築されている．行列計算の実装は NVIDIA 社の GPU 専用の並列計算プラットフォーム CUDA を用いて行う．CUDA プラットフォームにおけるプログラミング言語である CUDA C 言語を用いることで，GPU 上の多数の演算コアを用いて行列計算を実装することができる．また上位のライブラリとして行列積などを実装する cuBLAS，畳み込みなどを実装する cuDNN が提供されており，ハードウェアの性能を十分に活用できるライブラリが普及している．DNN の実行制御および，上述のライブラリで実装されていないレイヤーを実装するのが深層学習フレームワークである．Caffe [17]，Chainer [18]，Tensorflow [19] など多数存在し，モデルの定義方法，データの前処理，アプリケーションへの組み込み手段などにおいて特色がある．複数の計算ノードを用いた学習のためには，科学技術計算で広く活用されている通信ライブラリ規格 MPI に準拠しハードウェアに最適化された実装が存在しており，分散計算に対応している深層学習フレームワークの多くはこれを利用して計算結果の同期を行う．分散計算の性能向上に当たっては，主に計算・通信に関するハードウェアを改良することが重要となる．

クラウド環境と比較し消費者向けのスマートフォン端末や IoT デバイスの潜在的な計算能力を活用することのメリットは第 1 章で述べたとおりであるが，これらのデバイスは様々なハードウェア・OS が混在している．本研究におけるヘテロジニアス計算環境とは，計算資源となる CPU や GPU の命令体系の違い，OS などの深層学習フレームワークからみて下位のソフトウェアの違い，計算資源や通信回線の性能の違いをもつコンピュータが含まれる環境である．先頭 2 つは数学的なアルゴリズムではなくプログラミング言語やライブラリの呼び出し方法等のソフトウェアの実装方法に大きな影響を生じ，クラウド環境において用いられてきたソフトウェアを利用することが難しく，また環境ごとに別個のソフトウェア実装を行うことは非常にコストが高い．このような環境において深層学習基盤を実装するにあたり生じるボトルネックの解消が主な課題となる．

DNN に関する計算負荷の大部分を占めるのが行列積をはじめとした行列計算であり，これは GPU 上で計算することに適している．一般的なノートパソコンにおいて，CPU の 1 コア当たりの理論計算性能は 50GFLOPS 程度で 4 コア程度が搭載されているのに対し，GPU は 500GFLOPS 程度の理論計算性能を持つため，GPU の利用は計算の高速化において重要である．ヘテロジニアス計算環境において同一のソフトウェアを実行可能にする手段がいくつか考えられる．VMware をはじめとした仮想マシンソフトウェアは，ある OS が実行されているシステム上で別の OS を実行するものであり，特定の OS 向けに開発されたアプリケーションを多くの環境で実行することを可能とする．Java 仮想マシンは，Java 言語で開発されたソフトウェアを動作させるプラットフォームであり，多くの OS に対応している．しかし，これらのプラットフォームは一般的な情報端末の多くにはインストールされておらず，利用に準備が必要である．CPU 上で実行されるソフトウェアについては CPU のアーキテクチャを仮想化し高い互換性を誇るものの，GPU を統一的に利用する機構については整備されているとは言い難い．様々

なメーカーの GPU を統一的に利用できるようにするプラットフォームとして，Windows であれば DirectX，MacOS であれば Metal があり，各 OS 向けのドライバレベルでメーカー間の互換性が担保されている．OpenCL は OS 非依存に GPU およびマルチコア CPU を用いた並列計算を実装可能なプラットフォームとして提唱された．しかしながら MacOS では非推奨となっており，また Android では一部機種でしか利用できないなど，現実的には多くの課題が残されている．そのため，GPU を利用して実行速度の向上を図る必要のある行列計算部分については，どのような実装方法を取るにせよ，複数の規格への対応が避けられないのが現状である．

本研究では，ヘテロジニアス計算環境において同一ソフトウェアを実行する具体的な手段として，Web ブラウザをプラットフォームとして用いる．Web ブラウザはほぼすべてのスマートフォン・パソコン等の端末に標準搭載されている．Web ブラウザは Microsoft · Google · Apple 等様々な会社により提供されているが，JavaScript 言語をプログラミングに共通して用いることができる．一種の仮想環境とみなすこともできるが，数値計算に向いた言語ではないうえ，原則 CPU の 1 コアのみの利用となる．GPU を利用することにより速度ボトルネックの解消を目指すため，Web ブラウザ特有の様々な制約を考慮した実装法を第 4 章，第 6 章にて論じる．Web ブラウザの主な仕様について，第 3.4 節以降で解説する．

DNN の実行制御部分は，計算量が行列計算部分と比べ相対的に小さくソフトウェア実装自体の高速化の優先度は低く，また具体的な言語・ファイルフォーマットに依存しない部分では既存の深層学習フレームワークと同様のアルゴリズムを実装すれば最低限の実行は可能である．一方，行列計算部分を GPU で実行する場合，効率の良い CUDA 環境でも行列積 · ReLU 関数など 1 つの処理単位の開始に 10 マイクロ秒程度必要とする．ハードウェア抽象度が高くなるヘテロジニアス計算環境ではより長い時間がかかり，また ResNet50 [14] 等層が多いモデルにおいては，数百個の処理が必要となるため，無視できない処理時間となる．このボトルネックを解消するため，実行制御の一部として処理単位の数を削減する等の最適化を可能とする機構を第 6 章にて論じる．また，深層学習に関する研究は既存の深層学習フレームワークを用いて行われる場合が多いが，複数のフレームワークが用いられており互換性がない．本研究で想定するヘテロジニアス計算環境において動作する深層学習フレームワークは少なく，学習された DNN モデルを本研究で作成するシステム上で統一的に利用可能とする手段についても同時に論じる．

通信部分は，複数の計算ノードを連携させて DNN モデルの学習を行う場合に，均質なクラウド環境においては，同一の計算量のタスクを全計算ノードに配布することで通信と計算を交互に休みなく行うことができ，通信にかかる時間を短縮すれば処理全体の時間を短縮することが可能となる．通信速度自体の高速化はハードウェア・OS レベルで研究されており，また通信すべきデータ量の削減についても勾配の圧縮に関する研究が多数なされている．一方，本研究のソフトウェア環境において MPI ライブラリが利用できないだけでなく，各計算ノードに同一の計算量のタスクを割り振る従来のアルゴリズムをそのまま適用することができない．計算性能が計算ノードごとに異なる場合，計算性能が高い計算ノードが計算を終えても，計算性能が低い計算ノードは計算を続けているため，次の処理へ進むことができない．計算も通信も行わない時間が発生することがヘテロジニアス環境特有の課題であり，計算性能の異なるハードウェア同士を連携させるための特有のアルゴリズムについて第 4 章で検討する．DNN モデ

ルの推論段階においては，他のコンピュータで学習されたモデルを，アプリケーションを実行するコンピュータに受信する．モデルを一度受信した後はコンピュータ内で処理が完結するため，学習時のように計算性能の差による同期の問題は生じない一方，学習済みのモデルを利用するアプリケーションはユーザがインタラクティブに利用するものが多いと考えられるため，モデルを読み込む時間はユーザにとっての待ち時間となる．特にスマートフォン上のアプリケーションを想定した場合には，あらかじめソフトウェアをインストールして出荷する特定目的専用のデバイスと異なり，ユーザの要求に応じてインターネット回線経由で直ちにモデルを読み込める必要がある．同一のモデルを大量に配布するため，通信量を削減するために JPEG 画像などと同様，事前の圧縮を行うことが有用である．様々な通信環境において，できるだけ高速にモデルを読み込ませることが必要となるため，第 5 章でその手段を検討する．なお，本研究では計算を行うコンピュータは固定された単一のサーバとのみ通信すると仮定する．インターネットを介した大規模な活用においては，Content Delivery Network のようにネットワーク上の距離などを考慮した通信仲介手段と組み合わせることが性能向上につながると考えられる．

クラウド環境と対比される計算環境の一種として，IoT デバイス，自動運転車など DNN を適用すべきデータが発生する場所に近い位置で計算を行う方策をエッジコンピューティングと呼び，リアルタイム性の確保，セキュリティリスクの軽減，通信量の削減などのメリットがあるとされている [20]．ビジネス上の文脈により，(1) 携帯電話の基地局など，クラウドと端末の中間的な位置に専用のサーバを設置し計算を行うものと，(2) センサを搭載した装置そのもので計算を行うものの 2 種類が存在する．文脈 (1) は主に通信事業者により推進されており，Kanai ら [21] によれば，エッジコンピューティングの主な用途は 3 つに分類される．1 番目はコンテンツのキャッシングで，動画等の大容量のコンテンツを各端末がクラウドと通信するのではなくエッジコンピューティングサーバから取得することにより，ネットワークの負荷軽減を行う．これは端末側では代替できない機能である．2 番目は端末の計算コスト低減である．Augmented Reality などのアプリケーションにおいて，カメラ画像からの物体認識などの高コストな処理を代替するものである．これはタスクに対して端末の処理能力が十分高ければ端末上で行うことも可能であると考えられる．端末側で処理できるタスクが増加すれば，エッジコンピューティングサーバを設置するコストが不要となるメリットがある．3 番目はビッグデータ処理であり，計算能力を持たない IoT センサ等の情報を集約・処理する．常時送信されてくるデータの保管が重要な機能であると考えられ，信頼性が低い端末で代替することは難しいと考えられる．文脈 (2) は主に半導体製造企業・AI 関連企業により推進されており，Raspberry Pi [1] のような汎用コンピュータ基板のほか，DNN 向け半導体を搭載した Intel Movidius Neural Compute Stick [2]，NVIDIA Jetson [3] 等のコンピュータ基板が多数提案されている．これらの基板を組み込む装置はハードウェアに特化したソフトウェアを実装することが一般的と考えられるが，常時計算能力を使い切るアプリケーションばかりではないと考えられ，余剰計算能力を

[1]https://www.raspberrypi.org/
[2]https://software.intel.com/en-us/movidius-ncs
[3]https://www.nvidia.com/ja-jp/autonomous-machines/embedded-systems/

分散計算に応用できる可能性がある．基板ごとにアクセラレータの利用方法や性能は異なり，ヘテロジニアス計算環境の一種とみなすことができる．

有志を募り，インターネットに接続されている大量のデバイスを用いて，大規模な演算を行う枠組みを Volunteer Computing と呼ぶ．Volunteer Computing のもっとも著名な成功例として SETI@Home [22] がある．このプロジェクトは，天文台で観測された電波を解析し，宇宙人の痕跡を発見することを目的としている．プロジェクトでは Windows パソコンで動作するスクリーンセーバーアプリケーションとしてデータ解析プログラムを配布し，参加者はこれをインストールした．このプログラムはインターネットを通じて解析すべきデータを受け取り，結果を返す仕組みとなっている．専用ソフトウェアのインストールが必要ではあったが，300 万台以上のコンピュータの参加を得た．これはデスクトップパソコンを想定したソフトウェアであった一方，実際の計算プロジェクトとの関連付けはないものの，スマートフォンが登場する以前の携帯電話端末で分散計算を行う早期の研究として i アプリ (NTT Docomo 社の携帯電話端末で実行できる Java アプリケーション) を用いた事例が提案されている [23]．なお，スマートフォンを計算資源として利用する場合のハードウェア上の課題としてバッテリー消費の問題がある．DNN の推論を行わせる場合はインタラクティブなアプリケーションとしての実行であり，高負荷なゲームアプリケーションの実行時と同様ユーザがバッテリー残量に対し配慮するのが自然である．学習においてはインタラクティブな挙動はなく，長時間にわたる処理が必要となる．このような処理はユーザが端末を操作しない就寝時に動作させることが妥当であり，その場合には充電器へ接続しているため問題は生じにくいと考えられる．

SETI@Home にて端末の潜在的な計算能力を活用することの有用性は広く知られたが，計算用のアプリケーションのインストールが必要であり，これが一般的なパソコン利用者にとっては参加の障壁となった．インストール作業を不要とすることはより多くの端末を利用するために有用であるとの考え方から，Web ブラウザをプラットフォームとして Volunteer Computing を行うシステムが提案されている [24][25][26]．

インターネットサービス Kaggle [4] においては，企業，公共機関が画像データを提供し，それを認識する精度を競うコンペティションが多数開催されている．画像認識だけが課題ではないが，機械学習を用いることを想定した課題に多額の賞金が賭けられ，100 万人以上のユーザが参加している [5]．このように，公開可能な画像データに基づいて DCNN 等のモデルを学習させる需要が存在する．個人がこのようなモデル学習の試行錯誤を行う場合，計算資源が潤沢ではないことが課題になる．計算資源の調達方法として，Volunteer Computing によってインターネット上の人々から計算資源を借りるという方法が考えられる．クラウドの計算資源にかかるコストは電気代と比べて高額であるため，資金を募ってクラウドを借りるよりも経済的な合理性が存在する．通信回線の速度がボトルネックになってしまう点で現時点では実用的でないものの，2020 年以降に実用化が目指されている携帯電話の 5G 回線は 10Gbps を目標として開発が進められているほか，公衆から広く資金調達を行うクラウドファンディングの一種として計算資源の調達を受けるというスキームが考えられ，将来的にインターネットを介した計算資源のやりとりがなされる可能性がある．

[4] https://www.kaggle.com/
[5] http://blog.kaggle.com/2017/06/06/weve-passed-1-million-members/

Federated Learning [27] は，世界中のスマートフォンに蓄積されたデータを用いて DNN モデルを学習する枠組みである．応用先の例として，文章入力における予測変換モデルが挙げられている．ユーザが予測変換候補の中から選択した候補を正解とし，教師あり学習によってモデルを更新する．Federated Learning における主眼は効率的な計算ではなく，ユーザの端末上に蓄積されたデータを活用するため，プライバシー保護のために秘密計算アルゴリズムをベースとした通信内容の暗号化や，各端末あたりの通信量を抑えるための強力な圧縮にある．

3.3 ブラウザコンピューティング

本論文において科学技術計算を専用のサーバではなく，消費者向けのパソコン・スマートフォン端末上の Web ブラウザをプラットフォームとして実行することを「ブラウザコンピューティング」と呼ぶ．端末上で計算を行わせるうえで，特別なソフトウェアのセットアップを必要とせず，通常の Web ページと同様 Web ブラウザにおいて特定の URL へアクセスするだけで計算を行えることを特徴とする．英語において browser computing [28]，machine learning in the browser [29]，volunteer computing in browsers [25] などの表現があるが，日本語での文献は乏しく [30][26]，また Web ブラウザを科学技術計算に用いることに対する用語は確立していない．

本論文では，Deep Neural Network の中でも特に画像分類に用いられる Deep Convolutional Neural Network (DCNN) を主に扱い，これを用いたアプリケーションの開発と利用の両面において Web ブラウザをインターフェースとして端末を計算資源として活用可能にする．ブラウザコンピューティングを用いたアプリケーションとして，Web ブラウザからアクセス可能なセンサーデバイスとして大容量の情報を生成するのはカメラであり，これをネットワークを介して転送せずに処理できる点で最もブラウザコンピューティングが有効であると考えられるためである．他の大容量データとしてマイクからの音声情報も存在するが，これを利用する最も重要なタスクである音声認識の機能はほとんどの OS に標準搭載されている．また，入力画像に類似した画像を膨大なデータベースから検索するようなサービスでは，データベースを端末に持たせることは記憶媒体・通信量上現実的ではない．すべての科学技術計算を端末上で行うのではなく，画像特徴量の抽出のみを端末側で行い，データベースとの照合はクラウド側で行うなど，課題の性質に応じてクラウドとの協調が望ましい．

3.4 Web ブラウザ

Web ブラウザは，ネットワーク上から文書や画像，音声等を読みこみ，画面上に表示するソフトウェアである．ハイパーリンクにより複数の文書同士が関連づけられ，それらの間を移動できる機能が特徴的である．

初期の Web ブラウザは静的なページを表示する機能しかなかったが，クライアント側でスクリプトを実行する機能が付加された．スクリプト言語として JavaScript が開発され，これは通信等の入出力には制約があるものの，任意のアルゴリズムを実装可能 (チューリング完全) な仕様をもつ．この機能を用いて，Web ブラウザ上でインタラクティブに動作する地図，ワードプロセッサ，ゲーム等の Web アプリケーションが多数開発されている．

Web ブラウザはあらゆる端末に初期搭載されており，Web アプリケーションは手入力，E メール，QR コードなどに含まれた特定の URL を開くだけで必要なデータが自動的に読み込まれ動作するため，パソコンの操作に詳しくないユーザであっても簡単に利用することができるものが多い．またユーザにとって，アプリケーションのインストールは不便なだけでない．例えば Android においては，アプリケーションに不適切なアクセス権限を付与してしまうと端末内の情報を外部に送信するなど，重大なセキュリティ上の脅威が発生することが警告されている [6]．Web ブラウザにおいては，セキュリティ上の分離が強く意識されており，JavaScript から端末内のデータへアクセスすることはできず，また Web サイトのドメインごとにデータが分離されており，不適切な情報へのアクセスはできなくなっている．そのため，Web ブラウザ自体に脆弱性がない限り Web アプリケーションを利用することによる情報漏洩は発生しにくい設計となっている．アプリケーションを気軽に試すという観点において，アクセス権限によるセキュリティ上のリスクをユーザが意識する必要がない Web アプリケーションは大きな利便性を持っている．

JavaScript のほか，クロスプラットフォーム性を重視したプログラミング言語である Java 言語で開発された Java アプレットを Web ブラウザ上で動作させる仕組みも開発され，こちらも任意の計算を行うことができた．これを用いた科学技術計算システム [31] が開発された時期があったものの，2019 年時点では Java アプレットは古いテクノロジとみなされ，廃止予定となっている [32]．JavaScript と Java は名称が似ているものの，文法およびライブラリ等のエコシステムが全く異なる言語である．また，Adobe 社によるゲーム・動画コンテンツ作成技術 Flash も，Web ブラウザ上でクロスプラットフォームにスクリプトを動作させることができ，科学技術計算に応用することが技術的に可能であった．しかし JavaScript の表現力の向上，iOS でのサポートが行われなかったこともあり，2020 年末に終息する予定となっている．

3.5 JavaScript による科学技術計算

本節では，Web ブラウザ上で動作するプログラミング言語である JavaScript の特徴について，科学技術計算に応用することを念頭に解説する．

JavaScript は動的型付けのスクリプト言語である．数値，文字列，配列等の基本的なデータ構造を持つ．数値型は 64 ビット浮動小数点数のみであり，整数型は存在しない．インタプリタにより実行されるため，C 言語などのコンパイラ言語と比較して低速な傾向にある．シングルスレッドを前提としており，家庭用のコンピュータにおいてもすでに普及しているマルチコア CPU の性能を活用することは難しい．このように，JavaScript は科学技術計算を行うのに向いているとは言い難い側面が強い．しかしながら，Web ブラウザ上で動作することにより，科学技術計算を行うアプリケーションを明示的にインストールすることなく，特定の Web ページを開くことで直ちに実行可能となる側面は非常に強いメリットとなる．以下に，素因数分解を行うアプリケーションのプログラム例を示す．

`index.html`:

[6] https://www.keishicho.metro.tokyo.jp/smph/kurashi/cyber/security/cyber414.html

```html
<!DOCTYPE html>
<head>
  <meta charset="UTF-8">
  <title>Prime Factorization</title>
  <script src="prime.js"></script>
</head>
<body>
  <input type="text" id="inputNum">
  <button type="button" id="compute" onclick="compute()">
    Compute
  </button>
  <div id="result"></div>
</body>
</html>
```

prime.js:

```javascript
function compute() {
  // Get input number from GUI
  let inputTextBox = document.getElementById('inputNum');
  let inputValue = Number(inputTextBox.value);

  // Do prime factorization
  let factors = [];
  let divider = 2;
  while (inputValue > 1) {
    if (inputValue % divider == 0) {
      inputValue = inputValue / divider;
      factors.push(divider);
    } else {
      divider++;
    }
  }

  // Print result to GUI
  let resultText = factors.join(' * ');
  let resultBox = document.getElementById('result');
  resultBox.innerText = resultText;
}
```

このプログラムを Web ブラウザに配信するには，HTTP サーバが必要である．HTTP サーバは，HTTP プロトコルによりリクエストされたファイルをクライアントである Web ブラウザに送信したり，クライアントから送信されたデータに応じて何らかの処理 (例えばディスクへの保存) を行う．

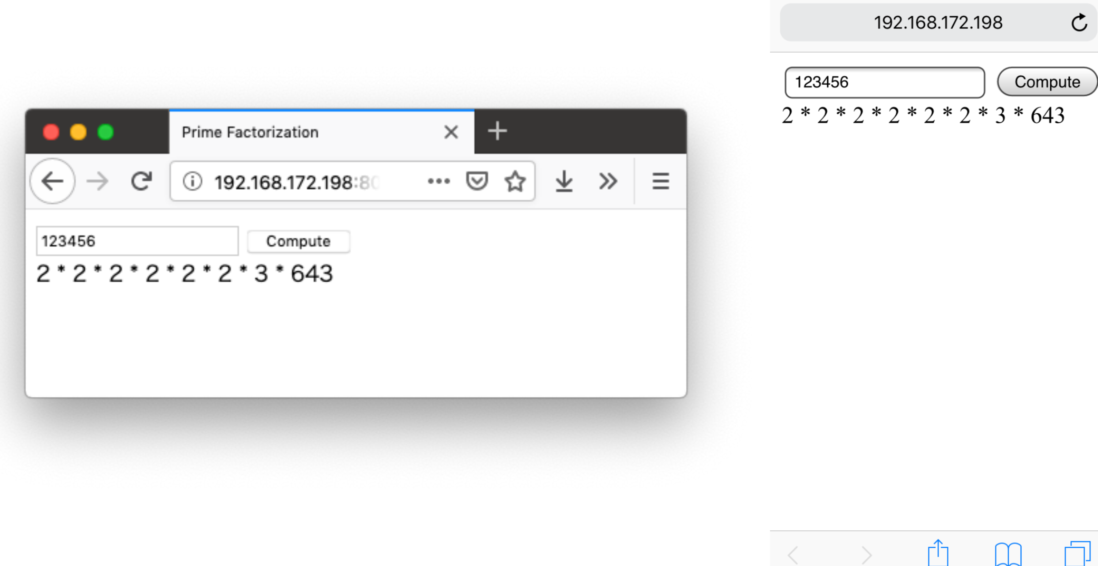


図 3.2 Web アプリケーションの例の動作画面．左: Mac OS・Firefox ブラウザ，右: iPhone SE・Safari ブラウザ上での表示．

実行結果を図 3.2 に示す．画面のサイズに起因する差異はあるが，パソコン・スマートフォンで同一のプログラムを実行結果を得ることができる．ユーザが URL を入力またはリンクをクリックすることにより，Web ブラウザはまず HTTP サーバから HTML コード `index.html` をロードし，解釈する．解釈結果をもとに，画面上にテキストボックス，ボタンなどを配置する．`index.html` から参照されている，JavaScript 言語で記述されたスクリプト `prime.js` を HTTP サーバからロードし，解釈する．これでプログラムの実行準備が完了する．ユーザがテキストボックスに数値を入力し，実行ボタンをクリックする．ボタンに紐づけられた JavaScript の関数 `compute` が呼び出され，テキストボックスに入力された数値を JavaScript 側から読み取る．JavaScript 上の数値変数として解釈された入力データに対し，素因数分解を行う．結果が得られたらテキストボックスに表示する．素因数分解のアルゴリズム自体は C 言語で記述する場合と大きく変わらないが，入出力はファイルではなく，Web ブラウザ特有の API を用いて行うこととなる．

入出力の API は，画面上のユーザインターフェースへの入出力だけではなく，インターネットへのアクセス (HTTP サーバへのデータ送信) も動的に行うことが可能である．この機能を用いることで，文章をオンラインの複数のユーザで同時に編集できるワードプロセッサアプリケーションである Google Document など，インターネット上の他の端末と協調するアプリケーションが実現可能となる．本研究では，この機能を用いて複数の端末を用いた分散計算システムを実装する．ただし，通信可能なホストはセキュリティ上の制約があるため，任意の Web サイトを訪問してクローリングを行うようなアプリケーションは実現できない．特定のホストからしか閲覧できない Web ページの内容を窃取するなどの攻撃を防ぐためである．データの入

力元としてカメラからの画像，マイクからの音声，端末の加速度なども生の数値列として取得することが可能であり，これらを用いて Web アプリケーションを開発することができる．

Web ブラウザはパソコン・スマートフォンのほぼすべてのデバイスに搭載されている．デバイスの OS は主に Windows，MacOS，Android，iOS が用いられ，それぞれの OS で主流のプログラミング言語や，GUI 開発，GPU アクセラレーションなどの手段は大きく異なっており，クロスプラットフォーム開発は高コストである．Web ブラウザは各デバイスに対応したものが開発されており，いずれも JavaScript を実行可能である．画面表示を制御する API (Document Object Model) も原則として統一されており，クロスプラットフォームなアプリケーションを開発するプラットフォームとして非常に有用である．さらに，アプリケーションは明示的なインストールが不要であり，特定の URL を開くだけで直ちに実行されるためユーザにとってのハードルも非常に低い．

JavaScript は長らく Web ブラウザ上でのみ活用されてきたが，2009 年に node.js フレームワーク [33] が提案され，サーバサイドのサービス開発にも用いられるようになった．node.js 環境での JavaScript ライブラリ管理を行うシステム npm (node package manager) が開発され，Web ブラウザ環境でも活用できるように発展しており，本研究でのライブラリ開発に利用している．

JavaScript は動的型付けであり，実行するまで変数の型が不明である．このことは大規模開発においては型の不一致などに起因する実行時エラーを発生させる要因となる．この問題に対処するため，JavaScript に型定義機能などの拡張を施した言語である TypeScript [34] が Microsoft により提案された．TypeScript で書かれたプログラムはトランスパイラ (コンパイラの一種で，ソースコードを入力としソースコードを出力するもの) により型チェックのうえ JavaScript に変換され，Web ブラウザで実行可能なコードになる．本研究においても，Web ブラウザ上で動作するプログラムの実装の大部分に TypeScript を用いている．なお，トランスパイラは型定義を取り除いて JavaScript として妥当なソースコードへの変換を行う機能が主であり，実行速度の高速化には寄与しない．

3.6 Web ブラウザにおけるハードウェアアクセラレーション

JavaScript は上記のように科学技術計算を行える最低限の言語仕様および通信機能を備えており，本研究で対象とする深層学習が実装可能である．一方，深層学習は GPU を用いて計算を行うことが一般的である．深層学習において最も計算量が大きいのは行列積の演算であり，GPU ではこれを高速に実行することが可能である．JavaScript は CPU 上でのみ実行可能であるが，Web ブラウザ上で 3D ゲームを実装する需要の高まりにより，JavaScript から GPU へアクセスする API が実装されている．

本節では，JavaScript の実行速度を超える速度で深層学習を行うことが可能となる API について紹介を行う．具体的な利用方法については，第 4 章にて実際のコードを用いて解説する．各 API は 2019 年現在も発展中であり，新機能が積極的に追加されている．また，Web ブラウザの種類により，実装されている API が異なる．Windows7 に標準搭載されている Web ブラウザ

は Internet Explorer である一方，iOS では Safari しか動作しない[7]．Internet Explorer は WebGL による浮動小数点テクスチャが利用可能である一方，WebAssembly には対応していない．逆に Safari では WebGL による浮動小数点テクスチャが利用不可能である一方，WebAssembly は動作する．現状，多くのデバイスにおいてハードウェアの性能を引き出すためには，複数の API をサポートすることが必須である．

3.6.1 WebGL

WebGL は，2007 年ごろより Web ブラウザに搭載され始めた API であり，JavaScript から GPU を利用できる最初の API である．クロスプラットフォームなグラフィックス API 規格である OpenGL のサブセットとして API 体系が構築されている．3D グラフィックスを実現するための API であり，それを前提とした API 体系となっているため，WebGL では基本的に計算結果を画面にレンダリングする構成となっているが，メモリ上のテクスチャ (3D オブジェクトに貼り付けて質感を表現するための画像) を読み込み，計算結果を別のテクスチャに書き込む機能が存在する．これを用いることで，GPU を科学技術計算へ応用することができる．テクスチャはあくまでグラフィックス用途を想定しているため，そのアクセスは必ず 2 次元の座標で指定することとなり，補間機能があるため浮動小数点数で座標が指定される．また，カラー画像を想定しているため 1 ピクセルごとに RGBA の 4 つのカラーチャンネルが存在する．このような制限は，4 次元のテンソルに対して畳み込み演算を実装するような課題において，非効率性を生じる．

WebGL には大きく分けて WebGL1 と WebGL2 というバージョンがあり，また Web ブラウザによって対応機能の差異がある．WebGL1 の最低限の機能はほとんどの端末の Web ブラウザで利用可能であるが，Safari においてはテクスチャに浮動小数点数を使用することができず，8 ビットの整数に限定されるという制限がある．WebGL2 では，R チャンネルのみのテクスチャを作成することができ，必ず RGBA の 4 要素をまとめてアクセスする必要がないよう緩和されている．

WebGL において，GPU 上で実行するプログラムはシェーダと呼ばれる．シェーダは C 言語に類似した文法をもつ GLSL 言語により記述する．グラフィックス向けのパイプラインを応用することにより，行列計算を行わせることができる．より具体的には，画面に表示するピクセルの色を計算するフラグメントシェーダを利用する．フラグメントシェーダには，出力すべきピクセルの座標および複数のテクスチャが与えられ，ピクセルの色を出力する．処理の過程でテクスチャ上の座標を計算し必要な要素を読むことができるため，入力行列を読み取って計算を施し，出力することができる．出力は通常画面に表示されるが，その代わりにテクスチャに対して書き込むことが可能である．入出力テクスチャを変更しながらフラグメントシェーダを実行することにより，多数のステップからなる計算が可能となる．

[7]Chrome ブラウザも提供されているが，Windows 環境と異なり iOS 環境では JavaScript の解釈等を行うエンジン部分は Safari と同じである．

3.6.2 WebMetal

WebMetal [35] は，macOS および iPhone へ標準搭載されている Web ブラウザである Safari に対し実装された，GPU を利用できる API である．2017 年に提案され，Safari の実験的機能として搭載されている．新しい API が提案される背景として，WebGL はハードウェアの抽象化レベルが高く，性能を十分に発揮できていないという考え方に基づいている．新しい API は WebGPU という名称で Google，Apple，Mozilla などの関係者により標準化がすすめられている [36]．WebGPU 規格の Proof of Concept として Apple 社が提案したものが WebMetal である．発表時点では WebGPU という名称で提案されたが，Apple 社以外がこの規格そのものに追従することはなく WebGPU という名称で別の規格の制定が進み，2019 年に WebMetal に改称された．本論文では混乱を避けるため，時期にかかわらず WebMetal の名称で統一する．

WebMetal においても主なフォーカスはグラフィックスであるが，コンピュートパイプラインと呼ばれる汎用的な計算用途に適したインターフェースが含まれており，WebGL に存在した非効率性を回避することが可能である．仕様として，Apple 社独自の GPU API である Metal を Web ブラウザ上からアクセス可能にするラッパーとしての特徴が強く，他社デバイスでの実装にはハードルが高い内容となっている．シェーダ言語は Metal 言語が利用され，これは WebGL における GLSL とは互換性がない．

3.6.3 WebGPU

WebGPU は，次世代 GPU API として標準化が進められている規格である．OS ごとにローレベルの GPU API が異なっているため，この差異を吸収しつつ高いパフォーマンスを確保し，さらにセキュリティを担保するという点が難題となっている．2019 年現在，シェーダ言語は決定していないが，WebGL が用いているものより新しいバージョンの OpenGL のシェーダ言語を利用するプロトタイプが Google Chrome のベータ版において実装が開始されている．

3.6.4 WebAssembly

JavaScript は，言語処理系の性能が Web ブラウザの動作の快適性に影響し，Web ブラウザ開発業者間の競争があるため，言語仕様上可能な範囲で実行速度の最適化が積極的に行われている．しかしながら，動的型付け言語であることから C 言語などと比較して最適化が困難である．そこで，クロスプラットフォームな仮想マシンを定義し，そのうえで動作する低水準な言語およびコンパクトなバイナリ形式として WebAssembly が提案された．C 言語や C++言語をコンパイルし，WebAssembly を出力するコンパイラ Emscripten が提供されており，これを用いることでネイティブアプリケーションに近い速度で CPU 上で計算を行うことができる．ただし，CPU の SIMD 命令やマルチコアの利用は現状行えないため，科学技術計算用途において全く同等の性能を得るまでには至っていない．

JavaScript はシングルスレッドを想定した言語であるが，Web Worker と呼ばれる API が存在し，これを用いることで複数の JavaScript プログラムを同時に実行することが可能である．ただし，JavaScript からみたメモリ空間は独立しており，メッセージベースの明示的な通信に

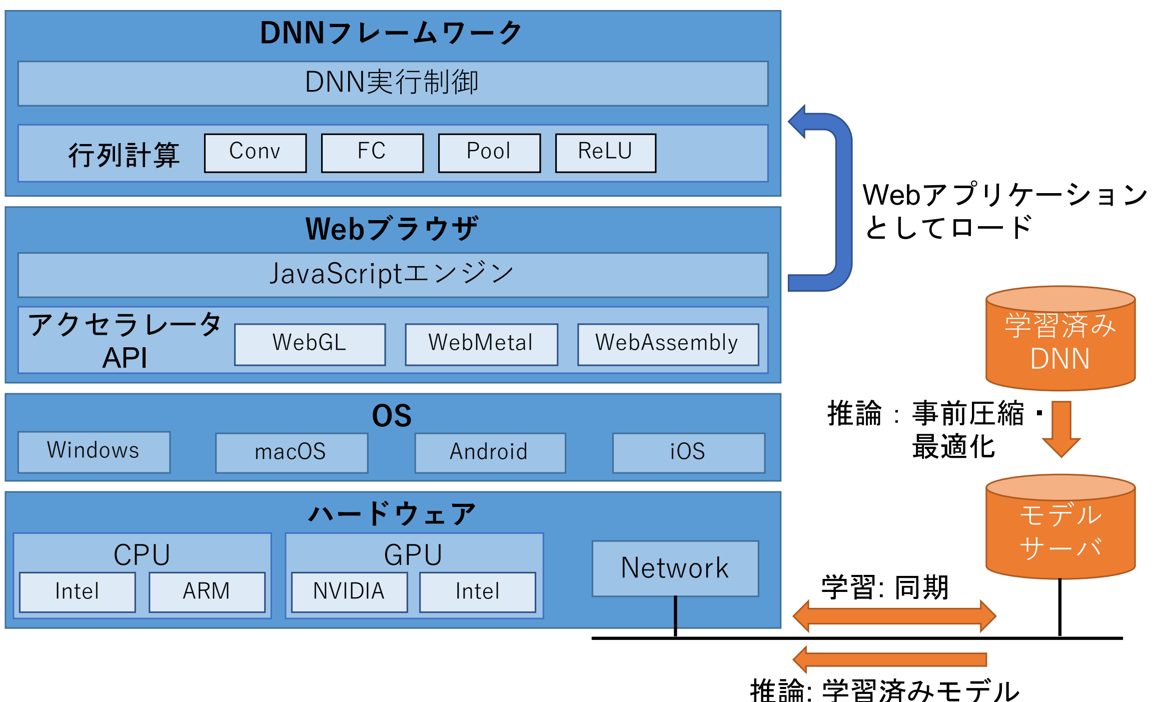


図 3.3 Web ブラウザによる DNN 学習・推論システムのソフトウェアスタック

よってのみ連携を行うことが可能である．メモリ空間の共有機構として SharedArrayBuffer と呼ばれる API が提供されていた時期があるが，CPU の脆弱性 Spectre を悪用する攻撃への懸念から無効化された経緯がある [37]．このように，ハードウェアの性能を発揮させることとクロスプラットフォーム性・セキュリティの両立は容易でなく，各種規格の制定と普及には時間がかかる．そのため実行性能重視のアプリケーションを開発するためには環境ごとに API を使い分けることも必要となってくる．

3.7 ソフトウェアスタック

ソフトウェアスタックを図 3.3 に示す．本研究において DNN の学習・推論を行うフレームワークを Web アプリケーションとして実装し，これを Web ブラウザが HTTP サーバからロードすることにより一連の処理が開始する．Web ブラウザにより，OS およびハードウェアを抽象化する．ネットワークを通じて処理対象のモデルパラメータを読み込み，DNN モデルの構造に従って行列計算を実行する．Web ブラウザにより対応しているアクセラレータ API が異なるため，これを使い分けることにより各ハードウェアの性能を引き出す．Web ブラウザ上のソフトウェアスタックのほかに，学習における計算ノードを統括するサーバおよび，推論におけるモデルパラメータを圧縮・配信するサーバが存在する．

Chapter 4

Web ブラウザを計算ノードとした Deep Neural Network の分散計算

4.1 序論

本章では，科学技術計算向けではないコンピュータを低コストで連携させて深層学習を行わせることを目的とし，Web ブラウザを搭載したデバイス (パソコン・スマートフォン) を計算ノードとして DNN の学習を分散計算するフレームワークを提案する．デバイスには JavaScript が実行可能な Web ブラウザが搭載されていることを仮定し，特定の URL を開くだけで計算ノードとして参加できる仕組みを構築する．また，同一機種のデバイスが揃えられていることを仮定するのは非現実的であるため，デバイス間に性能差が存在する場合でも効率的な処理が可能となるようなアルゴリズムを検討する．

JavaScript から GPU を利用できる API として，WebGL が多くの Web ブラウザに実装されている．WebGL はグラフィックス向けに設計されており，行列計算への応用が可能ではあるものの GPU 本来の性能を発揮することが難しい．2019 年現在，次世代 API として WebMetal，WebGPU が発展途上である．これらはコンピュートシェーダと呼ばれる汎用的な計算向きのインターフェースを持っており，行列計算を高速に行えるポテンシャルがある．一方でこれらは一部の Web ブラウザにのみ実験的に搭載されており，多数のデバイスの性能を引き出すという観点からは，複数の API への対応が必要となる．

Web ブラウザを介してデバイスを計算ノードとする分散計算フレームワークはいくつか提案されている [38][39] が，基本的に map 処理の並列化を行うものである．map は独立した入力 $x_1, x_2, \ldots, x_G$ に対し同じ関数 f を適用し $f(x_1), f(x_2), \ldots, f(x_G)$ を生成する処理である．DNN の学習に適用する場合，N サンプルからなるミニバッチを N/G サンプルずつのミニバッチに分割し，それぞれについて勾配を計算することで map 処理に帰着させることができる．モデルの更新に当たっては分割された勾配の総和を取る必要があり，これは reduce 処理の一種である．

同一性能のハードウェアを連携させたコンピュータクラスタにおいては，上記の map 処理において計算ノード数を G に代入して分散処理が可能となる [40]．map・reduce を行うための通信機構の実装に当たってはハードウェアベンダ等による MPI ライブラリが存在しており，

DNN を分散計算させるために行う特別な実装は少ない．一方，計算ノード間に性能差が存在する環境では，ミニバッチを均等に分割すると，低性能なデバイスの処理を高性能なデバイスが待機することとなり，大きな非効率となる．ミニバッチをデバイス数より細かく分割し，1 単位の処理が完了するたびに次の単位の処理を行うことで，高性能なデバイスがより多数の処理を担う仕組みが考えられる．しかし，高性能なデバイスにおいて小さな単位で行列計算を行うことは非効率という課題がある．GPU は一回の計算を開始する際の遅延が CPU より大きい点と，処理が少ないと多数存在する演算コアが余ってしまい，利用効率が低くなる．高性能なデバイスを活用するためには，極力大きな単位で計算を与えることが必要である．そのため，従来の Web ブラウザ向け分散計算フレームワークは深層学習に対してあまり効率的とは言えない．

また，深層学習に用いられるコンピュータクラスタにおいて，GPU は NVIDIA 社の製品が圧倒的なシェアを誇っており，汎用計算プラットフォームである CUDA，これを用いた行列演算ライブラリ cuBLAS，DNN の畳み込みなどを高効率で行うライブラリ cuDNN などが提供されており，これらを活用することでハードウェアの性能を引き出すことができる．本研究で用いるデバイスの GPU は NVIDIA 製とは限らず，また JavaScript からは CUDA を利用することができない．そのため，行列演算に関する処理を独自に実装することが必要となる．

4.2 関連研究

4.2.1 Web ブラウザにおける DNN の学習

ConvNetJS [41] は，Web ブラウザ上で深層学習を行うことができるライブラリである．単一端末上で完結することを志向しており，分散計算に関する機構はない．また，GPU を利用する機能がないため小規模なモデルにしか対応できない．MLitB [29] は，Web ブラウザをプラットフォームとして深層学習の分散を行うシステムである．mapreduce 方式で実装されており，計算ノードの性能差については所定の時間内に計算できたサンプル数分だけ勾配を送信するという方式で吸収している．計算の処理単位が小さくなるため，端末性能の利用効率が低下する．また，GPU の利用はなされていない．評価実験の結果が掲載されておらず，性能は不明である．Hidaka ら [42] は，Web ブラウザから GPU を利用できる API である WebCL [1](WebGL とは異なる) を用いて高速な行列演算ライブラリを実装することにより，大規模な画像分類モデルである VGG16 を分散学習した．WebCL はマルチコア CPU，GPU 等に対応するクロスプラットフォームなアクセラレータ API である OpenCL のラッパーとして Khronos Group により提案された．Firefox 用のプラグインとして動作するプロトタイプ実装が存在したものの，2019 年現在 Firefox のプラグイン方式が変更されており，もはや動作しない．WebCL はデータセンター向けの強力な GPU を活用できたため，大規模な DNN の学習に向いていた．一方モバイル端末では依存先の OpenCL に対応していない場合が多く，モバイル向けの Web ブラウザに API を実装することは困難な仕様であった．分散計算の方式は，ミニバッチを計算ノード数で均等に分割するものであり，計算ノード間の性能差は考慮されていない．

[1] https://jp.khronos.org/webcl/

4.2.2 Web ブラウザにおけるアクセラレータの活用

Web ブラウザにおいて，3D グラフィックを実現するための API として WebGL が存在する．WebGL を用いるライブラリの多くはグラフィックス向けである一方，WebGL を用いて行列演算を行うことを志向した試みを挙げる．weblas [43] は，WebGL を用いて行列積等の演算を行うシンプルなライブラリである．DNN で必要となる演算すべてを網羅しているわけではない．gpu.js [44] は，JavaScript のサブセットで記述されたコードを，WebGL 上で動作するように動的に変換するライブラリである．JavaScript の構文解析エンジンを内蔵し，WebGL のシェーダ言語へと変換することで実現している．JavaScript のみでアプリケーション開発が完結するため，Web 開発者にとって新たな言語習得が不要であるというメリットが大きい．一方で主としてグラフィックス用途を志向しているためか，行列は 3 次元までしか扱えず，また汎用計算に適した新しい GPU API への対応もされていない．本研究では，Python のサブセットで記述された行列演算コードを Web ブラウザの WebGL および WebMetal で動作するように変換するライブラリを提案する．本研究で目指す DNN の分散計算においては，Python 言語を多くの部分で用いることが妥当である．計算ノードとなる Web ブラウザに対してデータを供給するには Web サーバが必要であり，これは Web ブラウザ自体では実現できないため，Web ブラウザ以外のプログラミングがいずれにせよ必須である．サーバ側で動作する JavaScript 処理系として node.js が存在するものの，データセットの入出力・前処理に関して Python のほうが優れたエコシステムを持っている．そのため Python 言語でサーバ側の処理を記述しつつ，計算ノード上で動作する計算コア部分を Web ブラウザで実行可能な形式に変換する構成をとる．

4.2.3 ヘテロジニアス計算環境における DNN の分散学習

MoDNN [45] は，ネットワークで接続された複数の端末を用いて DNN の推論を高速化する．デバイスの性能，通信速度に応じてモデル行列を分割およびスパース化し，速度の向上を図っている．あらかじめデバイス数・性能が判明している前提でモデルを分解する点，モデル行列が不変となる推論のみをターゲットとしており本研究とは異なる．

Yang らは，異なる機種の GPU を搭載したサーバからなるヘテロジニアスなクラスタシステムにおいて，すべてのサーバに同じバッチサイズの勾配計算を割り振ると低速なサーバでの処理に他のサーバと比較し時間がかかり，パフォーマンス上のボトルネックとなることを指摘した [46]．他のサーバの処理を待たせる低速なサーバを straggler worker と呼び，この問題を解消するためサーバの性能に応じたバッチサイズを配分することを提案した．具体的には，サーバの i のバッチサイズ β_i に対する処理時間を $A_i\beta_i + b_i$ と一次関数でモデリングし，サーバの性能を測定して線形回帰により求める．そして，全サーバの合計バッチサイズ β_G を入力し，各サーバのバッチサイズを表すベクトル $\boldsymbol{\beta} = (\beta_1, ..., \beta_N)$ を以下の式の最適化により求める．

$$\min_{\boldsymbol{\beta}} \max_i (A_i\beta_i + b_i)\, s.t. \sum \beta_i = \beta_G, \beta_i \geq 0 \tag{4.1}$$

β_i は整数であり，この問題は整数計画法のソルバーを用いて求めることが可能である．直観的には，すべてのサーバが同じ時刻に処理を終えられるような処理の配分を行うこととなる．

Chen らは，ヘテロジニアスなクラスタシステムにおける straggler problem に対し，GPU からなるクラスタと CPU からなるクラスタで異なる解決策を提案した [47]．GPU からなるクラスタでは Yang らとほぼ同じモデルを採用している．一方，CPU からなるクラスタでは，他の計算タスクとサーバを共用する状況を想定している．この環境においては CPU の負荷状況により動的に処理時間が変化するため，これに対応する必要が生じる．そのため，システム全体の CPU 利用率・メモリ利用率の時系列を取得し，次回のイテレーションにおける計算速度を予測する Nonlinear AutoRegressive eXogenous モデルを提案し，これを用いることで CPU 負荷が変動する環境における学習速度の最大化を図っている．

これらの手法では，サーバの台数，おおむねの性能が判明している状況での最適化を行っている．また，多少の処理時間の変動はあっても切断されることはないという暗黙の仮定がおかれている．本研究では，計算に用いるデバイスが最低 1 台から動作し，動的に参加，切断されることを想定した手法を提案する．

また，同じ性能のサーバを用いたクラスタシステムにおいても，様々な影響で確率的にいくつかの計算ノードの処理が遅延し [48]，同期更新を用いる場合には性能低下の原因となる．その解決策として，N 台の計算ノードがある場合に $N-b$ 台分の勾配を受信した時点でモデルの更新を行い，残りの b 台分は破棄するという backup worker 方式が提案されている [49]．遅延の度合いにかかわらず一定の計算コストが無駄になるため，改良の余地があると考えられる．

4.3 手法

ブラウザコンピューティングにおいて深層学習の効率的な分散計算を実現するためには，各デバイスのハードウェア性能を引き出すことおよび，他のデバイスの待機等の分散計算に伴い生じるオーバーヘッドを最小化することが必要である．本研究では，各デバイスの性能を引き出すため行列演算コードを GPU 上で動作するプログラムに変換するシステムと，性能の異なるデバイス上で計算グラフの分散計算を行うための分散計算システムを開発し，これらを組み合わせる．

本研究においては，1 台のサーバに分散計算ノードとなる複数のスマートフォン等の端末を接続し，DNN の教師あり学習を行う．端末側には特別なソフトウェアのインストールを要さず，Web ブラウザを用いて，サーバ上で稼働する HTTP サーバに接続することで計算ジョブを受信するものとする．Web ブラウザ自体は他の Web ブラウザに対して Web ページを配信することができないため，HTTP プロトコルにより計算プログラムを含んだ Web ページおよび動的にジョブを配信する専用ソフトウェアをインストールしたサーバが少なくとも 1 台必要となる．端末は学習中の任意のタイミングで接続・切断することを想定する．端末の計算性能は端末ごとに異なり，動的に推定する必要がある．また，高負荷な処理を連続して行うと発熱を抑えるため性能が低下する場合があり，計算性能は非定常である．サーバ側はデータセットの読み込み，通信を担当し，各端末からの勾配の平均を計算する程度の処理負荷となり，GPU は利用しない．

4.3.1 コード変換による複数アクセラレータの活用

本研究では，端末の性能を発揮させるため，Python のサブセットで記述された行列処理コードを Web ブラウザ上で GPU を利用して動作可能な形に変換するシステムを提案する．これは，深層学習基盤における行列計算部分の高速化に対応する．

システムは，次のような目標に従って実装する．

- Python 言語としてそのまま実行可能であること．
- 多次元の行列計算のための記述を容易にすること．
- WebGL だけでなく，次世代 GPU API へも変換可能であること．

本研究では，Web ブラウザ上で GPU を利用するための API(バックエンド) として，WebGL および WebMetal に対応する．また，CPU 上で高速計算可能な API として WebAssembly に対応する．WebGL・WebMetal は機械語・バイトコードではなく C 言語に類似した高級言語 (GLSL・Metal) のみを受け取るため，機械語を出力する一般的なコンパイラではなくソースコードを出力するトランスパイラを開発する必要がある．WebAssembly はバイトコードの形式をとるが，Emscripten ツールにより C 言語からコンパイルできる．ソースコード生成部の共通化を行うことおよび，ツールに搭載された実行速度最適化機構を活用するため，他のバックエンドと同様に C 言語のソースコードを生成することとする．Web ブラウザでは使用できないが，Windows 上のネイティブアプリケーションで利用可能な HLSL 言語や，NVIDIA の GPU で利用可能な CUDA C 言語なども C 言語ベースであり，同様の仕組みが適用しうる．

構文の対応付け

Python 言語のすべての機能を GPU 上で実行可能とすることは現実的ではないため，深層学習を実装するにあたり必要な機能に絞った変換機構を開発する．

テンソル

深層学習の計算グラフにおいて，データは多次元テンソル (以下，次元数に関わらず，行列と表記する) で表現される．各バックエンドは指定サイズのメモリ領域 (バッファ) を確保する機能があるため，1 つの行列につき 1 つのバッファを対応付けることとする．C 言語ではコンパイル時に形状が確定しない多次元配列を直接的に扱うことはできないため，1 つの行列に対応する 1 次元のメモリ領域をデータの単位とし，多次元のインデックスで要素を操作する機構が必要である．計算グラフのノード (分散計算の計算ノードとの混同を避けるため，以下ではオペレータと表現する) には 1 つ以上の行列が入力され，1 つの行列を出力するものとする．ここで，バッチサイズを実行時に可変としたいため，少なくとも入力行列の 1 つの次元についてサイズを動的に変化させられることを必要とする．また，出力行列の形状は入力行列の形状とオペレータのオプション (畳み込み層のストライド等) によって定まるものとする．

関数定義

各オペレータの計算内容の記述に必要な言語機能を考える．WebGL において，一度の実行では出力行列の 1 要素のみを出力することができる．そのため，入力行列および出力すべき行

列要素のインデックスを入力とし，スカラー値を出力する関数を記述する方式が適切である．全結合層の勾配計算に対応するオペレータを考えると，入力変数に対する勾配と重みに対する勾配のように 2 つの出力が必要となるが，これは 2 つのオペレータに分解して 1 つのオペレータあたり 1 つの出力を持つとする．これは，WebGL において同時に複数のテクスチャへの出力を生成できないという制約があるため，WebGL のプログラムと計算オペレータを 1 対 1 対応させるためである．このような関数を記述するため，Python における関数定義構文を用いて，`def main(ret: ArrayOut2, in_a: ArrayIn2, in_b: ArrayIn2)` のように入出力行列に対応する変数名およびその次元数を記述する．`ArrayOut2` の部分は Python 3.5 より導入された型アノテーション構文を用い，次元数を明示することとした．また，畳み込み層のストライド等のパラメータを与えるため，追加のスカラー引数を与えることを可能とする．

行列へのアクセス

出力要素のインデックスをもとに入力行列の必要な要素にアクセスする必要があるため，入力行列のインデックスを指定しスカラー値を取り出す文法が必要である．これは numpy において `array[x, y, z, w]` のようにアクセスする構文が用いられており，これを踏襲する．インデックスの計算のため，整数値を取る変数および四則演算が必要である．また，入出力行列の形状が必要なため，numpy と同様 `array.shape[0]` のようにアクセス可能とする．出力インデックスは numpy には対応する構文がないが，`array.ret[0]` のようにアクセス可能とする．

数値計算と変数

入力行列から取り出した値を処理するためには，実数値 (浮動小数点数) を取る変数および四則演算，さらに Softmax 関数で用いられる指数関数等の数学関数が必要である．Python では `math.exp(x)` のように数学関数を呼び出すため，これを各バックエンドの数学関数の名称に置換する．Python では変数は宣言不要で型を持たず，同じ変数に整数，実数のほか配列や文字列なども代入することができるが，C 言語においては型を宣言する必要がある．関数の引数として多次元の配列を受け取るほかは，整数および実数のみを変数として受け入れることとし，最初に代入された型をその変数の型として推論し，矛盾する代入はエラーとする．JavaScript では整数と実数の区別はなく，すべての数値が 64 ビット浮動小数点数で表現される．整数は小数点以下の値が 0 という特殊な場合である．しかし，GPU においては 64 ビット浮動小数点数は利用できないか，性能が極めて低い場合がある．実数を 32 ビット浮動小数点数で表す場合，正確に表現できる整数は $\pm 2^{24}$ までであり，大きな行列の操作ではオーバーフローしてしまう．そのため実数と整数は異なる型であるとする．Python においてはもともと別の型であり，文法に手を加える必要はない．なお，Python においては比較演算の結果は整数とは異なる bool 型であるが，整数と区別する必要性は低いため整数型に統合した．WebGL・WebMetal はグラフィック向けに設計されており，座標変換等に利用される 1〜4 要素のベクトル型変数も宣言可能であるが，任意サイズの多次元テンソルを扱う場面でこれを利用して最適化するのは容易ではないため使用しないこととした．このような機能は実行速度のボトルネックにおいて，Python コードからの変換ではなく各バックエンド固有の最適化されたソースコードを記述するべきである．

制御構文

制御構文は，if・while・for へ対応する．if・while 文は Python と C 言語で同等である．for 文は，Python においてはイテレータオブジェクトの列挙を意味する．例えば `file` がファイルに対応するオブジェクトとした場合，`for line in file` という構文でファイルの各行を変数 `line` に受け取る．このような機構は複雑であり GPU 上での実行に向かないため，数値計算で用いられる `for i in range(start, stop, step)` というイディオムにのみ対応した．これは C 言語における `for (int i = start; i < stop; i += step)` というコードに変換できる．ただし，`step` が負の場合は条件判定が異なるため，最初に `step` の符号により分岐し，2 種類の for 文のコードを生成することとした．ソースコード上は冗長となるが，`step` が定数であればコンパイラの最適化により削除されるため問題ない．

トランスパイラ

トランスパイルのパイプラインを [50] に即して示す．ソースコードをトークン単位に切り分ける字句解析 (Lexical analysis) および構文木を作成する構文解析 (Syntax analysis) においては，入力が Python 言語として妥当なものであるため，Python 言語用の構文解析器をそのまま用いることが可能である．Python 言語処理系の標準ライブラリである ast モジュールを用いて抽象構文木を生成する．意味解析 (Semantic analysis) においては，変数の型を解析する．変数に代入される数値が整数か実数かを判定し，変数名と型の対応テーブルを作成するとともに矛盾がないことを確認する．同時に，未対応の構文はエラーとする．Python の抽象構文木に変数の型テーブルを付与したものを中間表現として利用する．次にバックエンドごとのコード生成を行う．変数へのアクセス，四則演算等の式，制御構文，プリアンブルの出力が必要となる．変数はスカラー値の場合と行列要素のアクセスの場合で構文が異なる．`array.shape[0]` のようなインデックス情報へのアクセスについては，アクセスすべき次元は定数で十分であるため `array_shape_0` のように単純なスカラー変数へのアクセスとして出力する．いずれのバックエンドも基本的に C 言語ベースであるため，スカラー変数のアクセス・四則演算についてはどのバックエンドも同様である．制御構文については，WebGL ではループ回数が定数でなければならないという制約があるため，ダミー変数による 16384 回のループを定義し，ループ内で実際のループ条件を判定して break 文を挿入する．プリアンブルはここでは数学関数利用のための include 文や関数の入出力の型定義，ローカル変数の定義を指し，バックエンドにより異なる．

行列の内容および行列形状等のメタデータを CPU から与え，またソースコード上でアクセスする手段はバックエンドごとに異なり，その記述に関する部分が C 言語との主な相違点となる．変換元となるコードの具体例を示す．このコードは 2 つの行列 `in_a`，`in_b` を入力とし，行列積を計算するものである．`main` 関数が計算の本体であり，これを各バックエンドのソースコードに変換する．`calc_shape` 関数は Python 上で実行されるもので，入力行列の形状から出力行列の形状を計算するものである．バッチサイズに対応する次元のサイズは 0 を代入しておき，実行時にバッチサイズを代入することで動的に計算する．

```
from uniaccel.operator import ArrayOut2, ArrayIn2

def main(ret: ArrayOut2, in_a: ArrayIn2, in_b: ArrayIn2):
    s = 0.0
```

```
    for k in range(in_a.shape[1]):
        s = s + in_a[ret.idx[0], k] * in_b[k, ret.idx[1]]
    return s

def calc_shape(in_a: ArrayIn2, in_b: ArrayIn2):
    assert in_a.shape[1] == in_b.shape[0]
    return [in_a.shape[0], in_b.shape[1]]
```

各バックエンドに対する変換結果を示しながら，本課題におけるバックエンドの利用方法と制約事項を解説する．

上記の Python コードを提案システムで WebGL のシェーダに変換したものは以下のようになる．紙面上での可読性のため，文法上の差異が生じない範囲で適宜整形を行なっている．WebGL におけるコンピュータグラフィックス向けの機構を用いて行列計算を行わせる手法について，簡単に解説する．WebGL では，3D モデルのポリゴンの頂点座標を計算する頂点シェーダと，画面上の各ピクセルの色を計算するフラグメントシェーダの組み合わせでコンピュータグラフィックスを生成する．フラグメントシェーダは，出力画像の各ピクセルごとに同じプログラムが呼び出され，プログラム内で変数 `gl_FragColor` に書き込むことによりそのピクセルの色を出力する．どのピクセルの色を計算すべきであるかは，変数 `gl_FragCoord.x`, `gl_FragCoord.y` として与えられる．このプログラムは C 言語と類似した GLSL 言語で記述することができ，`main` 関数が各ピクセルの計算のために呼び出される．出力先は通常，Web ページ内に設置された canvas オブジェクトであり物理的な画面に計算結果の画像が表示されるが，スクリーンバッファと呼ばれる画面に表示されないメモリ領域に書き込むよう設定できる．フラグメントシェーダに対し，CPU からデータを与える手段として uniform とテクスチャが存在する．いずれも全要素の計算を通じて同じ値を与える．uniform は数個程度のスカラー値 (整数・実数) または 1〜4 要素のベクトル値を与えることができるため，メタデータの設定に用いる．最大個数の保証値は 16 であり，また計算結果に応じて動的にアクセスする先を変える機構がないため，行列の内容を与えるのには適さない．テクスチャは 2 次元インデックスで実数値にアクセスできるメモリ領域であり，多くのデバイスで 4096×4096 要素を格納することができる．1 つのテクスチャを 1 つの行列に対応させて用いる．テクスチャの要素は，`texture2D` 関数にテクスチャを表す変数と要素の座標を指定することにより取り出すことができる．3 次元以上の行列に対応するため，論理的な多次元のインデックスを 1 次元のインデックスに変換した後，テクスチャ上の 2 次元座標に変換してアクセスする補助関数を生成する．同様に，出力座標についても `gl_FragCoord.x + gl_FragCoord.y * textureWidth` で 1 次元のインデックスに変換した後，論理的な多次元のインデックスに変換する．なお，4096×4096 を超える要素数はデバイスによっては対応しておらず，これを WebGL で扱える行列サイズの上限としている．同時に，整数型が $\pm 2^{24}$ までしか扱えないデバイスも存在する．整数の表現に内部的に 32 ビットの浮動小数点型を用いている可能性があり，その場合前述のように $\pm 2^{24}$ より大きな整数は正確に表現できない．この問題のため，$4096 \times 4096 = 2^{24}$ 要素より大きな領域でのインデックスを指し示すことが難しいという制約もある．著名な大規模モデルである AlexNet [8] は 4096×9216 サイズの行列があり，これを学習する場合には関連する行列を分割

する必要が生じ，一般化した実装は非常に複雑となる．これを回避するには連続した大きなメモリ領域を確保できる機構が必要である．深層学習において通常行列に格納されるのは誤差を許容可能な実数値であるが，Max Pooling レイヤーの誤差逆伝播を実装する場合にはどのインデックスの要素が最大値であったかという整数値の情報が必要であり，大きなメモリ領域では 2^{24} を超えてしまう．32 ビット整数を確実に扱えるメモリ領域が必要となる．

```
precision highp float;
precision highp int;
precision highp sampler2D;
int mod(int x, int y)
{
  return x - (x / y) * y;
}
uniform int ret_webgl_texture_w;
uniform int ret_webgl_texture_h;
uniform int ret_shape_0;
uniform int ret_strides_0;
uniform int ret_shape_1;
uniform int ret_strides_1;
uniform sampler2D in_a;
uniform int in_a_webgl_texture_w;
uniform int in_a_webgl_texture_h;
uniform int in_a_shape_0;
uniform int in_a_strides_0;
uniform int in_a_shape_1;
uniform int in_a_strides_1;
uniform sampler2D in_b;
uniform int in_b_webgl_texture_w;
uniform int in_b_webgl_texture_h;
uniform int in_b_shape_0;
uniform int in_b_strides_0;
uniform int in_b_shape_1;
uniform int in_b_strides_1;

float get_in_a(int idx0, int idx1)
{
  int flat_idx = idx0 * in_a_strides_0 + idx1 * in_a_strides_1;
  return texture2D(in_a, vec2(
    (float(mod(flat_idx, in_a_webgl_texture_w)) + 0.5)
      / float(in_a_webgl_texture_w),
    (float(flat_idx / in_a_webgl_texture_w) + 0.5)
      / float(in_a_webgl_texture_h))).r;
}
```

```
float get_in_b(int idx0, int idx1)
{
  int flat_idx = idx0 * in_b_strides_0 + idx1 * in_b_strides_1;
  return texture2D(in_b, vec2(
    (float(mod(flat_idx, in_b_webgl_texture_w)) + 0.5)
      / float(in_b_webgl_texture_w),
    (float(flat_idx / in_b_webgl_texture_w) + 0.5)
      / float(in_b_webgl_texture_h))).r;
}

void main()
{
  int ret_idx_0;
  int ret_idx_1;
  int ret_idx_flat;
  float s;
  int k;
  ret_idx_flat = int(gl_FragCoord.x) +
    int(gl_FragCoord.y) * ret_webgl_texture_w;
  ret_idx_0 = ret_idx_flat / ret_strides_0;
  if (ret_idx_0 >= ret_shape_0)
  {
    return;
  }
  ret_idx_1 = mod(ret_idx_flat / ret_strides_1, ret_shape_1);
  s = (0.0);
  k = 0;
  int loop_stop_1 = (in_a_shape_1);
  int loop_step_2 = 1;
  if (loop_step_2 >= 0)
  {
    k -= loop_step_2;
    for (int loop_0 = 0; loop_0 < 16384; loop_0++)
    {
      k += loop_step_2;
      if (k >= loop_stop_1)
      {
        break;
      }
      s = s + get_in_a(ret_idx_0, k) * get_in_b(k, ret_idx_1);
    }
  }
  else
  {
```

```
    k -= loop_step_2;
    for (int loop_0 = 0; loop_0 < 16384; loop_0++)
    {
      k += loop_step_2;
      if (k <= loop_stop_1)
      {
        break;
      }
      s = s + get_in_a(ret_idx_0, k) * get_in_b(k, ret_idx_1);
    }
  }
  gl_FragColor = vec4((s), 0, 0, 0);
  return;
}
```

このソースコードを実行するための主要な WebGL API の利用手順の概略を解説する．なお，実際にはここで挙げていない多数のオプション設定が必要となる．

1. `gl = document.createElement('canvas').getContext('webgl')` により，WebGL の各機能を呼び出す「コンテキスト」を生成する．

2. `gl.createShader()`, `gl.shaderSource(source)`, `gl.compileShader()` により，シェーダのソースコード `source` をハードウェア固有のバイナリコードへとコンパイルする．ここで，シェーダにはポリゴンの頂点座標を計算する頂点シェーダと，画面の各ピクセルに対応する色を計算するフラグメントシェーダの 2 つがあり，これを順に実行することになる．頂点シェーダでは，画面全体を張るダミーのポリゴン座標を出力するコードを用いる．ポリゴンが存在しないピクセルの色は計算されないためである．フラグメントシェーダでは，先述のシステムが出力したコードを用い，行列計算を行う．

3. `gl.createTexture()` により，行列の要素を格納するメモリ領域であるテクスチャを生成する．`gl.texImage2D()` により，テクスチャの縦横のピクセル数を指定する．本来ポリゴンに貼り付ける画像を格納する領域であるため，多くのデバイスでは 4096×4096 以上であるが，最低保証値は 1024×1024 しかなく，汎用性のある行列計算実装を行うにあたっては制限が厳しい．

4. `gl.texSubImage2D()` により，CPU 上のメモリからテクスチャにデータを転送する．ここで，テクスチャが RGBA チャンネルの画像を保持するようになっており，行列のデータは R チャンネルにのみ格納する場合はダミーデータで GBA チャンネルの領域を埋める必要があり，若干のオーバーヘッドとなる．テクスチャから CPU 上のメモリに計算結果を読み取る際は `gl.readPixels()` を用いるが，同様にチャンネルの整合性を取る必要がある．

5. プログラムの実行結果は通常画面に表示されるが，これをテクスチャのメモリ領域に書き込むためにはフレームバッファと呼ばれる機能を用いる．`gl.createFramebuffer()`, `gl.bindFramebuffer()` を用いてフレームバッファを作成し，`gl.framebufferTexture2D()` によりテクスチャをフレームバッファに関連づける．

6. フラグメントシェーダ内部で読み取ることができるテクスチャは，`gl.bindTexture()` を用いて関連づける．ここで，同一のテクスチャをフレームバッファと読み込み用両方に関連づけることはできない．行列計算においては，要素ごとの ReLU レイヤーなどはあるメモリ領域から読み取り，その結果を同じ領域に書き込めばメモリを削減することができるが，そのような実装ができないことを意味する．

7. `gl.drawArrays()` により，一連のシェーダが実行され，計算結果がテクスチャに書き込まれる．

次に WebMetal について解説する．WebMetal には汎用的な計算を行うために用意されたコンピュートシェーダが存在し，WebGL より自然な記述でアルゴリズムを記述できる．プログラムは C 言語に類似した Metal 言語で記述する．同一のプログラムが，演算コアごとに引数を変えて呼び出される．`[[thread_position_in_grid]]` という属性が付与された変数が呼び出しごとに変化するので，これを計算すべき要素のインデックスであるとみなして処理を変えることができる．コンピュートシェーダでは，CPU からデータを与える手段として uniform とバッファが存在する．バッファは C 言語における通常の一次元配列としてアクセスすることができるため行列の内容，メタデータの設定両方に用いる．バッファのサイズはメモリが許す限り広くとることができ，また整数値は 32bit 幅を持つため WebGL のようなサイズの制約はない．ここでは，浮動小数点型のメタデータを `meta_buffer_float` という配列に格納することとし，CPU 側から行列形状などの値を設定し，プログラム内で取り出して利用している．`meta_buffer_int` は整数用である．

```
#include <metal_stdlib>
using namespace metal;
kernel void uniaccel_func(const device float *in_a[[buffer(0)]],
  const device float *in_b[[buffer(1)]],
  device float *ret[[buffer(2)]],
  const device float *meta_buffer_float[[buffer(3)]],
  const device int *meta_buffer_int[[buffer(4)]],
  uint gid[[thread_position_in_grid]])
{
  const int ret_shape_0 = meta_buffer_int[0];
  const int ret_strides_0 = meta_buffer_int[1];
  const int ret_shape_1 = meta_buffer_int[2];
  const int ret_strides_1 = meta_buffer_int[3];
  const int in_a_shape_0 = meta_buffer_int[4];
  const int in_a_strides_0 = meta_buffer_int[5];
  const int in_a_shape_1 = meta_buffer_int[6];
```

```
  const int in_a_strides_1 = meta_buffer_int[7];
  const int in_b_shape_0 = meta_buffer_int[8];
  const int in_b_strides_0 = meta_buffer_int[9];
  const int in_b_shape_1 = meta_buffer_int[10];
  const int in_b_strides_1 = meta_buffer_int[11];
  int ret_idx_0;
  int ret_idx_1;
  int ret_idx_flat;
  float s;
  int k;
  ret_idx_flat = gid;
  ret_idx_0 = ret_idx_flat / ret_strides_0;
  if (ret_idx_0 >= ret_shape_0)
  {
    return;
  }
  ret_idx_1 = (ret_idx_flat / ret_strides_1) % ret_shape_1;
  s = (0.0);
  k = 0;
  int loop_stop_0 = (in_a_shape_1);
  int loop_step_1 = 1;
  if (loop_step_1 >= 0)
  {
    for (int k = (0); k < loop_stop_0; k += loop_step_1)
    {
      s = s + (in_a[(ret_idx_0)*in_a_strides_0 + (k)*in_a_strides_1]) *
        (in_b[(k)*in_b_strides_0 + (ret_idx_1)*in_b_strides_1]);
    }
  }
  else
  {
    for (int k = (0); k < loop_stop_0; k += loop_step_1)
    {
      s = s + (in_a[(ret_idx_0)*in_a_strides_0 + (k)*in_a_strides_1]) *
        (in_b[(k)*in_b_strides_0 + (ret_idx_1)*in_b_strides_1]);
    }
  }
  ret[gid] = (s);
  return;
}
```

このソースコードを実行するための主要な WebMetal API の利用手順の概略を解説する．なお，実際にはここで挙げていない多数のオプション設定が必要となる．

1. `webmetal = document.createElement('canvas').getContext('webmetal')` により，WebMetal の各機能を呼び出す「コンテキスト」を生成する．

2. `webmetal.createLibrary(source).functionWithName()`, `webmetal.createComputePipelineState()` により，シェーダのソースコード `source` をハードウェア固有のバイナリコードへとコンパイルする．ソースコード中の関数定義で `kernel` という接頭辞があるものは汎用計算用のコンピュートシェーダとして処理される．グラフィック向けには `vertex`, `fragment` という接頭辞が別途存在する．

3. `gl.createBuffer()` により，行列の要素を格納するメモリ領域であるバッファを生成する．1 つのバッファの最大サイズは 256MB であり，行列計算において十分な規模であると言える．メモリの内容は CPU と GPU で共有されるメモリモデルとなっており，JavaScript から直接内容を読み書き可能である．WebGL のテクスチャと異なり，1 要素ずつアクセス可能なためカラーチャンネルについて考慮する必要はない．また，バッファそのものは型を持たず，単精度浮動小数点数，32 ビット整数いずれも格納することができる．

4. `queue = webmetal.createCommandQueue()` により GPU 上で実行する処理を受け付けるキューを作成する. `commandBuffer = queue.createCommandBuffer()` により GPU 上での一連の処理をまとめるコマンドバッファを作成する．
`commandEncoder = commandBuffer.createComputeCommandEncoder()` によりコンピュートシェーダを実行するコマンドを構築するオブジェクトを作成する．

5. `commandEncoder.setComputePipelineState()`, `commandEncoder.setBuffer()` により実行するシェーダ，入出力となるバッファを設定する．バッファは読み書き両用とすることができ，ReLU レイヤーの入出力を同一メモリ上で計算することも可能な仕様である．

6. `commandEncoder.dispatch(threadgroupsPerGrid, threadsPerThreadgroup)`, `commandEncoder.endEncoding()`, `commandBuffer.commit()` により, スレッド数を指定してコマンドを実行する. WebGL と異なり, 1 スレッド (=シェーダ内関数の一回の実行) が行列の 1 要素を計算するという関連付けは固定されていない. GPU は複数の演算器を組として同一のシェーダを同時に実行する. 同時に実行されるスレッドをスレッドグループと呼び, `threadsPerThreadgroup` は 1 つのスレッドグループで同時に実行されるスレッドの数を表す．
`threadgroupsPerGrid` は計算全体で利用するスレッドグループの数である．スレッドグループ内ではスレッド間で高速に共有できる共有メモリが存在し，これを用いることで同一データを複数回利用する行列積等の演算や，バイアス層の勾配計算等で必要な reduction 演算の高速化を図ることができる．WebGL ではこれに相当する機能が存在しないため本研究のシステムでは利用していないが，現代的な GPGPU 向け言語の多くでサポートされており，将来的な活用が期待される．

WebAssembly は通常の C 言語のコードを記述することができる．CPU (JavaScript 側) から，C 言語で実装された個々の関数を呼びだすことが可能である．関数の引数，戻り値はスカラー値であり直接的に行列の内容を与えることはできない．WebAssembly からアクセスできるメモリ領域は JavaScript 側からは 1 つの大きな配列 (ArrayBuffer) としてアクセスすることができる．C 言語上で malloc 関数により行列に対応するバッファを確保し，そのポインタを戻り値として JavaScript に転送すると，これを ArrayBuffer 上のインデックスとして用いることができる．この領域を JavaScript で読み書きすることにより行列の内容およびメタデータを転送することが可能となる．並列実行機能はないため，for 文を用いて各出力要素のインデックスごとに計算を進める．

```
void EMSCRIPTEN_KEEPALIVE SgemmNN_2_2(const float *in_a,
  const float *in_b,
  float *ret)
{
  const int ret_shape_0 = meta_buffer_int[0];
  const int ret_strides_0 = meta_buffer_int[1];
  const int ret_shape_1 = meta_buffer_int[2];
  const int ret_strides_1 = meta_buffer_int[3];
  const int in_a_shape_0 = meta_buffer_int[4];
  const int in_a_strides_0 = meta_buffer_int[5];
  const int in_a_shape_1 = meta_buffer_int[6];
  const int in_a_strides_1 = meta_buffer_int[7];
  const int in_b_shape_0 = meta_buffer_int[8];
  const int in_b_strides_0 = meta_buffer_int[9];
  const int in_b_shape_1 = meta_buffer_int[10];
  const int in_b_strides_1 = meta_buffer_int[11];
  int ret_idx_0;
  int ret_idx_1;
  int ret_idx_flat;
  float s;
  int k;
  ret_idx_flat = 0;
  for (int ret_idx_0 = 0; ret_idx_0 < ret_shape_0; ret_idx_0++)
  {
    for (int ret_idx_1 = 0; ret_idx_1 < ret_shape_1; ret_idx_1++)
    {
      s = (0.0);
      k = 0;
      int loop_stop_0 = (in_a_shape_1);
      int loop_step_1 = 1;
      if (loop_step_1 >= 0)
      {
        for (int k = (0); k < loop_stop_0; k += loop_step_1)
```

```
        {
          s = s + (in_a[(ret_idx_0)*in_a_strides_0 + (k)*in_a_strides_1])
            * (in_b[(k)*in_b_strides_0 + (ret_idx_1)*in_b_strides_1]);
        }
      }
      else
      {
        for (int k = (0); k < loop_stop_0; k += loop_step_1)
        {
          s = s + (in_a[(ret_idx_0)*in_a_strides_0 + (k)*in_a_strides_1])
            * (in_b[(k)*in_b_strides_0 + (ret_idx_1)*in_b_strides_1]);
        }
      }
      ret[ret_idx_flat] = (s);
      goto inner_loop_end;
    inner_loop_end:
      ret_idx_flat++;
    }
  }
}
```

WebAsembly においては，JavaScript から呼び出せる API ではなく，WebAssembly 内で malloc などを用いてバッファの確保や JavaScript 側へのバッファのポインタ転送を行う．レイヤーの計算本体だけでなく，そのためのコードを付加している．

```
static float meta_buffer_float[1024]; // 静的メモリ確保も可能
static int meta_buffer_int[1024];


// テンソル用のバッファ確保
float* EMSCRIPTEN_KEEPALIVE allocBuffer(int size) {
    return (float*)malloc(size * sizeof(float));
}

void EMSCRIPTEN_KEEPALIVE freeBuffer(float* ptr) {
    free(ptr);
}

// meta bufferへのポインタをJavaScript側から取得するための関数
float* EMSCRIPTEN_KEEPALIVE get_meta_buffer_float() {
    return meta_buffer_float;
}

int* EMSCRIPTEN_KEEPALIVE get_meta_buffer_int() {
```

```
    return meta_buffer_int;
}
```

その他，主要なオペレータの実装例を示す．3 次元以上のテンソルに対しても，各バックエンドの制約を意識することなく記述することが可能である．Softmax 関数をはじめとした少数のレイヤーにおいては，1 つの出力要素を計算する際に，全要素の統計量が必要となるような場合がある．提示したコードでは，各出力に対して毎回統計量 (要素の指数の合計) を計算しているため，冗長となってしまう．複数の出力要素で共通する処理を記述できない WebGL の制約から発生したものであり，WebMetal・WebAssembly などでは効率化の余地があるため，次世代 API が十分に普及すれば，それに応じて文法を変更すべきである．

Im2Col の実装.

```
def main(ret: ArrayOut3, img: ArrayIn4, ksize: int,
         stride: int, pad: int):
    # img: batch*inc*inh*inw
    # ret: batch*(outh*outw)*(inc*kh*kw)
    out_w = (img.shape[3] + pad * 2 - ksize) // stride + 1
    out_y = ret.idx[1] // out_w
    out_x = ret.idx[1] % out_w
    in_c = ret.idx[2] // (ksize * ksize)
    ky = ret.idx[2] // ksize % ksize
    kx = ret.idx[2] % ksize
    in_y = out_y * stride - pad + ky
    in_x = out_x * stride - pad + kx
    val = 0.0
    if in_y >= 0 and in_y < img.shape[2] \
        and in_x >= 0 and in_x < img.shape[3]:
        val = img[ret.idx[0], in_c, in_y, in_x]
    return val


def calc_shape(img: ArrayIn4, ksize: int, stride: int, pad: int):
    out_h = (img.shape[2] + pad * 2 - ksize) // stride + 1
    out_w = (img.shape[3] + pad * 2 - ksize) // stride + 1
    return [img.shape[0], out_h * out_w, img.shape[1] * ksize * ksize]
```

ReLU の実装.

```
def main(ret: ArrayOut2, in_a: ArrayIn2):
    s = in_a[ret.idx[0], ret.idx[1]]
    if s < 0.0:
        s = 0.0
    return s
```

```
def calc_shape(in_a: ArrayIn2):
    return in_a.shape
```

Softmax の実装．

$$\frac{\exp(X_c)}{\sum_{j=1}^{K}\exp(X_j)} \equiv \frac{\exp(X_c - m)}{\sum_{j=1}^{K}\exp(X_j - m)} \tag{4.2}$$

という事実を利用し，指数関数をオーバーフローさせないための一般的なトリックを実装している．

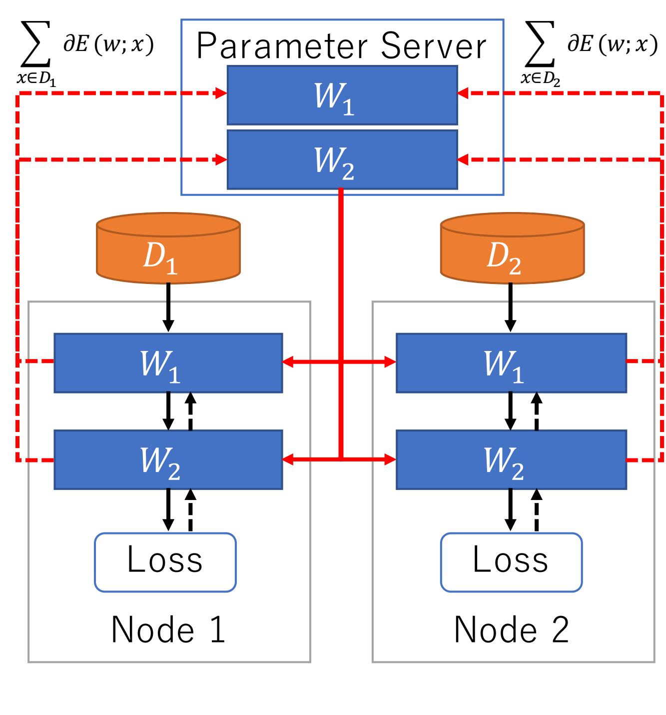
```
def main(ret: ArrayOut2, logit: ArrayIn2):
  row = ret.idx[0]
  max_val = -10000.0
  for col in range(logit.shape[1]):
    p = logit[row, col]
    if p > max_val:
      max_val = p
  exp_sum = 0.0
  for col2 in range(logit.shape[1]):
    p = logit[row, col2]
    exp_sum = exp_sum + math.exp(p - max_val)
  v = math.exp(logit[row, ret.idx[1]] - max_val) / exp_sum
  return v


def calc_shape(logit: ArrayIn2):
  return logit.shape
```

4.3.2 分散計算アルゴリズムの選択

深層学習において，DNN モデルの最適化手法として最も有力なのはミニバッチ確率的勾配降下法である．計算機 1 台で計算を行う場合と結果が等価となる，同期更新方式を用いる．

同期更新方式における分散計算を行う場合，計算の分割方法について大きく分けて 2 つの方針がある．データ並列とモデル並列である．図 4.1 に各方式の模式図を示す．データ並列は，ミニバッチを分割し，各計算ノードに割り当てる．各計算ノードでは割り当てられたミニバッチに対する勾配を計算し，パラメータサーバへ送信する．パラメータサーバはすべての勾配の平均を計算し，これをもとに SGD によるモデル更新を行う．更新されたモデルを全計算ノードに配布し，次のイテレーションを開始する．全計算ノードが 1 台のパラメータサーバと通信する必要があるため，通信がボトルネックとなる場合がある．これに対処するため，勾配を圧縮する手法 [51] や，ネットワーク的に近い計算ノード同士で通信して階層的に勾配の集計を行う手法 [52] が提案されている．モデル並列は，モデルを構成する各層の行列を分割し，各部分行列を計算ノードに割り当て，計算ノード内で更新を行う．この方針では，通信されるのは重



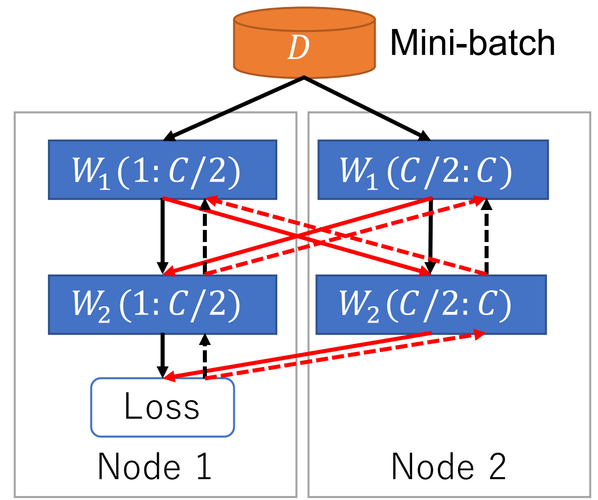


図 4.1 データ並列 (上) とモデル並列 (下)．赤い矢印は通信が必要な箇所を示す．実線は順伝播，点線は逆伝播を表す．$W_1(1:C/2)$ は，行列 W_1 の上半分，すなわち出力チャンネル C の半分に対応する．

み行列ではなく途中の層の出力データである．ある 1 つの全結合層に注目し，計算量および通信量を見積もる．計算ノード数を G，全計算ノードの合計バッチサイズ N，入力チャンネル数 C_{in}，出力チャンネル数 C_{out} とおく．データ並列について，1 台あたりの計算量は $C_{in}C_{out}N/G$ に比例し，(通信路が共有されていると考え) 全計算ノードの転送量合計は，各計算ノードが

$C_{in}C_{out}$ のパラメータ行列をパラメータサーバと通信するため $C_{in}C_{out}G$ となる．モデル並列について，計算量は $C_{in}C_{out}N/G$ に比例し，転送量は各計算ノードの出力 $C_{out}N/G$ を他の全計算ノードに転送するため，$C_{out}NG(G-1)/G \sim C_{out}NG$ となる．いずれの方式でも 1 計算ノードあたりの計算量を一定にするためには N/G を定数とする必要がある．この場合，転送量はデータ並列の場合 G に比例，一方モデル並列の場合は G^2 に比例するため，モデル並列は台数増加に従って通信量が大幅に増加してしまい大規模化が難しいと考えられる．同一データを全計算ノードに一斉配布するブロードキャストが利用できるか否かなどネットワークの条件により実際の効率は大きく変動するが，最近の大規模な分散計算においては，データ並列が用いられることが多い [15][53]．いずれの方式においても，全計算ノードは同一のモデルを共有するため，各イテレーションの計算結果を同期する必要がある．なお，学習データはモデルサイズと比べかなり小さい場合がほとんどであるうえ，実装を工夫すればモデルの同期と関係なく事前に配布しておくことも可能なため上記のコスト計算に含めていない．

一方非同期更新方式は，同期更新方式におけるデータ並列と類似しているが，ある計算ノードが勾配をパラメータサーバに転送した時点でパラメータサーバ上のモデルを更新する．その計算ノードは更新後のモデルを受信し，次のミニバッチに対する勾配を計算する．計算中の他の計算ノードは古いモデルに対する勾配を計算し続け，計算が終わった時点で転送する．この勾配もパラメータサーバ上の (すでに更新された) モデルの更新に用いられる．各計算ノードは同期を待たず計算を行うことができ効率が良いように見えるが，古いモデルに対する勾配を用いて新しいモデルを更新することはモデルの収束性を低下させる [49] ことが知られており，計算量のわりに効率が悪く，現在のところアルゴリズムとして有力ではない．

本研究においては，同期更新方式においてデータ並列を採用する．計算に用いる端末が計算中に増加したり，予測できないタイミングで切断されたりする状況を仮定する．モデル並列におけるメリットはモデル行列を計算ノード内で更新し，通信する必要がないという点であるが，端末が切断する事態に対応するためにはモデル行列を信頼できる場所にバックアップする必要がありこのメリットが得られない．また，端末の計算性能を動的に推定しながら計算量の配分を更新していく場合，担当する端末が変更される箇所の通信が必要であり，端末に性能差があり事前にその計算能力がわかっていない条件には適さない．学習データ・モデルパラメータ・勾配の通信は端末とサーバ間でのみ行う．Web ブラウザにおいて端末間の P2P 通信を行える API として WebRTC [2] が存在するが，階層的な勾配の集約による効率化を実装することは困難である．データセンターと異なりネットワークのトポロジーが不明であることや，特に同じ無線 LAN アクセスポイントにすべての端末が接続している場合，ある瞬間にアクセスポイントと通信できるのは原則として 1 台の端末のみであるため複数のペアが同時に集約を行うことができないということが挙げられる．そのため勾配の集約・配信は単一のサーバが行うものとする．本研究では Web ブラウザ特有の課題に注力するが，勾配の圧縮による通信の高速化と組み合わせることは可能であると考えられる．

[2]https://webrtc.org/

4.3.3 計算性能差を考慮した分散計算システム

本研究では，深層学習のモデル最適化を表現する計算グラフを入力とし，計算結果を変えない範囲で Web ブラウザを計算ノードとして分散計算させるシステムを提案する．これは，深層学習基盤における通信部分の学習時の高速化に対応する．まず，分散計算における負荷分散に関するアルゴリズムを提案し，次に Web ブラウザを計算ノードとしてそれを実現するソフトウェア実装について説明する．

ConvNetJS [41] は独自の DNN モデル定義構文を用い，Web ブラウザ内で学習・識別を完結させていた．しかしながら現実的な応用においてはインターネット上で配布されている学習済みモデルからのファインチューニングを行う等のタスクを考慮すると，他の深層学習フレームワークとの相互運用性が必要である．そのため本研究では DNN モデルの共通フォーマットである ONNX 形式 [3] で記述されたモデル構造及びパラメータの初期値を入力とし，これを学習することとした．ONNX モデルを解釈し，計算の列として展開する機構を設けた．DNN の勾配計算には誤差逆伝播法が用いられるが，順方向計算・逆方向計算の両方をあらかじめ明示的に展開し，指定された順序通り計算することで勾配を計算できる手順に変換した上で Web ブラウザへ転送することとした．変数に対応するバッファの確保は，各イテレーションの実際のバッチサイズに対応して動的に行う．

計算性能差を考慮した負荷配分

本研究では計算ノード間に性能差が存在することを想定し，適切な負荷配分を行うことにより学習時間の最小化を狙う．計算タスクの変更や計算ノードの参加・離脱が高頻度に行われると仮定し，詳細なベンチマークを行わず，計算を行わせながら動的に性能を推定し，配分を決定する．

ヘテロジニアスなサーバクラスタに対する先行研究 [46][47] では，全計算ノードの合計バッチサイズを固定し，それを分配する方式を取っていた．しかしながら計算ノードに動的に参加・離脱を許す場合，計算ノード数が少ない場合は 1 計算ノード当たりのバッチサイズが大きくなりすぎてメモリが不足する．一方，計算時間にはバッチサイズによらない GPU 制御・通信に対応する定数項があるため，計算ノード数が増加した際，1 計算ノードあたりのバッチサイズを低下させると各計算ノードにおける計算効率が悪化してしまう．代わりに，台数に応じて合計バッチサイズを増加させ，各計算ノードにおける計算効率を維持する手法を提案する．ここで問題となるのは，合計バッチサイズが変化すると，同じ学習率であってもモデルの収束に変化が生じてしまうことである．積極的にバッチサイズを制御し有効活用する例も存在するが [54]，ランダムにバッチサイズが変化する場合は問題となる．Goyal らは，最大 256 台の GPU を用いて ResNet-50 モデルの学習を 1 時間で完了させた [15]．その際，バッチサイズと学習率を比例させる「linear scaling rule」を提案している．学習中のバッチサイズは原則一定という前提であるが，本研究ではイテレーションごとにバッチサイズが毎回変化することを想定する．そのため，予備実験としてイテレーションごとにランダムにバッチサイズを変化させ，学習率をこれに比例して毎回変化させて学習することを試みた．

[3]https://onnx.ai/

表 4.1 CIFAR-10 を識別する DNN のモデル構造．深層学習ライブラリ Keras のサンプルを参考に作成．

層の種類	出力チャンネル数	カーネルサイズ	ストライド	パディング
Convolution	32	3	1	1
Convolution	32	3	1	0
Max Pooling	32	2	2	0
Dropout (25%)				
Convolution	64	3	1	1
Convolution	64	3	1	0
Max Pooling	64	2	2	0
Dropout (25%)				
Fully-connected	512			
Dropout (50%)				
Fully-connected	10			

CIFAR-10 データセット [55] を識別する画像分類 CNN を学習させる．CIFAR-10 データセットは，32×32px，10 クラス，50,000 枚のカラー画像からなる画像分類ベンチマークデータである．DNN モデルを表 4.1 に示す．畳み込み層 4 層，全結合層 2 層からなる．初期条件をそろえるため，ランダムに初期化されたモデルを 1 エポックの間バッチサイズ 32 で学習させ，次にバッチサイズを $\{32, 64, 128, 256, 512\}$ それぞれで固定した場合および，32 から 512 の間の一様乱数を用いて各イテレーションにおいてランダムに設定した場合で学習を行った．学習率はバッチサイズ 32 に対して 0.001 とし，バッチサイズに比例して変化させた．最適化アルゴリズムは Momentum SGD である．学習の経過を図 4.2 に示す．各条件において 10 回測定を行なった平均を表示している．横軸はエポックであり，処理したサンプル数に比例する．この結果より，イテレーションごとにバッチサイズが変化する場合でも，学習率を制御することにより学習の進行はバッチサイズが一定の場合と大きな違いがないことが確認された．そのため，バッチサイズは固定せず，単位時間当たりに勾配計算を行うサンプル数をスループットとし，この値を最大化することで学習時間を最小化するという方針をとる．

サーバクラスタにおける先行研究 [46] では，各計算ノード i におけるバッチの処理時間が，バッチサイズ β_i の一次関数 $A_i\beta_i + b_i$ で表現できると仮定した．Web ブラウザにおいても同様であることを確認する予備実験を行った．バッチサイズを 4 から 64 までランダムに変化させ，50 イテレーション実行し，各イテレーションの所要時間を測定した．所要時間は，サーバがイテレーション開始メッセージを送信してから，デバイスから勾配送信完了メッセージを受信するまでの時間をサーバ側で測定した．初回については初期化処理により通常より長い時間がかかるため除いている．複数のデバイス・バックエンドにおける測定結果を図 4.3 に示す．

実験結果より，バッチの処理時間はバッチサイズの一次関数でおおむね近似可能であることが示された．同時に，低い確率で通常よりかなり長くかかる場合がある場合も見られた．これは，端末上で突発的に高負荷の別プロセスが動作したり，通信経路が他のコンピュータによる通信で輻輳することなどによって発生すると考えられる．[47] では OS 上の API により CPU 負荷・メモリ使用量を取得し一部の要因については予測を可能としていたが，Web ブラウザ

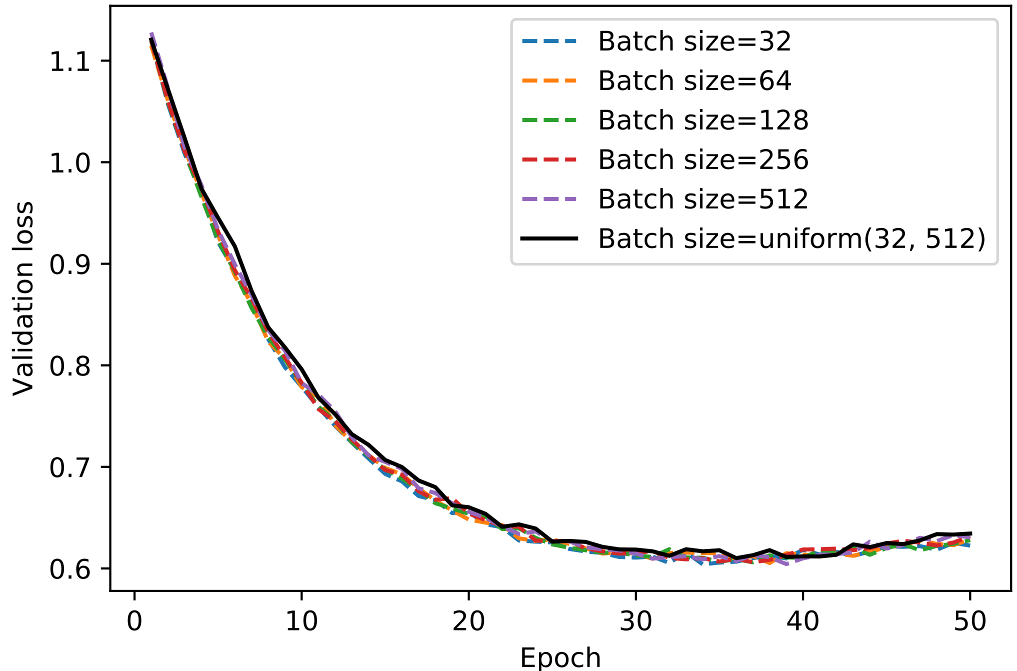


図 4.2 バッチサイズに比例して学習率を変化させた場合の validation loss の推移．uniform(32, 512) は各イテレーションにおいてバッチサイズを 32 以上 512 以下の一様分布からサンプリングした場合を示す．

上ではこれらの情報を得ることができない．そのため本研究においては，一定時間内に勾配が受信されなかった場合はその計算ノードを無視しモデルの更新を行うタイムアウト機構を設けた．タイムアウト時間は予測される処理時間の 2 倍とした．この機構により，不意にデバイスが切断された場合にも学習が停止しないという機能も同時に実現することが可能である．デバイスがタイムアウトした場合，バッチサイズが予定より小さくなるがそれにあわせて学習率を調整することにより影響を回避できる．また学習データはランダムシャッフルされているため，どのデバイスがタイムアウトしたかにかかわらず学習結果に統計的な偏りは生じない．

サンプル数配分 $\beta_1, \beta_2, \ldots$ を用いてスループット P は以下の式で求められる．

$$P = \frac{\sum \beta_i}{\max(A_i \beta_i + b_i)} s.t. 0 < \beta_i \leq \beta_{max} \tag{4.3}$$

ここで，β_{max} は 1 デバイスあたりの最大バッチサイズであり，メモリ容量の制約から定まる．β_{max} は本来ハードウェアごとに異なるパラメータとなり，GPU の空きメモリを取得する API が利用できればその情報を用いて設定することが妥当である．しかしながら，現存するいずれの API もこの機能を提供していない．また，メモリ確保エラーが生じるまでメモリ確保量を増加させることによりメモリサイズを取得する手段も問題がある．CPU と GPU が同一のメモリを共有している環境においては，メモリ確保が進むにつれ CPU 側で利用するメモリが

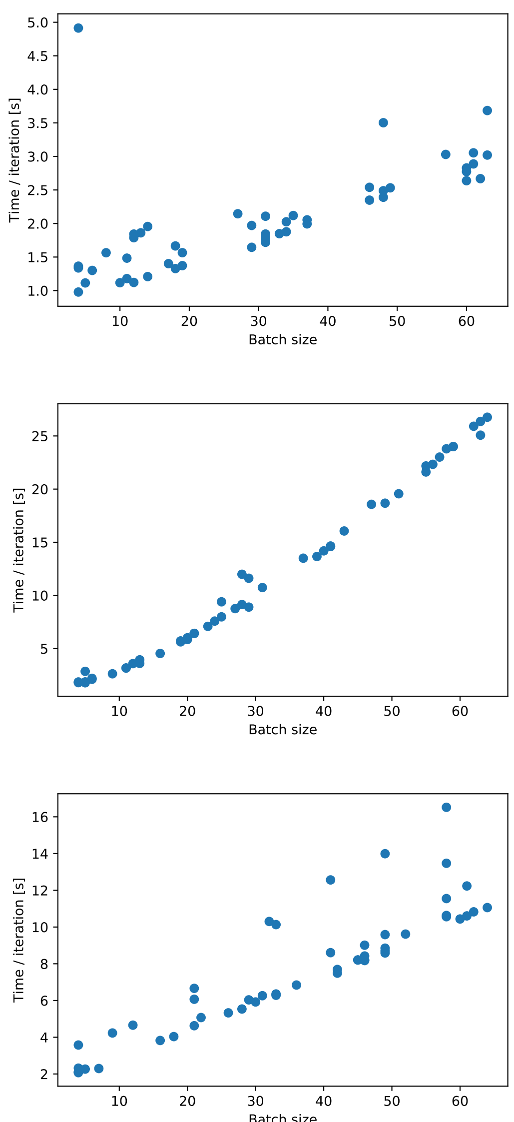


図 4.3 各デバイス・バックエンドにおける，バッチサイズに対する処理時間．上から，Mac1 (WebGL)，Mac1 (WebAssembly)，MediaPad (WebGL)．

ディスクにスワップアウトされ，明示的なエラーではなくデバイスが長時間にわたり応答しなくなるという挙動が生じた．デバイスの性能を最大限活用するためにはメモリ容量を取得する API が必要である．本研究における実験では，ヒューリスティックに β_{max} を定数 1GB とした．

スループットを最大化する解は，

$$\beta_i = \begin{cases} \beta_{max} & (i = \arg\min_i (A_i \beta_{max} + b_i)) \\ \frac{\min_j (A_j \beta_{max} + b_j) - b_i}{A_i} & (otherwise) \end{cases} \tag{4.4}$$

と求められる．

計算ノードの計算速度を表すパラメータ A_i, b_i は，計算時間の実測値をもとに線形回帰により行う．

学習データを用いて各計算ノードで計算された勾配を集約してモデルを更新する．計算ノード i に割り当てられた学習サンプルの集合を D_i とする．$|D_i| = \beta_i$ である．全計算ノードの学習サンプルの集合を $D = D_1 \bigcup D_2 \bigcup ...$，学習率をバッチサイズに比例させるため，1 サンプルあたりの学習率を $\hat{\lambda}$ とすると，SGD の更新則は式 (2.13) を変形し

$$w_{t+1} = w_t - \hat{\lambda}|D|\frac{1}{|D|}(\sum_{x \in D} \partial E(w_t; x)) \tag{4.5}$$

となる．さらに変形すると

$$w_{t+1} = w_t - \hat{\lambda}\sum_i(\sum_{x \in D_i} \partial E(w_t; x)) \tag{4.6}$$

となる．すなわち，各計算ノードにおいて学習サンプルに対する勾配の合計を計算し，サーバ側でそれらの合計を計算することで更新を行うことができる．一般的な深層学習フレームワークではバッチサイズで除算した $\frac{1}{|D_i|}\sum_{x \in D_i} \partial E(w_t; x)$ が勾配として得られる場合が多いが，この値の計算ノード間の単純平均を用いるとバッチサイズの違いにより偏った平均が得られてしまうため注意が必要である．

計算ノードの制御ソフトウェア

ここまでに提案した，データ並列分散学習を Web ブラウザに行わせるソフトウェアシステムについて説明する．

構成および 1 イテレーションの動作の流れを図 4.4 に示す．マスタープログラム (Master Program) は学習の進行を制御する主体である．制御サーバ (Control Server) は計算ノードからの接続を受け付け，制御メッセージをやりとりする．イテレーションの開始メッセージをサーバ側からプッシュ配信する必要があるため，これを低いレイテンシで行える WebSocket プロトコルで計算ノードと通信する．パラメータサーバ (Parameter Server) は，マスタープログラムと各デバイス間での行列の転送を仲介する．クライアント側のリクエストに応じて容量の大きな行列ファイルを送受信する．ライブラリのパフォーマンス上の理由により通常の HTTP を用いている．

マスタープログラムがイテレーションを開始する．負荷最適化器 (Load optimizer) が接続済の計算ノードの動作履歴から，各計算ノードのバッチサイズ β_i を最適化する (1)．また，各計算ノードごとの学習サンプル D_i および現在のモデルパラメータ w をパラメータサーバに転送する (2)．制御サーバを介して各クライアントにイテレーションの開始および計算すべきバッチサイズを通知する (3)．計算ノードはパラメータサーバから D_i, w を取得 (4) し，勾配計算を行う．計算が完了したら勾配をパラメータサーバに転送 (5) する．同時に，制御サーバを介し

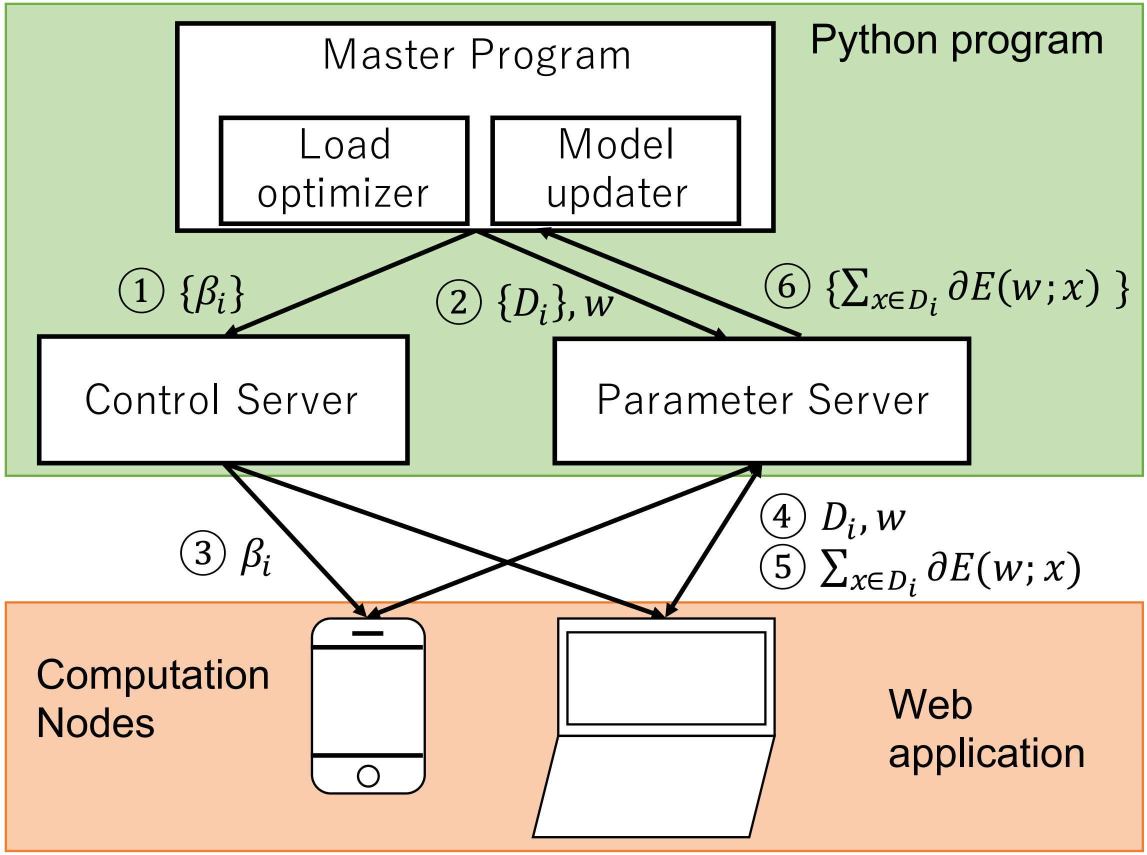


図 4.4 分散計算システムの構成

てマスタープログラムに完了を通知する．マスタープログラムは，全計算ノードが勾配計算を終えるかタイムアウトに達した時点で勾配を取得 (6) し，モデルの更新を行う．

デバイスを計算ノードとして用いるには，Web ブラウザで本システムの URL を開くだけであり，セットアップは不要である．制御サーバに自動的に接続を確立し，計算グラフを受け取ったのち，イテレーションの開始メッセージを待機する．行列演算部分は第 4.3.1 節で提案したシステムを用いて，GPU 上で計算する．

4.4　実験

4.4.1　コード変換ベンチマーク

提案システムにより，Python コードから自動生成された，Web ブラウザ上で動作する行列計算コードについてベンチマークを行う．

実験環境は，MacBook Pro (Retina, 13-inch, Late 2013)，Intel Core i5 CPU，Intel Iris GPU，macOS 10.14.5 である．WebMetal は Web ブラウザ Safari 上で，WebGL は Firefox 上で動作させた．

表 4.2 512 × 512 の行列同士の行列積の処理時間．CPU と GPU 間のデータ転送時間を含む．

GPU API / ライブラリ	実行時間 [ms]
WebMetal (ours)	**20.4**
WebGL2 (ours)	39.0
WebGL1 (ours)	88.3
WebGL2 (weblas [43])	27.3
CPU (Sylvester [56])	2597.9

512 × 512 の行列同士の行列積を求める処理時間を計測する．GPU API として WebMetal と WebGL を比較する．WebGL は 2 つバージョンがあり，WebGL1，WebGL2 それぞれで実行する．実験において関係する違いは，WebGL1 ではテクスチャの色情報が 1 ピクセルごとに常に RGBA の 4 チャンネルからなる一方，WebGL2 では 1 チャンネルのみのテクスチャを作成できる点である．現在のシェーダ生成システムでは 1 ピクセルあたり 1 チャンネルしか使用できないため，WebGL1 では不要なチャンネルが存在するためメモリアクセスの効率が低下する．WebGL1 を用いて行列積を求める既存ライブラリ weblas [43] とも比較を行う．このライブラリは 4 チャンネルすべてを効率的に用いるようシェーダが手動でチューニングされている．また，JavaScript のみで実装されたライブラリ Sylvester [56] とも比較した．

実験結果を表 4.2 に示す．シェーダコードが単純な実装であるため，WebGL 環境ではチューニングされている weblas に劣る結果となった．一方，WebMetal を利用することにより weblas よりも高速な計算ができるようになっており，新しい API への対応が有効であることが示された．

4.4.2 分散計算ベンチマーク

開発したソフトウェアを用いて，様々な端末・条件において DNN の学習を行いその計算速度を評価した．先述のように，バッチサイズの変化は学習率の変更によって吸収可能であるため，スループット (単位時間当たりに処理できたサンプル数) を学習速度の指標として用いる．

残念ながら，最もパフォーマンスの高い WebMetal については，高負荷な計算を連続して行うと応答しなくなる場合があり，長時間にわたる計算が必要な学習に利用することが不可能であった．実験的な API であるため内部実装に問題がある可能性があるが，利用例が極めて少ないため解決につながる情報が得られなかった．そのため，WebGL および WebAssembly を用いて実験を行った．WebMetal または次世代の API が利用可能となれば，より大きなモデルの学習が可能となることが期待できる．

1 端末あたりの最大バッチサイズ β_{max} については，GPU のメモリサイズを取得する API がないため，必要な GPU メモリ量が 1GB 以下となるようすべての端末で一律に 64 に定めた．

実験に用いるデバイスを表 4.3 に示す．デバイス 1 台のみで，バッチサイズ $\beta_{max} = 64$，100 イテレーションの間学習を行なった場合の平均スループットも示した．なお，タブレット・スマートフォンデバイスについては図 4.5 に示すように，時間とともにスループットが変動するものが存在した．発熱によるクロックダウン等により速度が非定常である場合を考慮し，計算ノードの計算速度を表すパラメータ A_i, b_i の推定は各イテレーションの開始時に行い，過去の

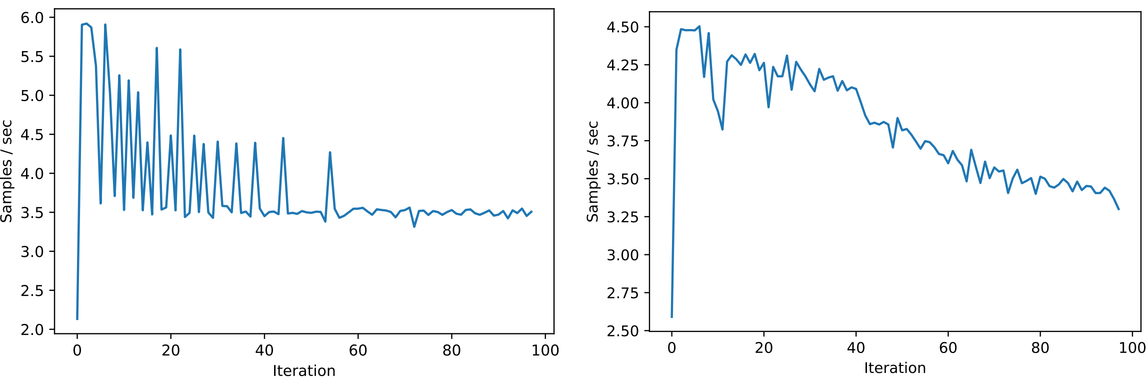


図 4.5 固定バッチサイズにおけるスループットの推移．左: MediaPad，右: iPhone 8.

表 4.3 分散計算に用いるデバイス一覧．スループット (Samples / sec) の G はバックエンドとして WebGL を使用した場合，A は WebAssembly を使用した場合．

記号	機種	端末の種類	OS	スループット
Mac1	MacBook Pro (13-inch, 2017, Four Thunderbolt 3 Ports)	ノート PC	MacOS	G: 25.5 A: 3.36
Mac2	MacBook Pro (13-inch, 2017, Two Thunderbolt 3 Ports)	ノート PC	MacOS	G: 22.4 A: 3.53
Mac3	MacBook Pro (Retina, 13-inch, Late 2013)	ノート PC	MacOS	G: 21.2 A: 3.03
MediaPad	HUAWEI MediaPad M3	タブレット	Android	G: 3.69
ZenPad	ASUS ZenPad 3 8.0	タブレット	Android	G: 6.28
iPhone8	iPhone 8	スマートフォン	iOS	A: 3.78
iPhone6	iPhone 6	スマートフォン	iOS	A: 1.30

計算時間の測定値に対して指数関数的に重み付けを行った．計算中の発熱を防止することは難しいため，各測定の間に時間を設けて独立性を担保した．すべてのデバイスは単一の無線 LAN アクセスポイントに接続しており，1 イテレーション・1 デバイスごとにパラメータの送信・受信それぞれ約 4.8MB が必要となる．

複数のバックエンドに対応しているデバイスについては，各バックエンドごとに測定を行なった．Mac OS・Android デバイスでは Chrome ブラウザ，iOS デバイスでは Safari ブラウザを用いた．WebGL・WebAssembly 両方が利用できるデバイスにおいても，デスクトップ向け OS (Windows・MacOS) では WebAssembly を利用することが有効な場合がある．デバイスでユーザが他の作業を行いながら余剰計算資源を分散計算に用いるという使い方が可能であるが，WebGL を計算に用いた場合は計算中に画面描画が停止してしまい作業が困難であった．一方 WebAssembly ではそのような問題は生じなかった．CPU においては計算途中であっても OS により他のタスクに強制的に切り替えられる (プリエンプション) が，GPU 上ではそれが行われないためであると考えられる．モバイル OS (Android・iOS) ではバックグラウンドのタスクは停止されてしまうため，このような使い方はできない．

表 4.4 分散計算に用いるデバイスの組み合わせ．#以降の数字は同一機種の区別のため．(G) は WebGL 利用，(A) は WebAssembly 利用．合計スループットは表 4.3 におけるスループットの単純合計．

グループ	デバイスリスト	合計スループット
Group1	Mac1(G),Mac2#1(G),Mac2#2(G),Mac2#3(G), Mac2#4(G),Mac3#1(G),Mac3#2(G)	157.5
Group2	MediaPad(G),ZenPad(G)	9.97
Group3	MediaPad(G),ZenPad(G),iPhone6#1(A), iPhone6#2(A),iPhone8(A)	16.35
Group4	Mac1(A),Mac2#1(A),Mac2#2(A),Mac2#3(A), Mac2#4(A),Mac3#1(A),Mac3#2(A)	23.54
Group5	Mac1(A),Mac2#1(A),Mac2#2(A),Mac2#3(A), Mac2#4(A),Mac3#1(A),Mac3#2(A) MediaPad(G),ZenPad(G),iPhone6#1(A), iPhone6#2(A),iPhone8(A)	39.89

表 4.5 分散計算によるスループット測定結果

グループ	Constant	FixedSum	ours
Group1	**83.8**	18.8	77.1
Group2	7.48	7.86	**8.96**
Group3	5.57	10.1	**13.7**
Group4	19.8	15.5	**21.4**
Group5	13.2	10.1	**31.2**

各測定で用いるデバイスの組み合わせをグループと呼び，表 4.4 に示す．ノート PC 同士，タブレット同士のほか，複数種類のデバイスを混合した環境を想定している．

分散計算を行いスループットを測定した結果を表 4.5 に示す．比較手法として，全デバイスのバッチサイズを β_{max} に固定した場合 (Constant)，Yang ら [46] による，全デバイスのバッチサイズ合計を β_{max} に固定した場合 (FixedSum) を用いた．Group2，3 においては提案手法のスループットが他の手法を上回り，またいずれのデバイス単独よりも高いスループットとなり，提案手法は有効であった．Group3 は Group2 に対して純粋にデバイスを追加しているが，Constant 条件においてはむしろスループットが低下している．これは，Group3 で追加された iPhone6 の性能が低く，性能が高い ZenPad 等のアイドル時間が長くなってしまうためである．提案手法では性能が低いデバイスへの割り当てバッチサイズが小さくなるため，他のデバイスを待たせる問題が生じない．なお，Mac1 を WebGL で動作させ，iPhone6 を組み合わせた場合は性能差が大きすぎ，iPhone6 に割り当てるバッチサイズがほぼ 0 となってしまった．極端に性能差がある場合には，非同期更新手法を検討すべきである．一方，Group1 については全てのデバイスのバッチサイズを一定にした方がスループットが高くなっている．この原因として，各デバイスの性能が高く計算時間に対して通信時間の比率が上昇していたことと，Constant 条件では各デバイスの計算終了時間に若干の差が生じ，単一の無線 LAN アクセスポイントを経由したパラメータサーバとの通信における干渉が少ない状態になったためと考えられる．そこ

で理想的な環境として，学習データ・パラメータの通信を行わず，ダミーデータを入力として勾配計算のみを行った．その結果，Constant 条件でのスループットは 178，FixedSum 条件では 150，提案手法では 201 となった．すなわちネットワークの高速化・圧縮技術の進歩によって通信時間の短縮・デバイス間の干渉が減少すれば，デバイスの性能差がわずかであってもこれを考慮した提案手法が有効となると考えられる．

図 4.6 に Group3 の提案手法における各デバイスのバッチサイズの推移を示す．MediaPad・iPhone8 は発熱による性能低下が生じ，これに応じてバッチサイズの割り当てが低下していることがわかる．提案手法では，デバイス性能の変化に対しても対応が可能である．図 4.7 に，計算途中でデバイスを接続・切断した場合の動作結果を示す．デバイスが接続されるとその性能が推定され，バッチサイズの配分が最適化される．また，デバイスが切断された瞬間はスループットが低下するが，タイムアウトが設定されているため学習が停止することなくロバストに続行可能である．

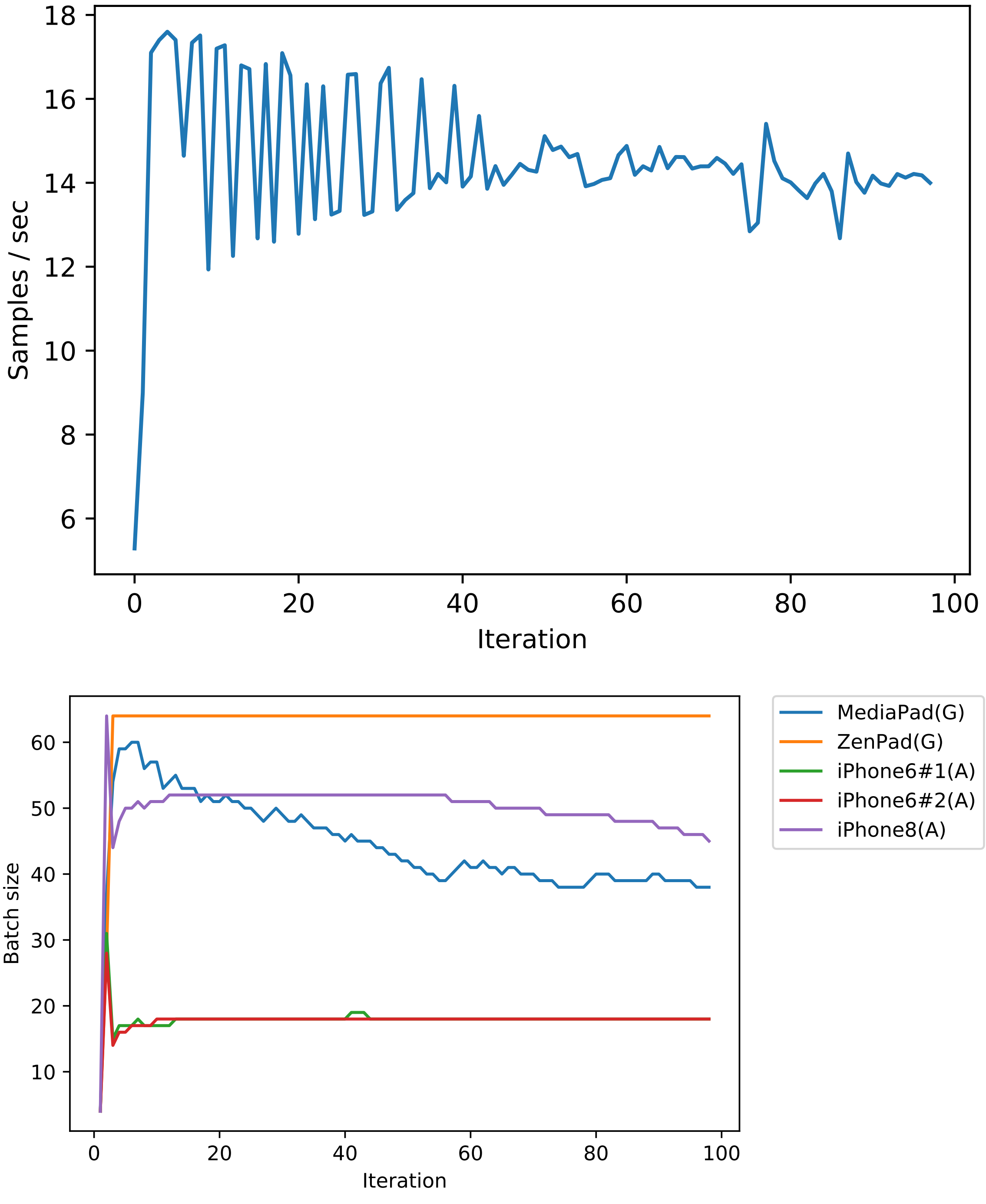


図 4.6 Group3 におけるスループット (上)・各デバイスのバッチサイズの推移 (下)

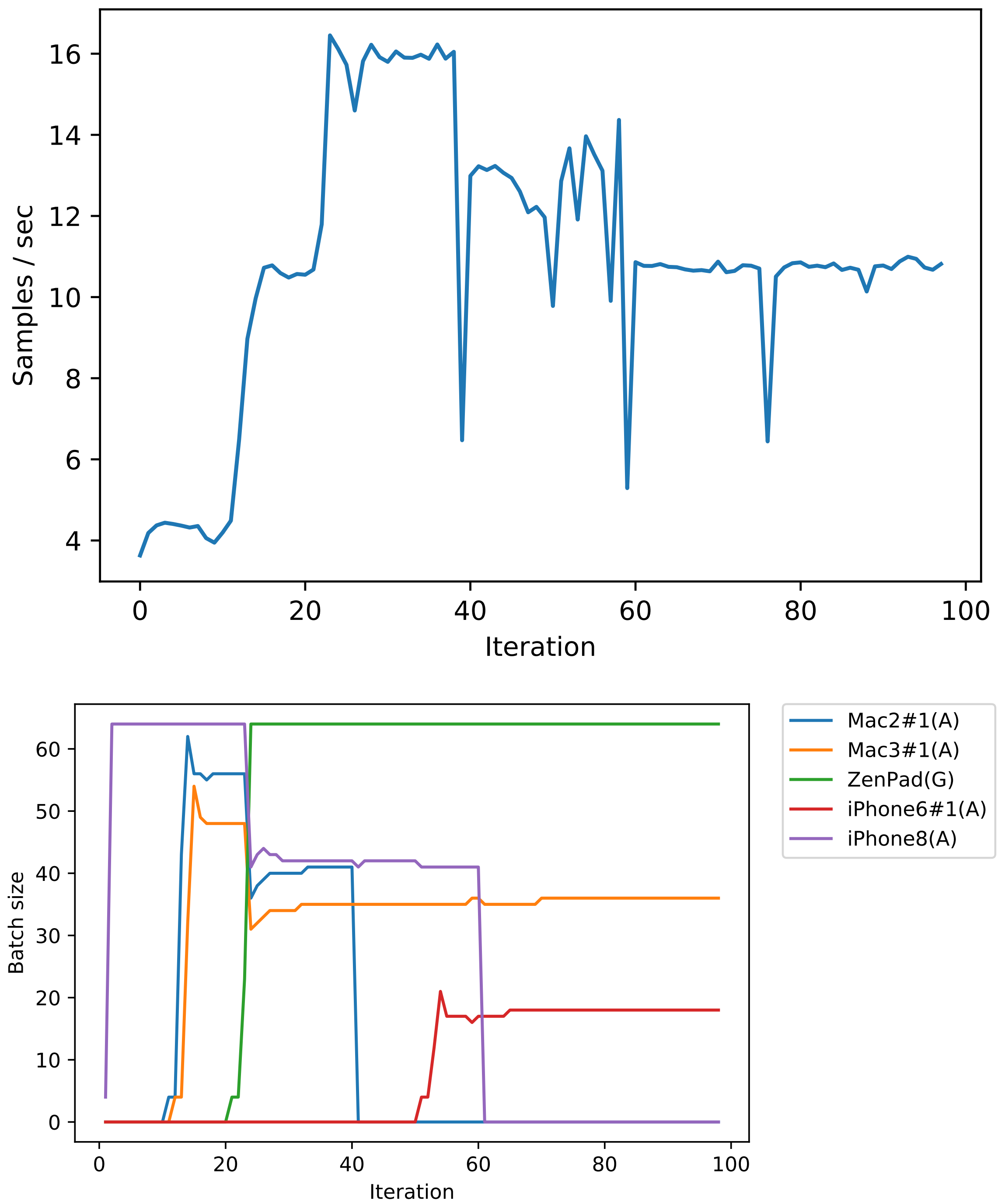


図 4.7 動的なデバイス接続・切断におけるスループット (上)・各デバイスのバッチサイズの推移 (下)．デバイスが接続されていない期間はバッチサイズ 0 として表示している．

Chapter 5

Web ブラウザのための Deep Neural Network の圧縮

5.1 序論

端末のアクセラレータの活用により，モデルのパラメータ量が 100MB 以上となるような最先端の画像分類モデルを Web ブラウザ上で実行することが可能となる．端末にインストールするネイティブアプリケーションではバックグラウンドでアプリケーションのインストール・更新が行われる一方，Web アプリケーションではユーザがアプリケーションを開いているときに必要なデータのダウンロードを行う必要があり，大きなモデルを利用するアプリケーションではモバイル回線でのダウンロードに時間がかかるという課題がある．例えば AlexNet [8] は 233MB，モバイルデバイス向けにコンパクトに設計された MobileNetV2 [57] でも 13.5MB ある．モバイル回線の平均的なダウンロード速度は 3MB/sec [58] であり，これらのモデルを瞬時に読み込める環境は少ない．携帯電話回線の進歩は著しいが，モデルを配信するサーバ側設備においても，100MB のデータをユーザに提供するには 1 円程度かかり [1]，様々な地域に居住するユーザに十分な帯域を確保することは容易ではない．

DNN をモバイルデバイスで活用するため，モデルを圧縮する研究が広く行われている．既存の手法は，圧縮率と精度のトレードオフをハイパーパラメータとして設定し，それに従って圧縮を行う．ユーザにとって，モデルの精度は高いほうがいいが，モデルのロードが長すぎることは問題となる．たとえば Akamai によれば"A 100-millisecond delay in website load time can hurt conversion rates by 7 percent"[59] と言われている．Augmented Reality の利用により衣服の試着 [2]，家具の設置 [3] などの広告目的の利用も検討され始めており，初回アクセスを行うユーザを待たせない工夫は重要である．また，野外で動植物などを認識し解説を提示するようなアプリケーションにおいても，事前にアプリケーションをインストールするのではなく，気にな

[1] Amazon Web Services における通信料より試算

[2] https://www.moguravr.com/adobe-ar-dressing-dev/

[3] https://www.moguravr.com/ikea-place-ar/

る動植物を発見した際に回線速度が十分でない環境においてアプリケーションをダウンロードしようとするケースが考えられる．

単純な解決策として，低速回線ユーザには小さなモデル，高速回線ユーザには大きなモデルをダウンロードさせる方法が考えられる．しかし，回線速度は一般的に事前に予測できないため，実現が難しい．

写真を圧縮する形式として，Progressive JPEG という形式が存在する．通常の JPEG 形式は，画像の上部から下部の順にファイルに格納され，ダウンロード途中では画像の上部から順に表示されていくため，全体像を把握することが難しい．Progressive JPEG では，画像全体の低周波成分をファイルの先頭に格納し，次に高周波成分を格納する．ダウンロード途中で低解像度ながら画像全体を表示することができ，全体像の把握を早い段階で行うことが可能となる．DNN においても，通常の形式であれば層ごとにパラメータが格納される．すべての層のパラメータがそろわなければ出力を計算することが不可能なため，ユーザの待機時間が長い．Progressive JPEG のようにすべての層の情報を粗く転送したうえで詳細な情報を転送することで，ダウンロード途中でも DNN の実行を可能とすることを目指す．

本章の目的は，回線速度が予測できないモバイル回線における DNN の読み込みに適した圧縮手法を提案することである．深層学習基盤における通信部分の推論時の高速化に対応する．本章では，学習済みモデルを，精度は低いがコンパクトな「ベースモデル」と，圧縮の過程で失われた情報を補完する複数の「パッチ」を生成する．この手法の概要を図 5.1 に示す．アプリケーションは最初にベースモデルのみをロードし，完全なモデルと比べて低い精度で動作を開始する．次にパッチを読み込み，逐次的にモデルを更新し精度を回復していく．この手順により，回線速度に依存する速度とサイズのトレードオフパラメータが不要となる．この手続きを Progressively Recoverable Compression (PR Compression) と名付ける．目的を達成するため，次の手法を提案する．(1) 情報量のオーバーヘッドなくベースモデルとパッチを分離するための反復的な量子化ビット削減とチャンネル単位のプルーニング，(2) 精度低下を補うためのバッチ正規化レイヤーのファインチューニング，(3) 理想的なエントロピーに近いパッチファイルを生成するため最適な符号化方式の選択．

本章の貢献は以下の通りである．

- 情報を削減する順序の最適化により，精度を極力維持できることを示した．
- バッチ正規化レイヤーのファインチューニングにより，少しのサイズオーバーヘッドで低下した精度を回復させられることを示した．
- 削減された情報をパッチとして符号化する際に，range encoder ベースのアルゴリズムにより理想的なエントロピーに近いサイズのファイルを生成した．
- 既存手法により 3.8 MB まで圧縮された AlexNet に対し 300kB のオーバーヘッドのみで段階的に読み込みが可能な圧縮を実現できることを示した．

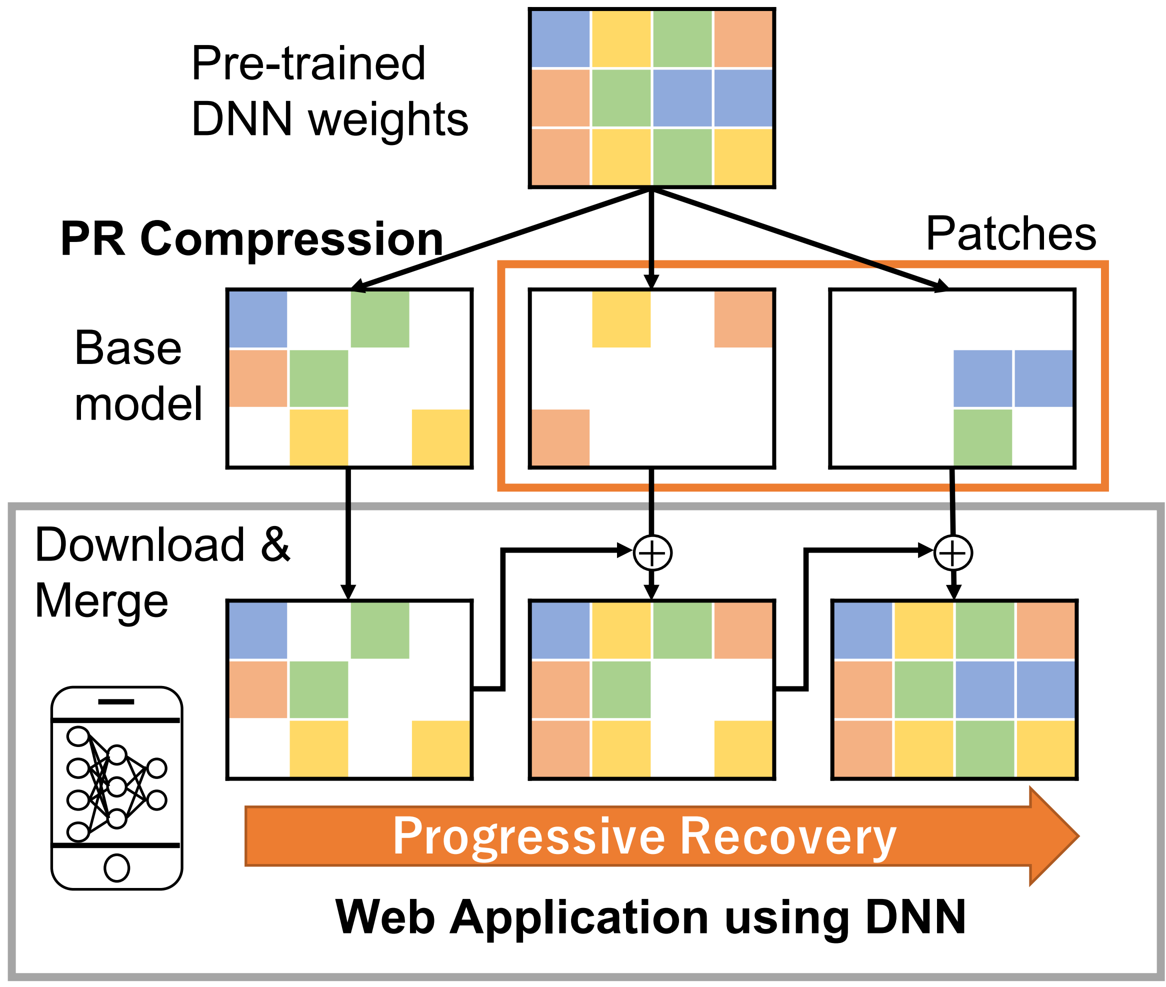


図 5.1 提案システムの概要．提案システムが学習済み DNN モデルを分解し，精度は低いがサイズが小さいベースモデルと，失われた情報を補完する複数のパッチを出力する．モバイルクライアントは最初にベースモデル をダウンロードし，アプリケーションの実行を開始する．次にバックグラウンドでパッチをロードし，モデルを再構築する．再構築が進むにつれてモデルの精度が回復する．

5.2 関連研究

DNN には基本的に冗長なパラメータが多数含まれており，これを除去することで，わずかな精度損失でサイズを非常に小さくできることが多くの研究で報告されている．既存の圧縮手法はパラメータのプルーニングと共有 (量子化)，低ランク分解，transferred/compact convolutional filters，knowledge distillation に大別される [60]．後ろ 2 つは，精度とモデルサイズのトレードオフごとに異なるモデル構造の設計が必要となる．本研究では，システマティックにサイズを細かく制御できるプルーニング (pruning)，量子化 (quantization) ベースの手法の上に開発する．プルーニングは，精度への影響が小さいパラメータを 0 にすることで，必要なパラメータ数を削減する手法である．非ゼロ要素値とともに，その値のインデックスを保存する必要がある．

初期の研究である Optimal Brain Damage [61] では，プルーニングはパラメータの二次微分をもとに行われていた．近年の深層モデルでの基本的な手法は，閾値未満のパラメータをプルーニングすることである [62]．より洗練された手法として，Yu ら [63] はあるニューロンの削除が最終層の 1 つ前の層に与える影響を定式化した．DNN の高速化に関しては，要素レベルでのプルーニングは理論値ほどの速度向上をもたらさない．不均一なメモリアクセスを行うスパース行列積が必要なためである．そのため，行列の行を単位としたプルーニング手法，すなわちチャンネル単位のプルーニングが提案されている．

Li ら [64] は，行ごとの L1 ノルムを計算し，それが小さい行から順にプルーニングする手法を提案した．Hu ら [65] は，データセットを用いて ReLU レイヤーの出力が非ゼロになる確率を計算し，その確率が大きいものから順にプルーニングした．He ら [66] は LASSO 形式の最適化問題に帰着させ，明示的な最適化を実行した．プルーニングでは，プルーニング率が主なハイパーパラメータである．プルーニングによる影響の大きさはレイヤーごとに異なるため，レイヤーごとに適切なハイパーパラメータを設定する手法が有用である．Chen ら [67] はハイパーパラメータと精度の関係をブラックボックス最適化問題としてとらえ，ベイズ最適化を適用した．He ら [68] は強化学習の応用を提案した．

量子化は，パラメータをクラスタリングし，1 つのクラスタに属するパラメータに同じ値を共有させる．各パラメータが属するクラスタインデックスと，クラスタを代表する値を保存する必要がある．最も単純な量子化は，線形量子化である．Sung ら [69] は DNN における線形量子化のビット幅と精度の関係を検証した．大きな値は DNN の出力に与える影響が大きいという考えの元，Miyashita ら [70] は非線形量子化として対数スケールを用いた．Park ら [71] は，大きな値の表現および頻繁に生じる小さな値の表現のバランスをとる手法として weighted-entropy-based 量子化を提案した．Park ら [72] は数少ない大きな値を量子化することなく浮動小数点数で表現した．Chen ら [73] は DNN の初期状態において各パラメータにランダムなクラスタ番号を与え，同じクラスタのパラメータは同じ実数値を持つように制約しながら学習を行った．より積極的な量子化として二値量子化も提案されている．XNOR-Net [74] はパラメータとともにニューロンの値も量子化し，ビット操作により計算が行える二値畳み込み層を提案し，58 倍の計算速度向上を達成した．本研究では，スカラー値のパラメータを独立して量子化するスカラー量子化を採用する．ベクトル量子化は，例えば畳み込みフィルタのように複数のパラメータからなるベクトルを量子化する [75][76]．

Wang らは，複数の量子化ビット数の段階における圧縮モデルを作成し，その差分のみを転送することでモバイルデバイスにおいて段階的に精度を向上させる枠組みを提案した [77]．本研究の枠組みと類似するが，精度低下を補う方法が量子化結果に対する各クラスタ中心の補正のみであり，ビット数が小さい場合に自由度が極めて低い点，大きな圧縮効果をもたらすプルーニングと組み合わせた場合に必要な符号化が考慮されていない点が異なる．

Han らは，プルーニングと量子化が相補的であると指摘し，プルーニングを行ったモデルをさらに量子化する手法を提案した [78]．DNN において大きな値を持つパラメータは相対的に少ないため，リニアスケールによる量子化結果をエントロピーベースの可逆圧縮方式である Huffman 符号によりさらに 20 から 30%圧縮した．結果として AlexNet を精度低下なしに 1/35 まで圧縮できた．Tung と Mori は，プルーニングと量子化を同時に行う枠組み CLIP-Q を提案

した [79]．学習済みモデルのファインチューニングの際に，潜在状態として実数値の重みを保持し，1 イテレーションごとにプルーニング，量子化を行い，誤差逆伝播で得られた勾配で実数値を更新する．AlexNet を精度低下なしに 1/51 まで圧縮できたとしている．なお，ここでいうファインチューニングはあるデータセットで学習されたモデルを他のデータセット向けに調整することではなく，圧縮により変化したモデルに対し同一のデータセットでの精度を回復するために調整するものである．本研究では，CLIP-Q により精度低下なしにプルーニング，量子化されたモデルを入力として用いて，精度の低下を許容しながらさらに圧縮していく．これらの研究では，レイヤーごとのプルーニング率や量子化ビットがハイパーパラメータにより定まる．基本的に，入力となる学習済みモデルに対して精度の低下が生じないハイパーパラメータにおいて精度が報告されているが，より強い圧縮となるハイパーパラメータを設定することで，精度の低下と引き換えに小さなサイズのモデルを得ることができる．単純に学習済みモデルのプルーニング, 量子化を行った場合，精度が大きく低下する．多くの先行研究において，失った精度を回復するため，ネットワーク全体のファインチューニングが行われる．図 5.2 は，CLIP-Q における 2 つのハイパーパラメータでのファインチューニング前後の値である．ファインチューニングの前はこれらの違いはわずかであり，アグレッシブに圧縮されたモデルパラメータに対し，小さな差分 (パッチ) を組み合わせることでより精度の高いモデルを復元できる．しかし，圧縮により失われた精度を補うために生じるパラメータの変化は予測できず，異なるハイパーパラメータに対するファインチューニングの結果は全く異なる．そのため，パッチを効率的に符号化することができない．本研究では，比較的少数のパラメータを持つバッチ正規化レイヤーのみをファインチューニングし，精度低下を補う．

特別なソフトウェアやハードウェア実装を用いて，プルーニングや量子化結果のモデルを高速に実行している研究もある [80][81]．速度の向上幅がハードウェアに大きく依存するため，本研究ではモデルのサイズのみに注目する．

5.3 手法

本研究では，学習済み DNN モデルを入力として受け取り，それを圧縮して小さなベースモデルと，圧縮過程で失われた情報を補完する複数のパッチを出力する．DNN モデルは画像分類等で用いられる Convolutional Neural Network を想定し，誤差逆伝播可能な損失関数が定義されている必要がある．圧縮における制約条件として，ベースモデルとすべてのパッチを組み合わせることにより，もとの学習済みモデルとほとんど同じ精度のモデルが復元できなければならない．この制約条件の下で，ベースモデルとパッチの合計サイズができるだけ小さくなるような圧縮手法を提案する．

5.3.1 パッチ表現を考慮した反復的な圧縮

プルーニング, 量子化は DNN の圧縮の一般的な手段の 1 つである．より多くのパラメータをプルーニングしたり，量子化のビット数を下げたりすることにより，精度の低下と引き換えにモデルサイズを小さくすることが可能である [78]．Tung と Mori は，学習済みモデルをプルーニ

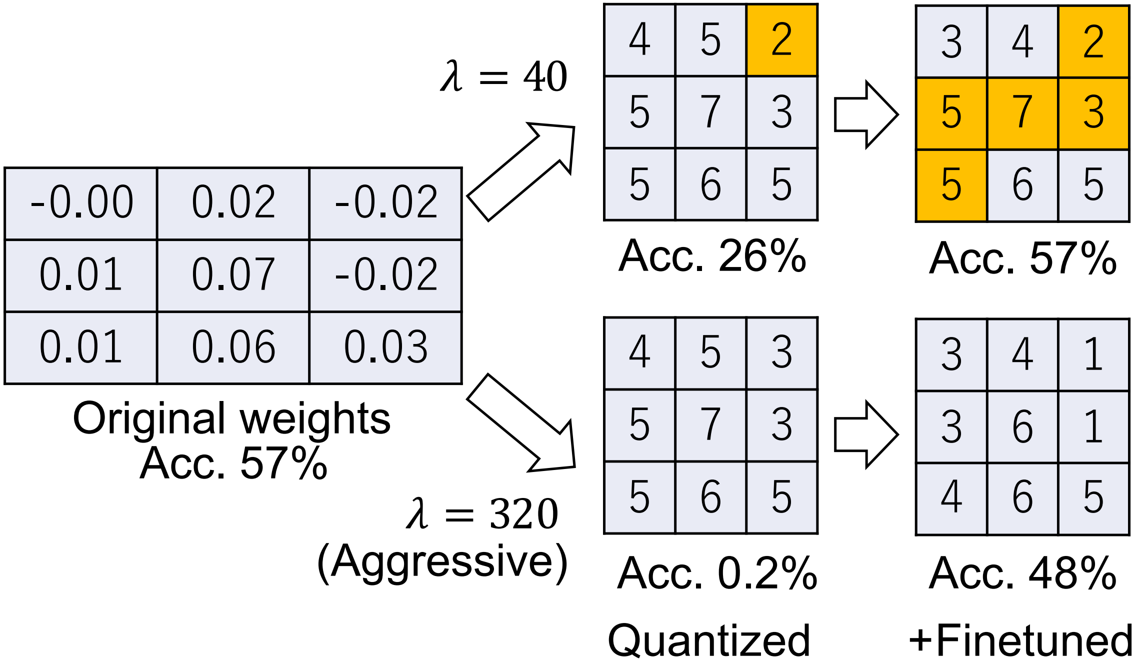


図 5.2 既存手法による AlexNet モデルの圧縮結果の例 (畳み込みカーネルの 1 つを抽出したもの)．Acc. はモデルの正解率 (accuracy) を表す．ハイパーパラメータ λ により精度とサイズのバランスが変化する．失われた精度を回復するためのファインチューニングにより，重みが λ ごとに異なる変化を生じるため，差分を効率的に表現できなくなる．

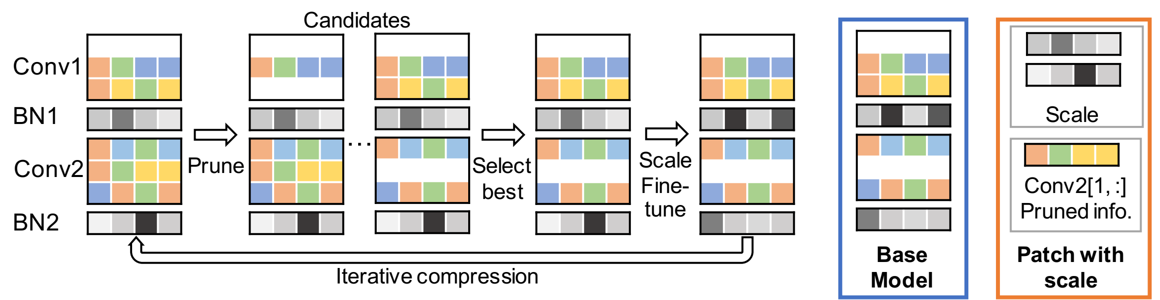


図 5.3 手法の概要．図示した圧縮ステップを，精度が閾値を下回るまで反復的に実行する．圧縮が完了すると，ベースモデルおよび複数のパッチが得られる．図の右側に，1 ステップで出力されるデータの例を示す．

ング，量子化し，精度を低下させずに AlexNet を 1/51 まで圧縮した [79]．このモデルを，学習済みモデルと同じ精度が得られる最小のモデルであるとみなし，本研究ではこのモデルを圧縮の出発点とし，完全モデル (complete model) C と呼ぶ．完全モデルから情報を反復的に削除することで，精度の低下と引き換えにさらにサイズを小さくしていく．事前に圧縮を行うのは，パッチ生成が可能となる制約条件を課さずに圧縮を行う方が精度の維持が容易であるため，精度が低下しない領域では自由度の高い手法を採用すべきであると考えられるからである．本研究では，削除された情報をパッチとして表現し，ベースモデルの精度を学習済みモデルの精度

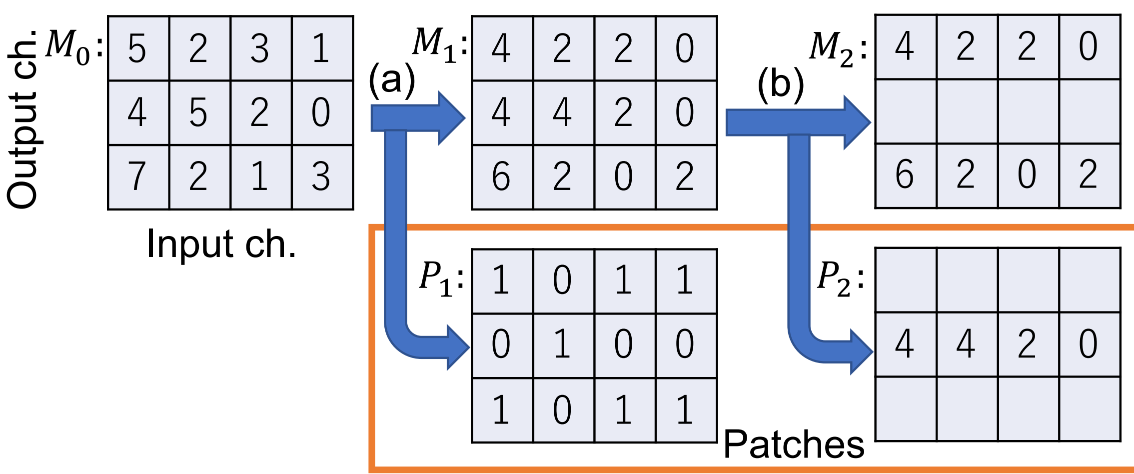


図 5.4 パッチ生成の 2 つの手段．1 ステップで，1 つのレイヤーの (a) 量子化ビット数または (b) 出力チャンネルが削除される．削除された情報はパッチとして表現される．

まで復元するために用いる．モデルのパラメータは事前に量子化されていることを前提とし，量子化結果の整数値は，実数値の大小関係と一致しているものとする．この条件は，圧縮の手段として最下位ビットを削除する操作を用いるためである．線形量子化や対数スケールによる量子化はこの条件を満たしている．

モデル M_t は量子化された重みレイヤー (具体的には畳み込み層および全結合層) $W_{t,i}$ および各重みレイヤーの直後に配置されるスケール (バッチ正規化) を 1 つのベクトルへまとめたもの S_t で表す．重みレイヤー $i \in \{1,...,N\}$ は初期ビット数 T_i，出力チャンネル K_i，出力チャンネルあたりのパラメータ数 M_i(畳み込み層においては，入力チャンネル × カーネル高さ × カーネル幅) を持つ．圧縮の過程を図 5.3 に示す．このアルゴリズムでは，モデルから少量の情報を削除する操作を反復的に行う．そして，PR Compression の条件として削除された情報はパッチとして効率的に表現できる必要がある．1 回の圧縮ステップ t では，1 つのレイヤー i を選択し，削除した情報をパッチとして効率的に表現可能な操作として，(a) 量子化ビット数削減 (b) チャンネル単位のプルーニング (図 5.4) のいずれかを適用する．量子化ビット数削減 $M_{t+1}, P_{t+1} = DecrementLSB(M_t, i)$ は，量子化されたパラメータの最下位ビットをモデルから削除し，削除された情報はパッチ P_{t+1} として分離する．1 回の操作で K_iM_i ビットの削除となる．量子化後の値に対応する実数値は，量子化ビット数ごとのテーブルに格納される．格納される値は，量子化後の値にひもづいたパラメータすべての平均値を用いる．このテーブルの容量は 1 レイヤーあたり $32(\sum_{b=1}^{T_i} 2^b)$ であり，$T_i = 8$ に対して 2kB となる．チャンネル単位のプルーニング $M_{t+1}, P_{t+1} = PruneChannels(M_t, i)$ は出力チャンネルに対応するパラメータをモデルから削除し，パッチ P_{t+1} として分離する．1 チャンネル削除するたびに T_iM_i ビットの削除とな

る．パッチには削除されたチャンネルのデータそのものとともに，削除されたチャンネルのインデックスが必要であるが，これはたかだか 16 ビットである．要素単位で削除すると，T_i ビット削除するためにそのインデックスを表す 24 から 32 ビット程度の情報が必要となり大きなオーバーヘッドとなってしまう．精度の低下を最小化するため，各ステップにおいて，すべての可能な操作に対する結果を候補として，学習データの一部で精度 $A(M_t)$ の評価を行い，最大精度の操作を採用する．これらを反復的に行い，モデルの精度が許容できる最小値 A_{min} まで低下したところで終了する．A_{min} の値はアプリケーションに依存する．この最終的なモデルがベースモデルとなる．パッチを生成の逆順に適用することにより，完全モデルを復元することが可能となる．

チャンネル単位のプルーニングにおいて，すべてのチャンネルを独立した候補として扱うことは計算量的に現実的ではない．例えば AlexNet では $\sum_i K_i = 10568$ の候補がある．そのため，レイヤーごとにチャンネルを重要度順にソートし，最も重要でないチャンネル K_i/G 個を一度に削除する．チャンネルの重要度は明らかではない．チャンネル単位のプルーニングを行っている従来研究では，チャンネルに対応する重みベクトルの L1 ノルムが大きいものが重要であると考えている [64]．一方で，L1 ノルムが大きな重みは量子化された際に低頻度の数値を含んでいるため，後述のエントロピーベースの符号化において大きなサイズを必要とする．そのため，このような重みをプルーニングすることによる圧縮の効果が高い．これはモデルにより異なるため，実験的に確かめる必要がある．最終層については，チャンネル単位のプルーニングを行うと該当するクラスの出力が不可能となるため行わない．

1 ステップあたりの精度評価回数は，$2N$ となる．すべての情報を削除する場合，$2N\sum_i(T_i + K_i/G)$ となる．実際にはすべての情報が削除される前に精度が A_{min} を下回るため，これよりは短時間で終了する．

5.3.2 精度回復のためのスケールファインチューニング

前の節で述べた圧縮が進行することにより，モデルの精度が低下していく．しかし，典型的な対策であるモデル全体のファインチューニングでは，パッチを用いて完全モデルを復元することができなくなってしまう．そこで，少量のパラメータの補正により精度の回復を試みる．全体に影響を与えるパラメータとして，量子化されたパラメータに対応する実数値と，バッチ正規化レイヤーにおけるチャンネル毎のバイアス・スケールが存在する．前者は，量子化ビット数が 3 ビットのような低いビット数の場合，ほとんど自由度がない．本研究で入力とするモデルはこのような低いビット数の場合もあることや，削除されたチャンネルに対応する出力に対して補正をかけることができない．後者では，常に 256 チャンネル程度のパラメータがあること，またチャンネル削除により出力が定数となったチャンネルにおいて，後続のレイヤーに対する影響が小さくなるような値へと補正を行えるというメリットがある．このアイデアは，中間特徴マップの変化を少数のパラメータで補正するもので，ドメイン適応における AdaBN [82] に近い．モデルの精度が一定量 ΔA 低下するごとに，バッチ正規化レイヤーのみをファインチューニングする．バッチ正規化レイヤーはプルーニング・量子化の対象とせず，ファインチューニングで得られたパラメータはパッチに添付する．完全モデルの復元には不要な情報で

あるためサイズのオーバーヘッドとなるが，実験で精度維持への大きな貢献があることを示す．スケールは，容量削減のため 16bit 浮動小数点数で表現した．実験において，32bit 浮動小数点数から 16bit 浮動小数点数へ変換することによる精度の低下は 0.1%未満であった．パッチ 1 つあたり，$2 \times 16 \sum_{l=1}^{N} K_i$ ビットを消費する．先頭の 2 はバイアスとスケールに対応する．

全体の圧縮アルゴリズムを Algorithm 1 に示す．ハイパーパラメータ G は大きいほうが，ΔA は小さいほうが良い圧縮結果が得られると期待できるが，一方で計算時間の増大を招くためトレードオフとなる．

Algorithm 1 モデル圧縮のアルゴリズム

Require: Quantized model M_0, minimum allowed accuracy A_{min}
Ensure: Base model M_B, patches $P_1, ..., P_B$, scales $S_1, ..., S_B$
1: $t \leftarrow 0$
2: **repeat**
3: $A_r \leftarrow A(M_t)$
4: **repeat**
5: $t \leftarrow t+1$
6: **for all** $i \in 1, ..., N$ **do**
7: $\tilde{M}_{t,i,bit}, \tilde{P}_{t,i,bit} = DecrementLSB(M_{t-1}, i)$
8: $\tilde{A}_{t,i,bit} = A(\tilde{M}_{t,i,bit})$
9: $\tilde{M}_{t,i,ch}, \tilde{P}_{t,i,ch} = PruneChannels(M_{t-1}, i)$
10: $\tilde{A}_{t,i,ch} = A(\tilde{M}_{t,i,ch})$
11: **end for**
12: $M_t, P_t \leftarrow \tilde{M}_{t,*,*}, \tilde{P}_{t,*,*}$ where maximum $\tilde{A}_{t,*,*}$
13: **until** $A(M_t) > A_r - \Delta A$
14: $S_t \leftarrow ScaleFinetune(M_t)$
15: **until** $A(M_t) > A_{min}$
16: $B \leftarrow t$

5.3.3 可逆圧縮による効率的な符号化

量子化の結果出現する数値は出現頻度が大きく偏るため，ベースモデル・パッチを具体的なファイルとして表現する際，Huffman code [83] などの可逆圧縮アルゴリズムを用いることは有用である [78]．Huffman code では，出現頻度の高い数値に短いビットの符号，低い数値に長いビットの符号を割り当てることによりデータを圧縮する．しかしながら，パラメータ 1 つを表現するために最低でも 1 ビットを要するため，*DecrementLSB* 操作で生じるパッチに含まれる 1 ビットまたは 2 ビットの値を効率的に圧縮することができない．なお，連続する *DecrementLSB* のパッチを結合した際に 1 ビットより大きなパッチが生じる．例えば，2 ビットの量子化された値 (シンボル)A，B，C，D があり，それぞれの出現確率が 3/4，1/8，1/16，1/16 であるとする．エントロピーの平均は $\sum -p_i \log(p_i) = 1.186$ と計算される．しかし Huffman code では，各シンボルを表現する符号は 0，10，110，111 となり，これらの頻度による重みづけ平均は 1.375 ビットとなる．すなわち理想値に対して 16%の増加となってしまう．Han らは，Huffman code とスパース行列の表現法を組み合わせることにより部分的に解決を行った．非ゼロの値は Huffman

code で表現し，隣接する非ゼロの値の間にあるゼロの個数を別に表現する．この手法は一種類の値がほかの値より圧倒的に高い出現頻度の場合に効率的である．本研究の課題において，パッチ内の値は単一の支配的な値があるわけではない．このような条件で効率よく圧縮を行うための手法として，range encoder [84] を活用した．この手法では，符号のビット長の最適値が整数でない場合でも整数値に切り上げられることなく理論値に近い符号を生成することが可能である．range encoder の概念を解説する．range encoder は，シンボル 1 つ 1 つに対応するビット列を割り当てるのではなく，シンボル列全体に対して単一の数を割り当てて符号化する．例えばシンボル A，B があり，出現確率が 0.8 と 0.2 である場合を考える (見やすさのため 10 進数で説明する)．シンボル列は実数 [0.0, 1.0) の区間を分割するものであるととらえる．シンボル列「A」は，[0, 0.8)，「B」は区間 [0.8, 1.0) というように出現確率に応じて分割する．後続のシンボルは，現在の区間をさらに出現確率に応じて分割する．「AA」は [0.0, 0.64)，「AB」は [0.64, 0.8)，「BA」は [0.8, 0.96)，「BB」は [0.96, 1.0) となる．そして range encoder における出力符号は，シンボル列に対応する区間を表す実数の最小限の接頭辞である．「AA」に対しては 0.1 や 0.5 などが考えられる．一方，「BB」に対しては 0.9 では「BA」との区別がつかないため，0.97 のように小数点以下 2 桁の接頭辞が必要となる．つまり，生起確率の高いシンボル列は区間が広いため短い接頭辞で，低いシンボル列は区間が狭いため長い接頭辞で表現されることにより，エントロピーに応じた符号長が実現される．ここでは 10 進数の実数を用いて説明したが，実際には二進数を用いて，整数演算により実装される．また接頭辞だけでなく，圧縮時と解凍時で共有されるシンボル頻度表が必要となる．

パッチの符号化の概要を図 5.5 に示す．ここでは，モデルの 1 つのレイヤーが 16 要素からなっており，完全モデルは 3 ビット，ベースモデルは 1 ビットとしている．また，チャンネル削除は適用されていない．チャンネル削除の機能を追加する際は，チャンネルを削除する順序の情報をベースモデルに含め，途中でチャンネルを追加するパッチを作成する．ベースモデルには符号表と全要素の 1 ビット時の符号を含める．符号表には，3 ビットの際の各符号の出現回数と，各ビット数における各符号に対応する実数値を含める．「4」$= 2^{3-1}$ は，プルーニングされた箇所を表す特別な符号として用いる．1 ビット時の符号 (プルーニングを含めた 3 値) は range encoder で圧縮する．圧縮・解凍に必要な符号の頻度表は，3 ビットの際の各符号の出現回数から動的に計算することができる．図の場合では，「1」が 8 回，「4」が 3 回，「5」が 5 回であると計算できる．事前圧縮の際に 95%以上のパラメータがプルーニングされる場合もあるが，range encoder を用いることで 1 符号あたり 1 ビット未満で符号化可能であり，プルーニングされた要素を特別にスパース符号化する必要はない．次に符号を 1 ビットから 2 ビットに拡張するためのパッチの符号化を説明する．ここで，現在の符号ごとに次の符号を別個に圧縮する．図の場合では，ベースモデルにおける符号が「1」の箇所について，次の符号が「3,3,1,1,3,3,1,3」になることを示している．次の符号列は，符号表から動的に計算できる頻度表を用いて range encoder で圧縮される．現在の符号ごとに分離せずに圧縮することも考えられる．現在の符号と次の符号の差分として，「+2,+2,0,0,+2,+2,+2,+2,0,0,+2,0,0」(プルーニングされた箇所は除く) という数列が得られるので，これを「+2」の頻度 7，「0」の頻度 6 を用いて range encoder で圧縮する手法が考えられるが，頻度が一様分布に近づいており，ほとんど圧縮することができない．符号の偏りを最大限利用するには，現在の符号ごとに別の頻度表で

Code Table

Code	0	1	2	3	4	5	6	7
Frequency	1	2	2	3	3	3	1	1
Value@1bit	-1.0				0	0.8		
Value@2bit	-1.7		-0.7		0	0.5	1.3	
Value@3bit	-2.0	-1.5	-1.0	-0.5	0	0.5	1.0	1.5

Patch (1bit → 2bit)

Current code	New codes	Frequency
1	3,3,1,1,3,3,1,3	1: 3, 3: 5
5	7,7,5,5,5	5: 3, 7: 2

Decoded codes

Base model (1bit)

4	1	1	1
1	5	4	1
5	4	1	5
1	5	5	5

Fill new codes

Model@2bit

4	3	3	1
1	7	4	3
7	4	3	5
1	3	5	5

Complete model (3bit)

4	3	2	0
1	7	4	2
6	4	3	5
1	3	5	5

Map value

Real-valued weights

0	-1.0	-1.0	-1.0
-1.0	0.8	0	-1.0
0.8	0	-1.0	0.8
-1.0	0.8	0.8	0.8

0	-0.7	-0.7	-1.7
-1.7	1.3	0	-0.7
1.3	0	-0.7	0.5
-1.7	-0.7	0.5	0.5

0	-0.5	-1.0	-2.0
-1.5	1.5	0	-1.0
1.0	0	-0.5	0.5
-1.5	-0.5	0.5	0.5

図 5.5 パッチの符号化の概要

圧縮することが必要となる．符号を用いて DNN を動作させる際は，符号表を用いて符号を実数値に置換する．

5.4　実験

提案手法の効果を確認するため，2 種類の実用規模の画像分類 CNN モデルで実験を行った．1 つは標準的なベンチマークモデルとして広く用いられている AlexNet，もう 1 つはコンパクトさを重視したより最近のモデル MobileNetV2 である．また，画像生成モデルへの応用も示した．

5.4.1　AlexNet on ImageNet

AlexNet [8] は物体認識コンペティション ILSVRC2012 の優勝モデルであり，6100 万パラメータ，233MB である [4]．畳み込み層が 5 層，全結合層が 3 層ある．AlexNet はバッチ正規化レイ

[4] 実際には，Caffe のサンプルとして用意されている，わずかに構造が異なる CaffeNet http://caffe.berkeleyvision.org/model_zoo.html を用いた．

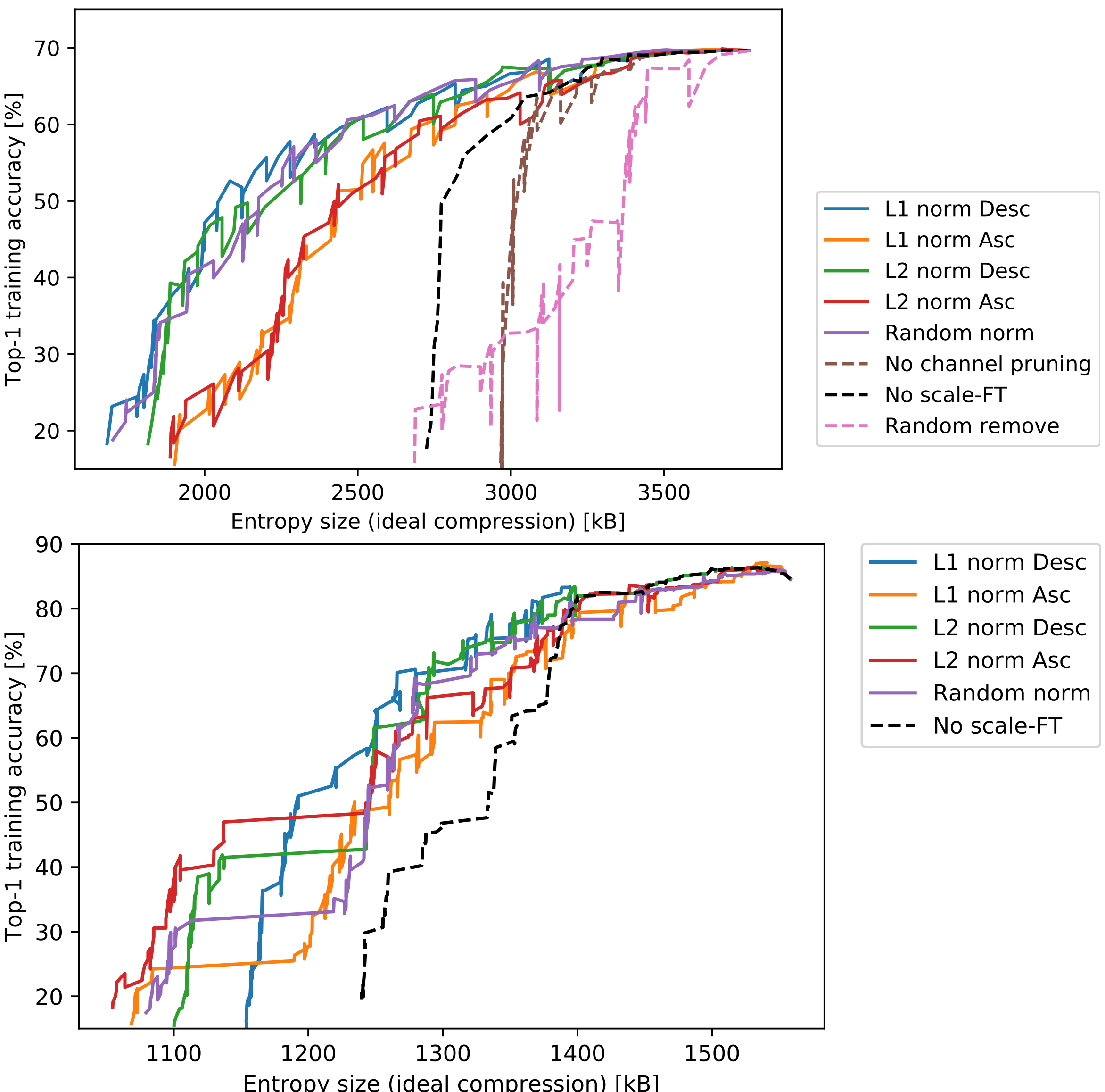


図 5.6 反復的な圧縮過程における，モデルのサイズと学習データに対する精度の関係．上は AlexNet，下は MobileNetV2 の結果を示す．凡例中，上の 5 つはチャンネル単位のプルーニングにおいてチャンネルを削除する順序を比較している．L1 norm desc (descending) は，チャンネルに対応する重みベクトルの L1 ノルムが最大のものから順に削除することを示す．Random norm は，ノルムを無視しランダムな順序でチャンネルを削除することを示す．No channel pruning は，チャンネル単位のプルーニングを行わず，ビット削除のみを用いた場合．No scale-FT は，スケールファインチューニングを行わない場合．Random remove は，精度を測定して最も精度が保たれる手段を選ぶのではなく，ランダムな手段で情報を削除することを示す．

ヤーを持たないため，事前圧縮の後にスケーリング層を $\gamma = 1, \beta = 0$ で初期化して挿入した．学習済みモデルの top-1 validation accuracy は 57.3%である．事前圧縮のため，我々は独自に CLIP-Q のアルゴリズムを実装し，完全モデルとして top-1 validation accuracy 57.4%，3.69MB

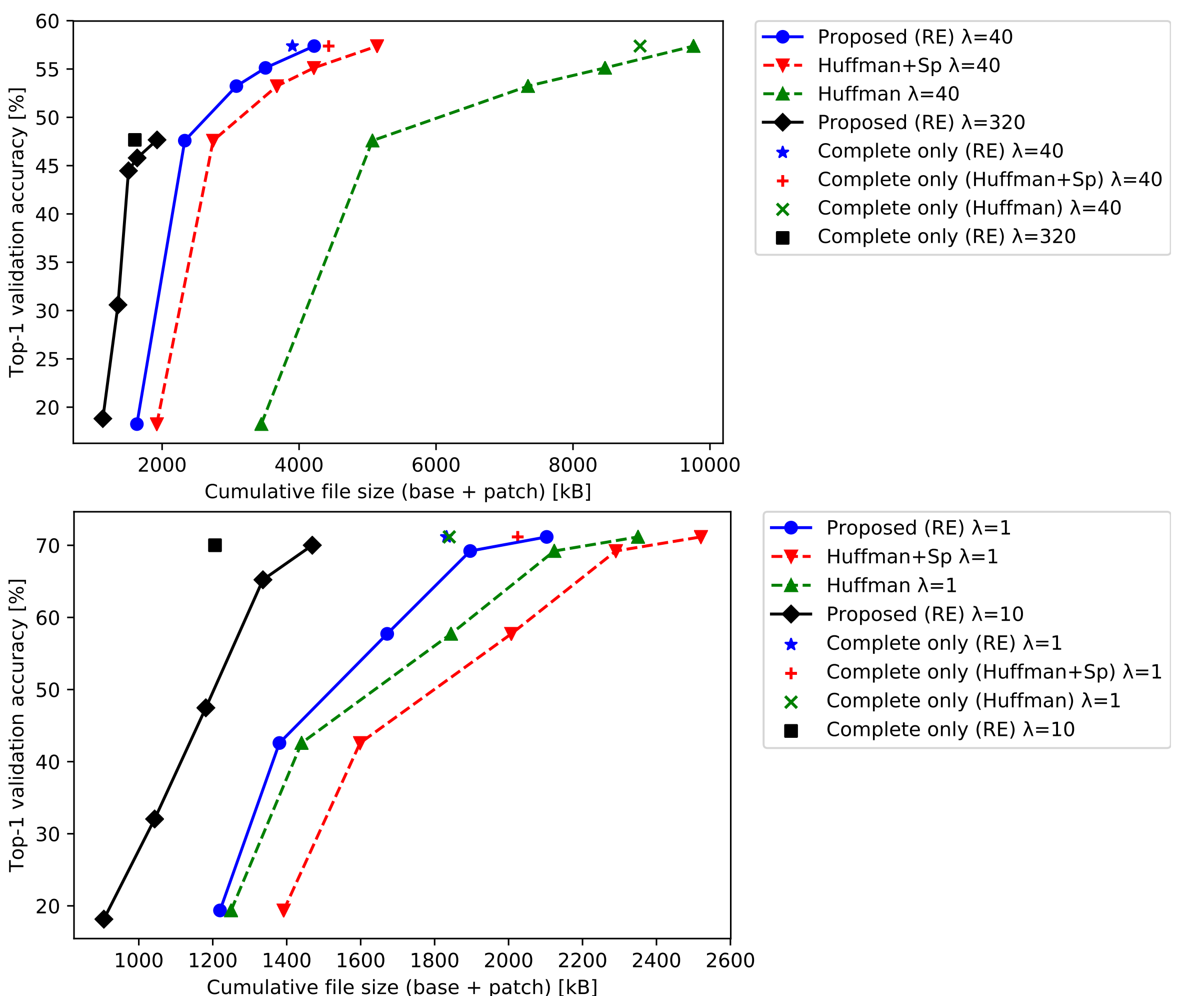


図 5.7 ベースモデルおよびそれにパッチを適用した際のファイルサイズと validation 精度の関係．上側が AlexNet，下側が MobileNetV2 の圧縮結果から生成されている．λ は完全モデルを生成した際の精度とサイズのトレードオフパラメータである．RE は量子化された値を range encoder により符号化した場合，Huffman は Huffman code により符号化した場合を示す．Huffman+Sp は Huffman code とスパース行列の符号化を組み合わせた場合．Complete only は，完全モデルをそのまま提案する符号化で表現した場合．

(entropy) を得た (トレードオフパラメータ $\lambda = 40$ とした)．ここで，entropy size は量子化されたパラメータを出現頻度分布に従って理想的に圧縮した場合の容量を示す．実際の符号化におけるオーバーヘッド・バッチ正規化レイヤーのスケールを含まない．我々はこの完全モデルを入力として提案手法により圧縮を行った．チャンネルグループのサイズ G は 16，スケールファインチューニングの頻度 ΔA は 5%とした．スケールファインチューニングは 5,000 イテレーション，バッチサイズ 128，学習率 0.001 で行った．各ステップでの精度の評価は学習データからランダムに選択した 1,000 枚の画像で行った．チャンネルの重要度計算方法として，L1 または L2 ノルムの昇順・降順のほか，ランダムな値を割り当てる手法を比較した．比較のため，チャンネル単位のプルーニングを行わない，スケールファインチューニングを行わない条

件および精度評価を行わずにランダムに情報を削除する条件での圧縮も行った．図 5.6(上) に結果を示す．右上から圧縮を開始し，圧縮の進行とともにサイズと精度が低下していく．精度が不連続に上がっている点は，スケールファインチューニングを行っている点である．スケールファインチューニングを行わなかった場合，早い段階ではほとんど差がないが，途中から急速に精度が低下している．また，Random remove では圧縮開始すぐに大きく精度が低下した．いずれのモジュールも重要であることがわかった．チャンネルの削除順序は，L1 ノルムが大きいものから順に削除する場合が最良だったが，L2 ノルム，ランダムとの差は小さかった．

圧縮結果をもとに，ベースモデルとパッチを実際のファイルに符号化した場合のサイズを図 5.7(上) に示す．L1 ノルムの降順に従ってチャンネルを削除する条件での圧縮結果を用いている．エントロピーベースの符号化方式として，range encoder と Huffman code を比較した．また，Huffman code にスパース行列の符号化を組み合わせる手法も加えた．ファイルには，バッチ正規化のスケールも含んでいる．パッチは数十から数百個にのぼるため，マージして 4 つのパッチとした．1 つのパッチには，スケールデータは 1 セット分しか含めていない．左下のポイントがベースモデルのサイズ・精度であり，パッチと合わせた累積ファイルサイズを横軸にプロットする．

例えば，range encoder により圧縮されたベースモデル (青色の線) は 1,632 kB で，精度は 18%である．最初のパッチは 699 kB であり，これを適用することにより精度は 46%に向上し，累積サイズは 2,330 kB となる．最終的に 4 つのパッチを適用した結果，累積サイズは 4,221 kB で精度は 57%となり，完全モデルと同等の精度に回復する．可逆圧縮手法の比較では，range encoder はもっとも小さなファイルを生成できた．range encoder とスパース行列表現の組み合わせより，range encoder 単体のほうが良い結果であった．青色の星形マーカーは完全モデルから情報を削除せずそのままベースモデルとみなして符号化した場合のサイズを示す．パッチの表現により生じるオーバーヘッドがないため，ベースモデルとパッチの累積ファイルサイズより若干小さなサイズとなる．別の見方をすれば，提案手法により 317 kB のオーバーヘッドにより PR compression が実現できたといえる．

比較のため，事前圧縮手法として用いた CLIP-Q のトレードオフパラメータ λ を 320 に設定し，圧縮結果としてより小さなモデルを得た．このモデルの validation accuracy は 47%となった．このモデルに対しても同様に提案手法による圧縮を行った．しかし，$\lambda = 40$ の場合と比較すると，圧縮による精度低下が大きい．冗長なパラメータのほとんどが事前圧縮の段階で除去されているため，わずかな情報の削除によってモデルが完全に破損してしまっていると考えられる．仮に $\lambda = 320$ で生成された完全モデルをダウンロードしたのち $\lambda = 40$ のモデルをダウンロードすることにより PR compression と同等の機能を実現しようとした場合，合計ダウンロードサイズは 5,505 kB となり，提案手法と比べて 1,284 kB のオーバーヘッドとなる．

図 5.8 は圧縮の過程における各レイヤーの entropy size の推移を表す．fc6 レイヤーは AlexNet において最大のレイヤーであり，圧縮前の状態で 144 MB を占める．事前圧縮により約 1/100 である 1,465 kB まで圧縮されるものの，依然として最大の容量を占める．PR compression の初期段階において，fc6 レイヤーの容量が最も大きく減少し，モデル全体の圧縮に寄与している．容量の減少は 2,300 kB までで停止し，次に減少し始めるのは当初よりサイズが小さい畳み込み層である．同時に，図 5.6 (上) において，サイズに対する精度の曲線の傾きが大きくなって

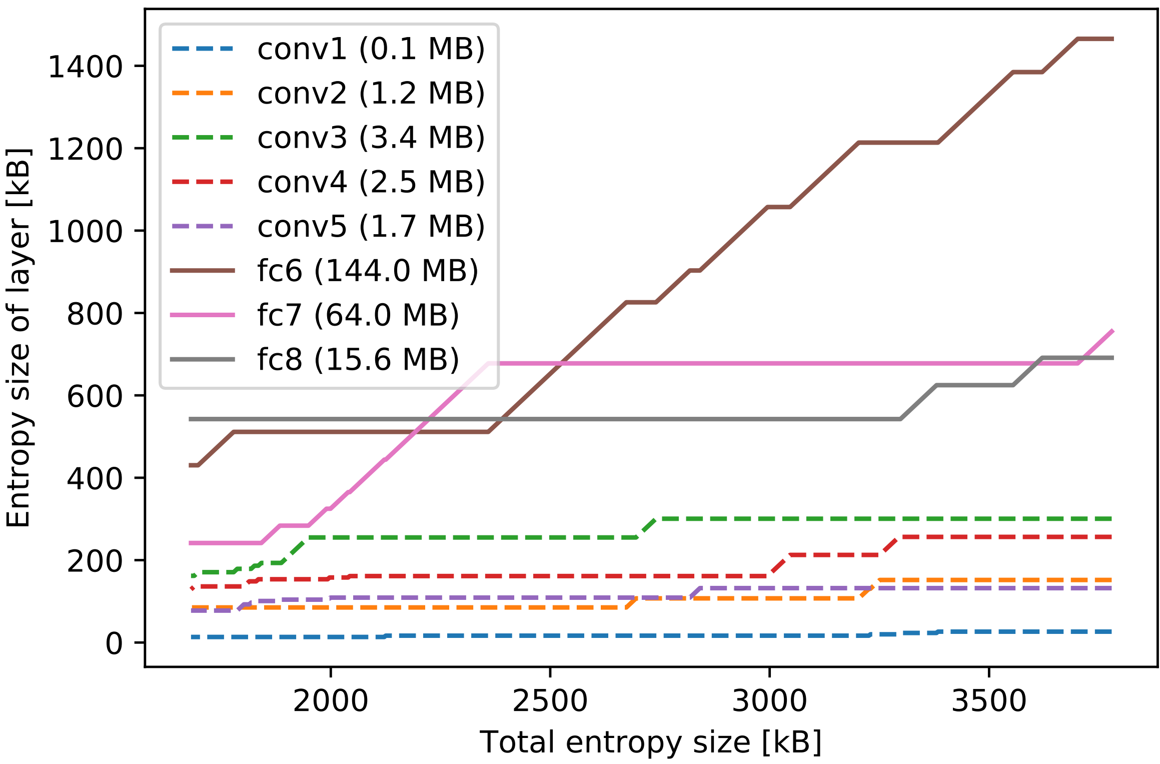


図 5.8 圧縮が進んだ際の各レイヤーのサイズ変化．括弧内のサイズは，無圧縮状態でのレイヤーのサイズを表す．

いる．より細粒度に情報を削除することができれば，大きな容量を占める fc6, fc8 レイヤーからさらに情報を削除することができ，畳み込み層から情報を削除する場合と比べて精度の低下を抑えられる可能性がある．

Wang らの手法 [77] との比較を図 5.9 に示す．Wang らは，畳み込み層を 10 ビット，全結合層を 5 ビットに量子化したものを入力とし，圧縮を行なっている．提案手法においては，層の種類によるヒューリスティックな条件設定は行わず全ての層を 8 ビットに量子化したものを入力とし，圧縮を行なった．プルーニングは適用していない．事前圧縮を行わない条件においては，Wang らの手法と提案手法は近い性能を発揮している．提案手法は，事前圧縮においてより低いビット数となったモデルに対して有効と考えられるスケールファインチューニング，プルーニングが適用されている際の効率的な符号化を提案している点で異なる．

5.4.2 MobileNetV2 on ImageNet

MobileNetV2 [57] は Depthwise Convolution [85] をコアブロックとして採用し，モデルサイズをコンパクトにすることを目指した画像認識モデルである．350 万パラメータ，13.5MB ある．畳み込み層が 53，全結合層が 1 ある．学習済みモデル [5] の top-1 validation accuracy は 71.6%．CLIP-Q による圧縮で，$\lambda = 1$，完全モデルとして top-1 validation accuracy 71.2%，1.52MB (entropy) を

[5] https://github.com/shicai/MobileNet-Caffe

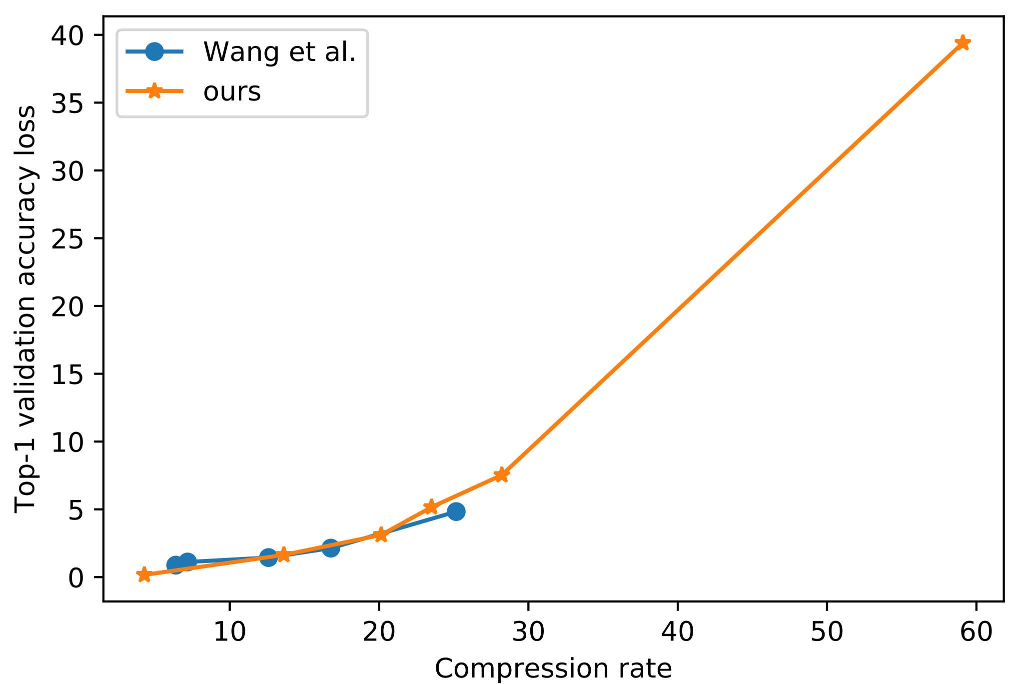


図 5.9 事前圧縮を行わない場合の手法間の比較

得た．この完全モデルを提案手法により圧縮した．チャンネルグループのサイズ G は 16，スケールファインチューニングの頻度 ΔA は 5%とした．スケールファインチューニングは 5,000 イテレーション，バッチサイズ 32，学習率 0.01 で行った．図 5.6 の下側に圧縮結果を示す．結果は AlexNet とおおむね同等の傾向を示したが，1,150kB 付近にて L2 ノルムが小さい順に削除する条件が最良となっている．このモデルはチャンネルの重要度とエントロピーの関係が AlexNet と異なるバランスを持っている可能性を示唆している．また，1,200kB 付近で精度に対して大きくサイズが減少している箇所では，fc7 レイヤーの量子化ビット数が削減されている．fc7 レイヤーは他のレイヤーと比較して多くのパラメータを持つため，大きなサイズ変化につながっている．しかし，この情報の削減は L1 ノルムの小さいチャンネルから削除する条件においては生じていない．提案手法は貪欲法であるため，途中の過程によってはこのステップが最善とはならない場合があることを示しており，全体最適解を求められる手法が将来課題となる．

圧縮結果をもとに，ベースモデルとパッチを実際のファイルに符号化した場合のサイズを図 5.7(下) に示す．L2 ノルムの昇順に従ってチャンネルを削除する条件での圧縮結果を用いている．$\lambda = 1$ で圧縮された完全モデルに対する PR compression の結果，range encoder で符号化されたベースモデルのサイズは 1,219 kB となった．全てのパッチを適用することで，正解率 71.2%で累積ファイルサイズ 2,103 kB となった．仮に完全モデルをそのままベースモデルとみ

なして符号化した場合，1,832 kB となる．パッチの表現によって生じるオーバーヘッドは 271 kB である．完全モデルをそのままベースモデルとみなして符号化した場合，range encoder と Huffman code のファイルサイズの差は小さい．MobileNetV2 はモデル構造の冗長性が小さく，精度を低下させない範囲で事前圧縮を行った場合，AlexNet の場合と比較してプルーニング率は低く，量子化ビットは大きくなる．そのため，可逆圧縮で Huffman code の圧縮率を悪化させる要因が少ないといえる．一方で，パッチの符号化を行なった場合，パッチに含まれる差分のビット数は小さいため，range encoder の優位性が大きくなる．事前圧縮のトレードオフパラメータを $\lambda = 10$ とした場合，完全モデルの正解率は 70.0%となった．同様に PR compression を適用したところ，サイズのオーバーヘッドが比較的大きかった．この原因は，重みパラメータの総数が 340 万である一方，スケールパラメータの総数は 3.4 万あり，スケールパラメータを格納するために各パッチファイルのサイズが 68 kB 増加することによる．AlexNet では重みパラメータの総数が 6,000 万，スケールパラメータの総数は 2.1 万と大きな開きがあり，スケールパラメータを格納することによるオーバーヘッドが比較的小さい．スケールパラメータによるオーバーヘッドを削減するためには，この部分の圧縮についても考慮する必要がある．

5.5 画像生成への応用

画像生成 DNN はエンターテイメントアプリケーションとして活用が見込まれる．PR Compression は誤差逆伝播が可能な損失関数が定義できれば，識別モデル以外への応用が可能である．

提案手法を Neural Style Transfer モデル [86] に適用した．写真を指定した絵画のスタイルに変換するモデルである．Neural Style Transfer の概念を提案したオリジナルのモデル [87] では画像生成に反復的な最適化を要したが，Johnson らにより高速化されたモデルでは，feed-forward deconvolution network [86] を用いて計算することができる．このモデルには畳み込み層が 13 個，deconvolutional 層が 3 個含まれる．畳み込みと deconvolution は提案手法においては同様に扱うことができる．このタスクには正解となる出力画像がないが，変換された画像とスタイル画像を VGG16 モデルに与え，特徴マップの類似度により損失が計算される．詳細は元論文を参照のこと．学習済みモデル [6] に対し，MSCOCO dataset [88] で損失を計算して PR Compression を適用した．手法の節では精度ベースでアルゴリズムを記述したが，符号反転した損失を用いて同様に処理可能である．

圧縮結果を図 5.10 に示す．圧縮前のモデルの容量は 6.5 MB である．ベースモデルは 230 kB で，パッチを合わせた累計サイズ 527 kB の半分以下である．モデルの読み込みが進むにつれて，より鮮明でソース画像を反映した変換が可能となった．

[6] https://github.com/gafr/chainer-fast-neuralstyle-models

Source image　　Style image (used in training)

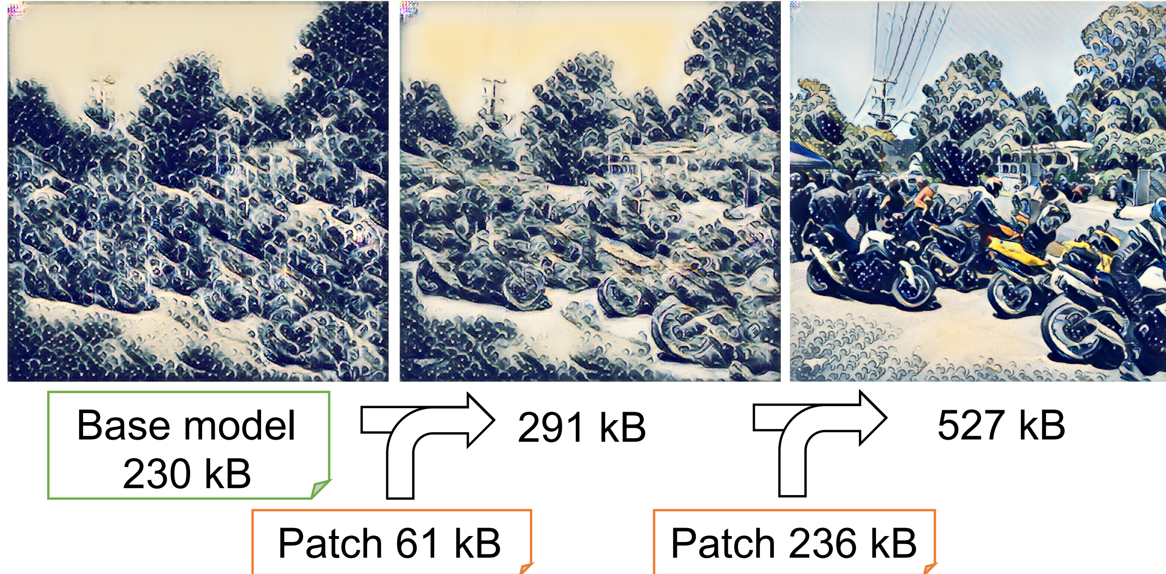


図 5.10 提案手法により圧縮された Neural Style Transfer モデルによる画像変換の結果．パッチ適用の段階ごとに表示．

Chapter 6

Web ブラウザにおける Deep Neural Network の推論高速化

6.1　序論

本章の目的は，DNN の計算を Web ブラウザ上で高速に実行できる計算基盤の構築である．Web ブラウザ上で DNN が実行できると様々なメリットが享受できる一方，実現に当たっては次のような課題がある．(1) 実行速度の問題と，(2) モデル変換の問題である．

実行速度の問題は，Web ブラウザ上のプログラムでハードウェアの性能を引き出すことが難しいという点である．Web ブラウザ上で動的な Web ページを実現するためのプログラミング言語として JavaScript がほとんどの Web ブラウザで実行可能である．任意の計算が可能な言語であるため，DNN を実行するアルゴリズムを記述することも可能である．しかしながら，JavaScript はインタプリタ言語であり，コンパイラ言語である C++言語等と比較すると CPU の性能を十分に活用することが難しいといえる．ConvNetJS [41] は JavaScript で記述された DNN 実行フレームワークであるが，小さなモデルしか現実的な速度で実行することができない．一方，現在利用されている Web ブラウザの多くには，JavaScript から GPU にアクセスするための WebGL と呼ばれる API が搭載されている．WebGL を用いる場合，GPU 上で動作するプログラムは GLSL 言語で記述され，JavaScript は CPU から GPU に GLSL 言語のプログラムおよび処理対象のデータを転送する役割を担う．GPU は CPU より高い演算能力をもつが，WebGL はコンピュータグラフィックス向けに設計されており，汎用的な科学技術計算に用いようとすると GPU の性能を引き出すことが難しい仕様となっている．本研究では，2017 年に Apple 社により提案された新しい API，WebMetal を活用することでこの課題の解決を図る．

モデル変換の問題は，2017 年 4 月現在，複数の深層学習フレームワークがシェア争いをしている状態にあり，標準的なモデルフォーマットが存在しないという問題である．フォーマットの差異は単純なレイヤー名の差異だけでなく，レイヤーがサポートしているオプションの差異など多岐にわたる．本研究では，各フレームワークのモデル構造を反映し統一的に扱うことができる中間表現を開発し，この課題の解決を図る．

本節では，これらの課題を克服した，Web ブラウザ上で高速に DNN を実行するためのソフトウェアフレームワーク WebDNN を提案する．WebDNN は Web ページに組み込まれて動作し，ユーザにとっては通常の Web ページにアクセスするのとなんら変わりなく DNN を用いたアプリケーションを利用できる．そして，Web ブラウザ上での実行速度を向上させるため，WebDNN では DNN の推論フェーズに特化した様々な最適化を実装している．WebDNN は次のような特徴を持つ．

- Web ブラウザ上で利用できるアクセラレータ API を最大限利用し，実行の高速化を図る．
- 複数の深層学習フレームワークのモデル形式に対応し，自動変換機構により多くのモデルを実行可能とする．
- モデルを表現する中間表現上での最適化により，推論に特化した高速化を実現する．
- アプリケーション開発者はどのアクセラレータ API が利用されるかについて意識する必要がなく，単一の簡潔なコードで DNN モデルを実行可能とする．

WebDNN の開発は木倉悠一郎氏と共同で行い，モデルの最適化を中心とした一部の成果については木倉氏の修士論文 [89] にて述べられている．当該部分については簡単な説明にとどめ，引用を表示することとする．

6.2 関連研究

Web ブラウザ上で DNN を実行するフレームワークの初期のものとして，Andrej Karpathy による ConvNetJS [41] が存在する．Web ブラウザ上でモデルの学習と推論両方を行う機構が含まれており，自己完結している．モデル構造は独自の書式で定義することとなっており，他のフレームワークとの互換性はない．また，GPU のサポートは実装されていない．そのため，CPU で学習できる規模の小さな DNN しか利用できない．Keras.js [90] は，Web ブラウザ上で DNN を実行する JavaScript フレームワークの 1 つである．深層学習フレームワーク Keras で学習されたモデルを読み込み，推論を行う．DNN 計算で必要となる行列積の実装には weblas [43] というライブラリを用いており，WebGL を介して GPU 上で計算を行っている．しかし WebGL を介することによるオーバーヘッドが大きく，GPU 本来の演算能力に対してかなり低い計算速度にとどまっている．また，GPU を用いず CPU のみで演算を行う機能も備わっているが，JavaScript で実装されているため，C++言語等で実装されたネイティブアプリケーションと比較すると実行速度が大きく劣る状況となっている．一方で WebDNN では Web ブラウザ上で使える次世代の API を活用することにより，ハードウェアの性能を最大限活用できる．GPU の利用には WebMetal，CPU の利用には WebAssembly と呼ばれる API を用いる．

DNN の実行を高速化するためには，GPU の活用とともに，DNN モデルの計算グラフを最適化することが重要である．DNN モデルは，畳み込み層や活性化層などの層が多数積み重ねられた構造をしており，それぞれの層は行列を受け取り，層の種類やパラメータ (これも行列で表現されることが多い) に従った演算を行い，結果を次の層に受け渡す．このような，行列

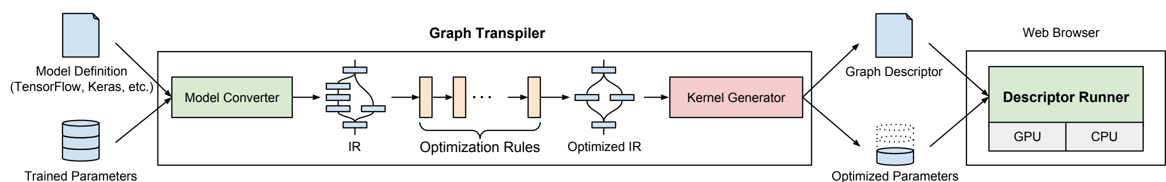


図 6.1 WebDNN の構成

を介した層同士のつながりを計算グラフと呼ぶ．Theano [91] は計算グラフを実行するフレームワークの 1 つであり，高速実行のための最適化機構を実装している．DNN の実行には 2 種類あり，学習段階と推論段階がある．学習段階では，学習用のデータを用いて DNN モデルのパラメータを誤差逆伝播法により更新する．この段階のことは一般に深層学習 (Deep Learning) と呼ばれているものである．推論段階はパラメータがあらかじめ決まった DNN モデルを実際に利用する段階であり，誤差逆伝播を必要としない．Theano は学習・推論の両方で動作するため，誤差逆伝播に必要な情報を保持している．仮に推論のみに特化すれば，この制約がなくなり最適化の自由度が高まる．WebDNN は推論段階に特化し，強力な最適化を実装している．最適化の元となる DNN モデルには，著名な深層学習フレームワーク (Chainer・Tensorflow 等) で学習されたモデルをそのまま用いることができ，学習段階のプログラムを新たに実装する必要はない．

6.3 手法

WebDNN の構成を図 6.1 に示す．

モデルを推論に特化して最適化するため，Graph Transpiler が学習済みモデルをオフラインで変換し，その出力である Graph Descriptor を Web ブラウザに読み込ませる方式を採用している．Graph Transpiler は，最初に各深層学習フレームワークにより生成されたモデルを中間表現に変換し，それを最適化したのち，計算の実行手順およびモデルパラメータを表す Graph Descriptor を出力する．Web ブラウザはまずランライムライブラリである Descriptor Runner を読み込み，Descriptor Runner が実行環境に応じた Graph Descriptor をダウンロード，実行する．

Keras.js では，Keras の各レイヤーに対応した演算を実装することにより Keras モデルを Web ブラウザ上で実行することを可能にしている．Keras ではモデルの構造や学習済みパラメータが HDF5 というデータベース形式で表現されているため，これを Web ブラウザ上で認識可能な形式である JSON に変換する部分のみ Python で実装され，オフラインでデータベース形式の変換のみが行われる．また，各レイヤーを計算するたびに出力のメモリ領域を確保する仕組みとなっているが，GPU 上でメモリの確保を行うために毎回オーバーヘッドが生じる．計算内容が固定されていれば，事前に全データを格納できるサイズのメモリを確保するほうが実行時のオーバーヘッドが少ない．

6.3.1 中間表現による DNN モデルの共通化

深層学習フレームワークは Caffe [17]，Chainer [18]，Tensorflow [19] など多数存在し，それぞれが広く活用されている．それぞれのフレームワークが生成する DNN モデルのファイルフォーマットには互換性がない．複数のフレームワークに対応するため，WebDNN では DNN モデルを表現する中間表現 (Intermediate Representation=IR) を定義し，各フレームワークで生成された DNN モデルを IR へ変換する．さらに，IR 上でモデルを加工することにより，実行速度の最適化を図る．どの深層学習フレームワークであっても計算グラフに相当するデータ構造を利用する点に本質的な違いはないが，モデルの定義方法，多次元テンソルのメモリ上の表現順序，レイヤーのオプションに相違があり，これらを抽象化する必要がある．

モデルの定義方法には，大きく分けて Define-and-run と Define-by-run と呼ばれる方式が存在する．Define-and-run は計算グラフを完全に定義したのち，データを入力して学習または推論を行う方式である．Caffe におけるモデル定義ファイルの一例を示す [1]．ここでは Protobuf と呼ばれる特有の書式で入力データの読み取りおよび各レイヤーの定義を行う．

```
name: "LeNet"
layer {
  name: "mnist"
  type: "Data"
  top: "data"
  top: "label"
  include {
    phase: TRAIN
  }
  transform_param {
    scale: 0.00390625
  }
  data_param {
    source: "examples/mnist/mnist_train_lmdb"
    batch_size: 64
    backend: LMDB
  }
}
layer {
  name: "conv1"
  type: "Convolution"
  bottom: "data"
  top: "conv1"
  param {
    lr_mult: 1
  }
  param {
```

[1] https://github.com/BVLC/caffe/blob/master/examples/mnist/lenet_train_test.prototxt より抜粋

```
    lr_mult: 2
  }
  convolution_param {
    num_output: 20
    kernel_size: 5
    stride: 1
    weight_filler {
      type: "xavier"
    }
    bias_filler {
      type: "constant"
    }
  }
}
```

後発のフレームワークである Keras 等では Python 言語で計算グラフを構築できるが，一度生成されたグラフを用いて全てのミニバッチを学習することが前提となっており，学習中に動的にグラフを変更することはできない．

Define-by-run は，データを用いて計算を行う中で動的に計算グラフを構築していく方式である．Chainer にて最初に提案された [18]．Chainer は Python 言語で記述されており，計算の途中結果をもとに if 文などにより次の計算を変更することも可能な柔軟な仕組みである．記述例を示す [2]．これは強化学習の例であるが，Python 上で for 文等の制御構造を用いて動的に計算内容を制御することができる．記述のしやすさの点において Define-and-run 方式より優れている一方，実行速度の最適化は難しくなる傾向にある．

```
for i in range(caption_length - 1):
    # Compute the loss based on the prediction of the next token in the
    # sequence
    x = Variable(self.xp.asarray(captions[:, i]))
    t = Variable(self.xp.asarray(captions[:, i + 1]))
    if (t.array == self.embed_word.ignore_label).all():
          # Preprocessed captions are padded to reach a maximum length.
          # Often, you want to set the 'ignore_label' to this padding.
          # If all targets are simply paddings, predictions are no longer
          # required.
          break
    y = self.step(x)
    loss += F.softmax_cross_entropy(
          y, t, ignore_label=self.embed_word.ignore_label)
    size += 1
```

[2]https://github.com/chainer/chainer/blob/master/examples/image_captioning/model.py より抜粋

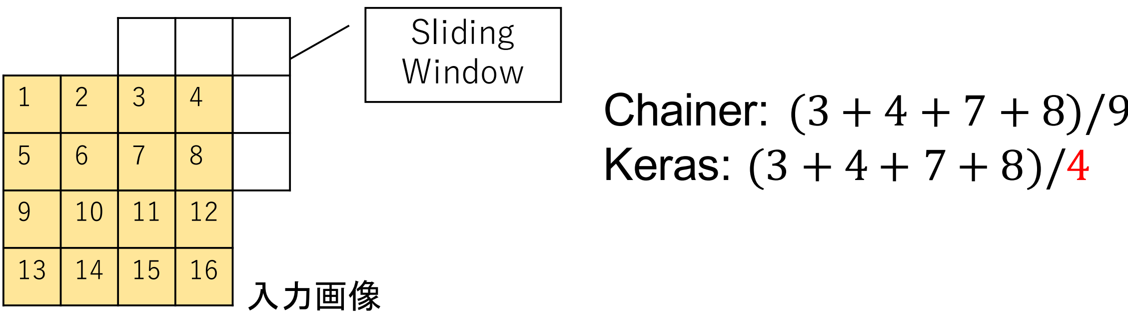


図 6.2 Average Pooling における深層学習フレームワーク間の差異

WebDNN において Web ブラウザ上でモデルを動作させる場合，Python 言語で記述された制御構造を Web ブラウザ上で動作する JavaScript プログラムに変換することは困難であるため，具体的なデータに対してメモリ上に動的生成された計算グラフを抽出し，それを静的なグラフとして用いることとした．Define-and-run 方式のフレームワークについても，各フレームワークの Python 言語インターフェースを用いて計算グラフを抽出するよう実装した．

以上のように，普及しているほぼすべてのフレームワークは Python 言語により計算グラフを操作可能なため，Graph Transpiler は Python 言語により実装した．

画像を扱う DNN では，2 次元の行列でなく，バッチサイズ (N)，チャンネル数 (C)，画像の高さ (H)，画像の幅 (W) からなる 4 次元のテンソルが変数として用いられることが多い．このような多次元テンソルがメモリ上で表現される場合に，深層学習フレームワークにより配置が異なる．例えば Chainer では NCHW という方式が採用される．一方で TensorFlow では NHWC がデフォルトで採用される．これらの記号は，テンソルを 1 次元上のインデックスで表現する場合に要素をスキャンする順序を表す．NCHW の場合，小文字を各次元のインデックスとした場合メモリアドレスは $nCHW + cHW + hW + w$ と計算される．NHWC では $nHWC + hWC + wC + c$ となる．メモリ上の配置順は，パラメータ行列のデータ構造にも反映されているため，この点を IR 上で表現しなければ，畳み込み層などが正常に動作しない．

また，各レイヤーの仕様もフレームワークによって異なる．Chainer では畳み込み層と活性化層は別のノードとして表されているが，Keras では畳み込み層の属性として適用する活性化関数が含まれており，統一する必要がある．Chainer において活性化層が単独で使われる場合にも対応できる一般性を持たせるため，IR 上では活性化関数は単独のノードとして表現することとした．また，暗黙的な挙動の違いが生じるレイヤーも存在する．例えば Average Pooling 層は，図 6.2 のように，Chainer では 0 で埋められた padding 部分を平均計算で考慮するのに対し，Keras では除外している．深層学習フレームワークごとの差異を吸収するため，IR 上のノードでは多数のオプションを規定している．このように，WebDNN 上での実行結果は元のフレームワークの結果と極力一致するように配慮している．数学関数の誤差や浮動小数点数の計算順序の違いによる誤差が発生するため，結果が完全に一致することはないが，実用上問題にならない．

IR では，レイヤーの名称だけでなく，ここで述べたメモリ上の配置方法を表現する．さらに，レイヤーの計算のもつ数学的特徴を属性として表現する．属性は，最適化の段階で活用される．

IR の例を示す．Chainer 上で以下のソースコードにより定義されたモデルは，Chainer 上の計算グラフとしては図 6.3 のように表される．

```
class Model(chainer.Chain):
    def __init__(self):
        super().__init__()
        with self.init_scope():
            self.l1=L.Convolution2D(None, 8, ksize=3)
            self.l2=L.Linear(None, 10, nobias=True)

    def __call__(self, x):
        h1 = F.sigmoid(self.l1(x))
        return self.l2(h1)
```

これを WebDNN における IR へ変換したものを図 6.4 に示す．長方形はオペレータであり，演算を表す．八角形は変数を表す．オペレータには，オプション (3 行目に表示) および属性 (4 行目に表示) が与えられている．オプションは畳み込み層におけるカーネルサイズなどの情報であり，変換元となるモデルから継承される．属性は最適化のためにオペレータごとに定義される情報である．変数のうち，`ConstantVariable` と書かれているものは学習された重みに対応する定数である．定数はモデル構造の情報とともにダウンロード可能なファイルとして出力する必要があるほか，一部の最適化が可能である．変数にはメモリ上の軸の順序が `Order` 属性として付与されている．メモリ上の順序にかかわらず，例えば `H` は画像の高さを表し，畳み込みオペレータはこれを認識して動作する．全結合層の直前の `Reshape` オペレータは任意の形状変換を受け付けるため，ここで軸の空間的な意味は失われる．この場合の `Order` 属性には`?119` のようなユニークな番号が割り当てられる．全結合層は行列積 (`Tensordot`) であり，そのオプションとして `axes` としてどの軸が内積の対象として用いられるか指定される．オペレータの属性を紹介する．`Tensorwise[N]` は，軸 `N` について独立に入力から出力を計算できることを示す．`InplaceOperator` は，入出力の軸の順序が同じ場合，入力と出力を同じメモリ領域上で行えることを示す．

6.3.2 中間表現上でのモデル最適化

モデル変換で生成された IR は，多数のルールにより最適化される [89]．WebDNN は計算グラフの部分構造をより高速処理可能な構造に変換するためのルールを多数備えている．例を挙げる．

MergeElementwise

ReLU ($f(x) = \max(0,x)$) をはじめとした活性化関数は DNN において頻出する処理の 1 つである．活性化関数は基本的に要素ごとに独立した処理であり，相対的に計算負荷が低い．そ

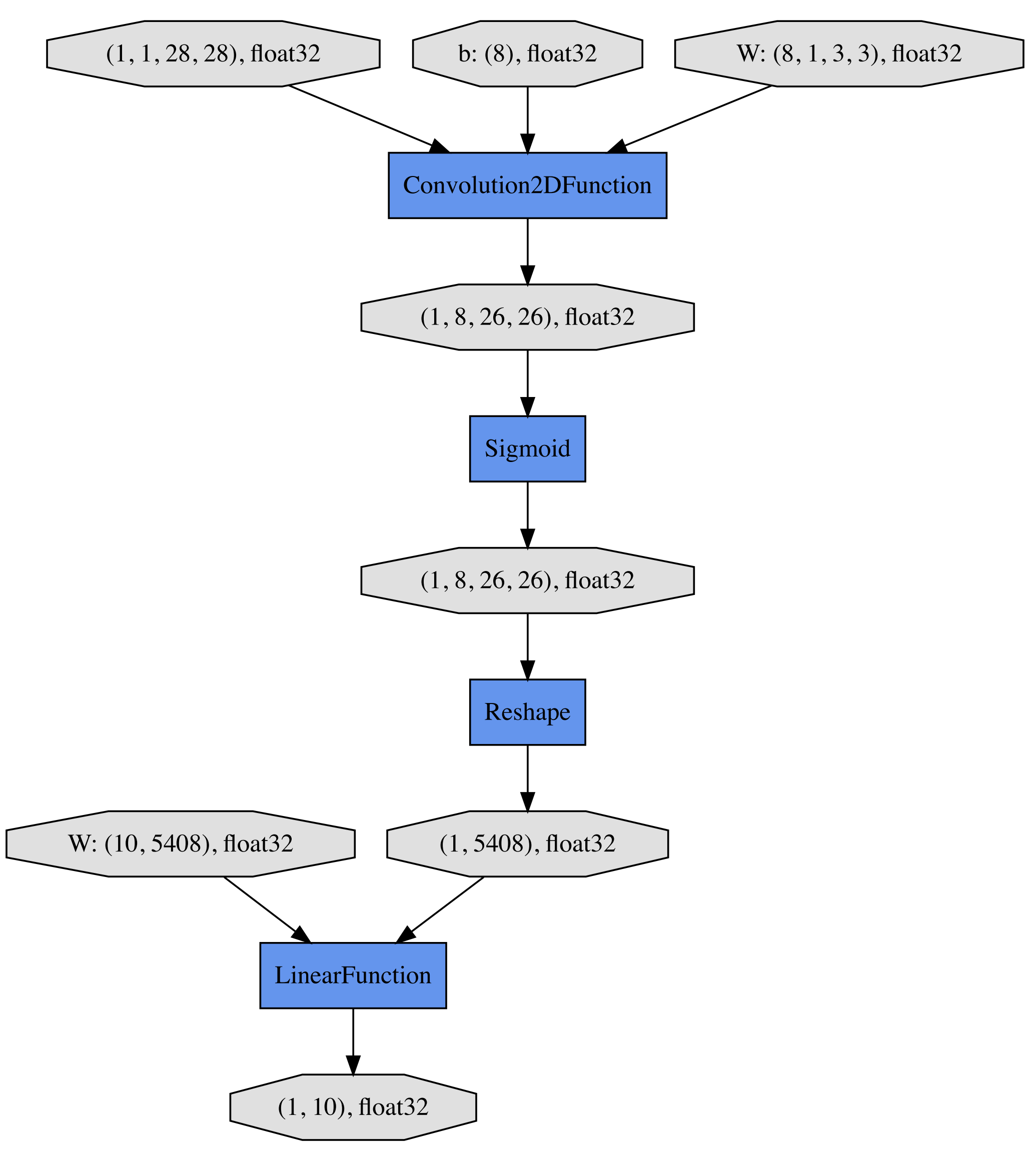


図 6.3 WebDNN に入力される計算グラフの例 (Chainer)

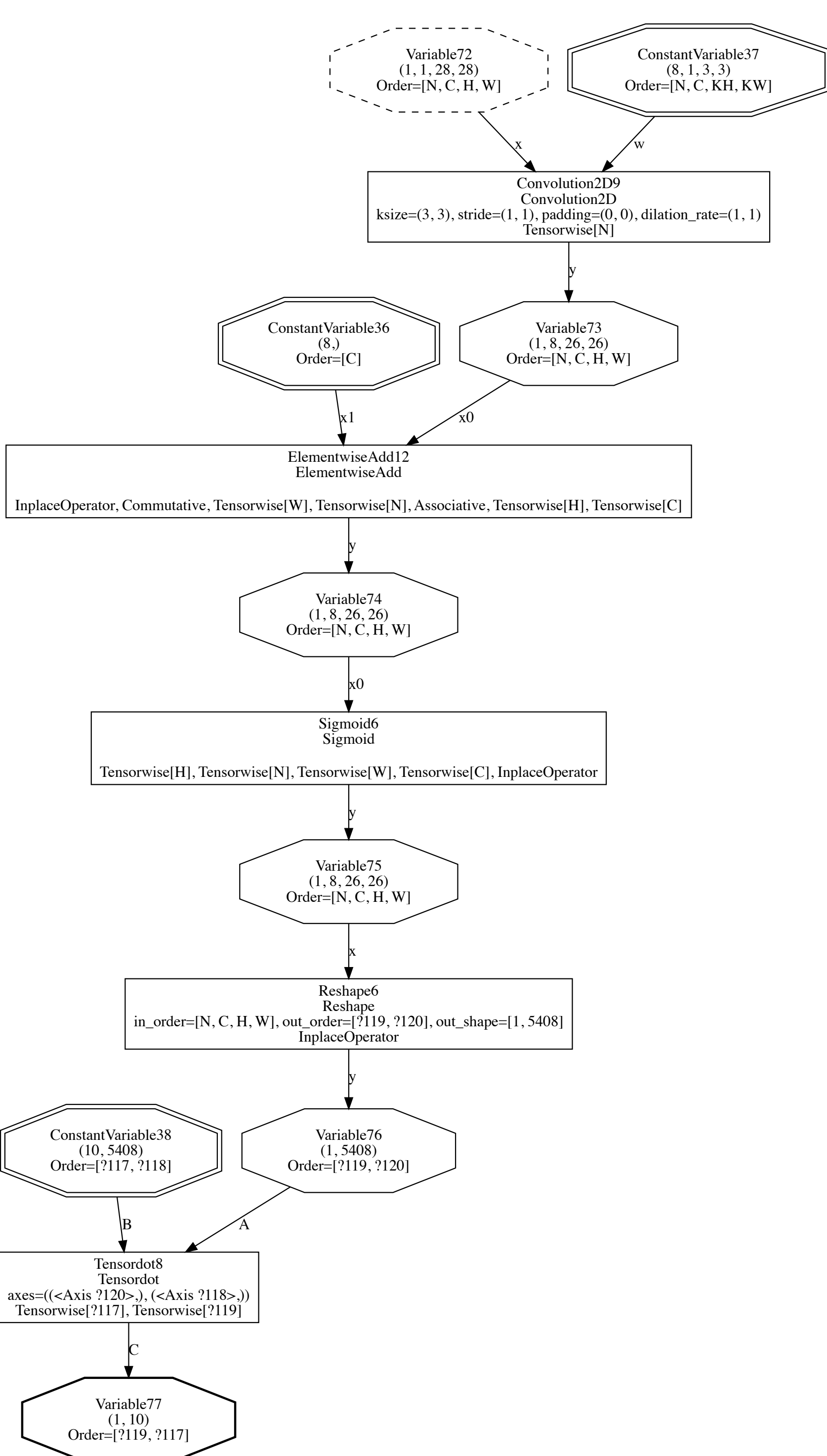


図 6.4 WebDNN の IR の例

のため，このような処理を GPU で実行させる時，JavaScript から GPU を呼び出す操作の所要時間がボトルネックになりやすい．そうすると，GPU の演算コアが活用されず本来の性能を発揮できなくなる．MergeElementwise ルールでは，バイアスの加算と活性化関数のような要素ごとの処理が連続する場合にこれらの処理を 1 回のシェーダに結合し，一度の GPU 呼び出しで処理を完了させる．

MergeAffine

アフィン変換は DNN で頻繁に出現する処理である．MergeAffine ルールは，アフィン変換と他のレイヤーを連結する．例えば全結合層とスケーリング層 (バッチ正規化処理の一部) が連結される．数式で例示すると，ベクトル $\boldsymbol{s}$ をスケール，行列 W を全結合層のパラメータとして，$f(\boldsymbol{x}) = diag(\boldsymbol{s})(Wx)$ という処理を MergeAffine で最適化する．$f(\boldsymbol{x}) = (diag(\boldsymbol{s})W)\boldsymbol{x}$ と変換し，$diag(\boldsymbol{s})W$ は定数であるため，Graph Transpiler 内で事前に計算して 1 つの行列にする．これにより実行時の計算コストが削減される．

WebMetal バックエンド向けに IR を最適化した結果の例を図 6.5 に示す．畳み込み演算を効率的に行える `Im2Col` と `Tensordot` の組み合わせで行うこと，要素ごとに独立に計算可能な，畳み込み層のバイアス加算 `Elementwise` と直後の活性化関数 `Sigmoid` を 1 つのオペレータにまとめた `FusedElementwise` で行うことなどの変換が行われ，実行速度の向上に寄与する．

6.3.3 複数アクセラレータを対象としたコード生成

Kernel Generator は，最適化済みの IR を受け取り Graph Descriptor を生成する機構である．Graph Descriptor は，Web ブラウザ上で DNN モデルを実行するために必要な情報を保持している．Web ブラウザ上で高速に演算するための手段が複数あり，その手段をバックエンドと呼ぶ．Web ブラウザにより対応する手段が異なっているため，複数のバックエンド向けの Graph Descriptor を生成する．現状対応しているバックエンドは，WebMetal，WebAssembly，Fallback (アクセラレータを用いない JavaScript 実装) である．現代のブラウザのほとんどはこれらでカバーすることが可能である．

以下，各バックエンドの特徴を述べる．

WebDNN では Web ブラウザに従来より実装されている WebGL に代わる，新しい GPU API 規格である WebMetal に対応している．WebMetal はコンピュートシェーダと呼ばれる汎用的な計算に有用な機能を備えている．WebMetal を用いることで，GPU を少ないオーバーヘッドで科学技術計算に活用することが可能となる．WebDNN プロジェクトにおいて，WebMetal 規格のコンピュートシェーダをオープンソースの Web ブラウザエンジンである WebKit に実装した [89]．この実装はすでに Apple 社の Web ブラウザ Safari の実験的機能として組み込まれている．WebDNN は WebMetal を活用した初めてのソフトウェアである．

WebAssembly は Web ブラウザ上で CPU を用いて JavaScript と比較し高速な演算が可能となる API である．インタプリタ言語である JavaScript と比較し，機械語に近い言語およびそのバイナリ表現を実行することができる．WebAssembly のコードは C 言語，C++言語で記述されたコードからコンパイルを経て生成することが可能である．WebDNN では，行列積の実装

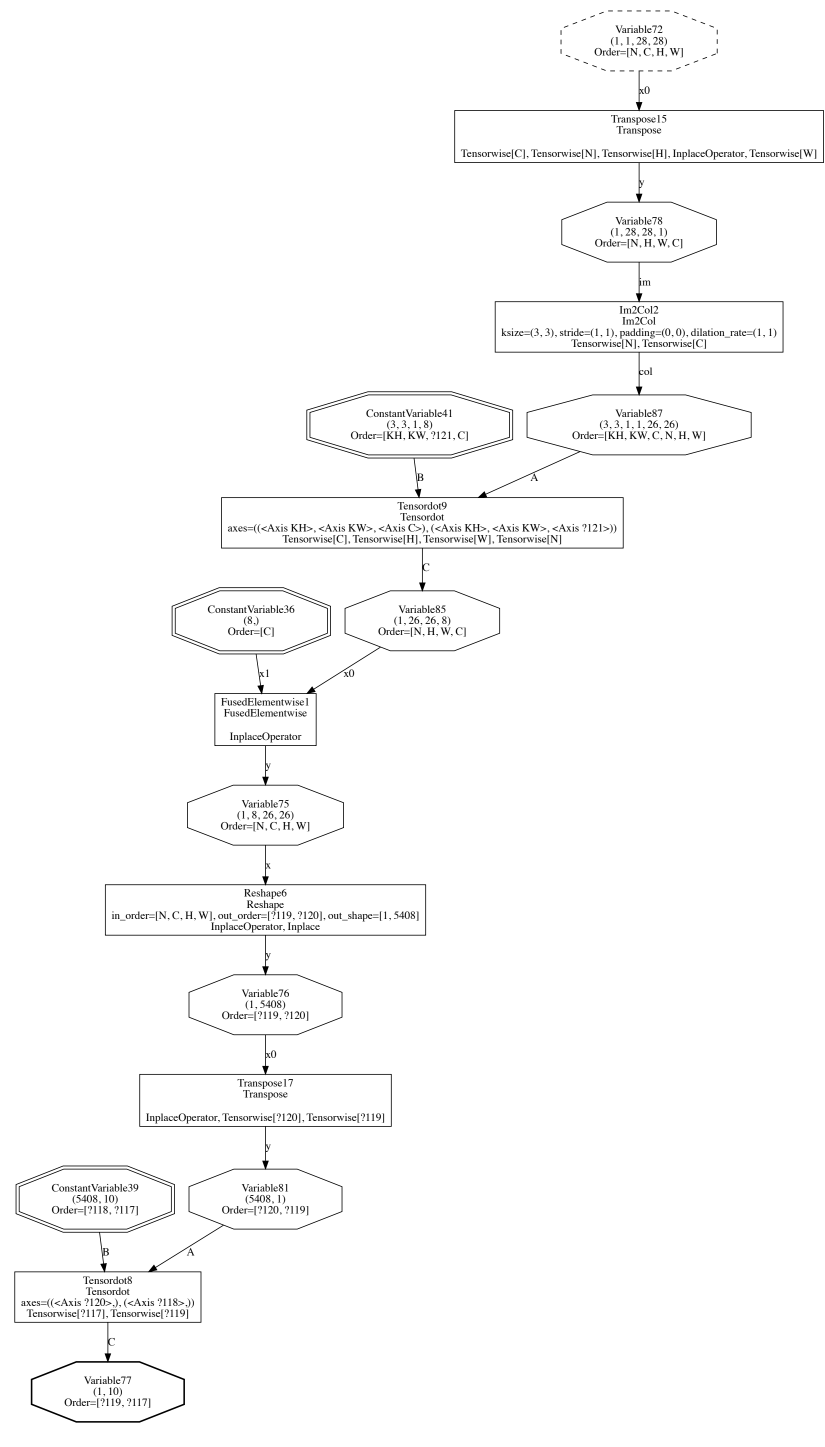


図 6.5 WebDNN の最適化された IR の例

には C++で実装された高効率な行列演算ライブラリ Eigen [3] を WebAssembly にコンパイルすることで，単純な実装と比較し実行速度を向上させている．WebAssembly は JavaScript から呼び出してコードを実行する．実行は JavaScript と同一スレッドで行われるため，メインスレッド上で演算処理を実行するとその間 GUI のイベント処理ができず，Web ブラウザがフリーズしたように見える．そのため，WebWorker API を用いて別スレッド上で非同期的に演算を行わせることにより，この問題を回避する．また，GPU を用いる API と異なり，複数種類の計算を JavaScript の介在なしに連続して行うことが可能であるため，WebAssembly 内ですべてのレイヤーの計算を行うソースコードを生成し，パフォーマンス上のオーバーヘッドを最小限としている．本来，WebWorker は複数のスレッドを生成することができ，マルチコア CPU による高速化も可能である．しかしながらメモリを共有することができないため，並列処理のためにはモデルをスレッド数分複製する必要が生じ，大幅なメモリ使用量の増加となってしまうため採用していない．複数の WebWorker 上でメモリを共有できる SharedArrayBuffer 機構が再度利用可能となれば，マルチスレッドによる高速化が可能となる．

なお，C++言語をコンパイルし WebAssembly を出力するツールとして Emscripten を用いているが，WebAssembly だけでなく，asm.js と呼ばれる JavaScript のサブセットとしてコンパイル結果を出力することができる．これは比較的古い Web ブラウザでも動作する．Windows 7，Windows 8 において公式にサポートされている Web ブラウザである Internet Explorer 11 に対応するため，WebAssembly に非対応のブラウザでは asm.js コードを読み込んで動作するように実装している．

Graph Descriptor は大きく分けて 3 つの要素からなる．オペレータの実行順序やテンソルの配置アドレスを示すメタデータ，オペレータに対応するプログラムコード，学習済みパラメータである．

WebDNN では計算グラフ中に現れるテンソルのサイズは原則として定数とみなしている．そのため，全てのテンソルを格納できる大きなバッファを事前に確保することにより，バッファの確保にかかるオーバーヘッドを削減することができる．各テンソルを書き込むアドレスは事前計算し，Graph Descriptor に格納する．なお，計算グラフ上の全てのテンソルが同時に必要なわけではなく，あるテンソルが不要となったのちに生成されるテンソルは，すでに不要となった領域を再利用することにより必要なバッファのサイズを削減することができる．計算グラフ全体の順序を考慮した上で割り当てアドレスを調整することにより，バッファのサイズをより小さくすることができる [89].

オペレータに対応するソースコードは，最も効率よく実行できるよう，各バックエンドごとに手動でテンプレートを記述し，文字列処理により定数を埋め込む方式をとった．以下は，Average Pooling を WebMetal に対して実装したコードである．%%で囲まれた部分については，入出力バッファへのアドレス，パラメータ等に置換される．

```
template = """
kernel void %%FUNC_NAME%%(device float * %%STATIC_BUFFER%%[[buffer(0)]],
                          device float * %%DYNAMIC_BUFFER%%[[buffer(1)]],
                          const device int * %%META_BUFFER%% [[buffer(2)]],
```

[3]http://eigen.tuxfamily.org/index.php?title=Main_Page

```
                          uint index[[thread_position_in_grid]],
                          uint num_threads[[threads_per_grid]])
{
    const device float *X = %%LOAD_BUFFER(average_pooling_2d_X)%%;
    device float *Y = %%LOAD_BUFFER(average_pooling_2d_Y)%%;
    const int N = %%LOAD_BUFFER(average_pooling_2d_N)%%;
    const int H1 = %%LOAD_BUFFER(average_pooling_2d_H1)%%;
    const int W1 = %%LOAD_BUFFER(average_pooling_2d_W1)%%;
    const int C = %%LOAD_BUFFER(average_pooling_2d_C)%%;
    const int H2 = %%LOAD_BUFFER(average_pooling_2d_H2)%%;
    const int W2 = %%LOAD_BUFFER(average_pooling_2d_W2)%%;

    const int KH = %%LOAD_BUFFER(average_pooling_2d_KH)%%;
    const int KW = %%LOAD_BUFFER(average_pooling_2d_KW)%%;
    const int SH = %%LOAD_BUFFER(average_pooling_2d_SH)%%;
    const int SW = %%LOAD_BUFFER(average_pooling_2d_SW)%%;
    const int PH = %%LOAD_BUFFER(average_pooling_2d_PH)%%;
    const int PW = %%LOAD_BUFFER(average_pooling_2d_PW)%%;

    for (int gid = index; gid < N * H2 * W2 * C; gid += num_threads) {
        const int c = gid % C;
        const int w2 = gid / C % W2;
        const int h2 = gid / C / W2 % H2;
        const int n = gid / C / W2 / H2;

        float v = 0;
        %%DIVIDER_INIT%%
        for (int kh = 0; kh < KH; kh++) {
            const int h1 = h2 * SH - PH + kh;
            if (h1 < 0 || h1 >= H1) continue;

            for (int kw = 0; kw < KW; kw++) {
                const int w1 = w2 * SW - PW + kw;
                if (w1 < 0 || w1 >= W1) continue;

                v += X[((n * H1 + h1) * W1 + w1) * C + c];
                %%DIVIDER_ADD%%
            }
        }
        v /= %%DIVIDER_GET%%;

        Y[gid] = v;
    }
}
```

```
"""

# using 1e-8 to avoid zero division in extreme case. 1.0+1e-8 == 1.0
# (in float precision)
statement_divide_without_padding = {
    False: {"%%DIVIDER_INIT%%": "",
            "%%DIVIDER_ADD%%": "",
            "%%DIVIDER_GET%%": "KH * KW"},
    True: {"%%DIVIDER_INIT%%": "float divider = 1e-8;",
           "%%DIVIDER_ADD%%": "divider += 1.0;",
           "%%DIVIDER_GET%%": "divider"}}
```

6.3.4 パラメータの圧縮

DNN モデルのパラメータ行列の容量は大きい．例えば，画像認識モデルとして広く用いられる ResNet50 [14] では，約 100MB になる．インターネット上でこのような容量のファイルの転送を行うには時間がかかる．第 5 章では学習データを用いた圧縮を提案し，これを WebDNN と組み合わせることは可能である一方，学習データへアクセスできない場合や研究成果を素早くデモとして公開したい場合には手間が大きい．そのため，WebDNN では各パラメータ要素を 8bit に量子化 [51] し，さらに zlib 形式により可逆圧縮する機構を搭載している．この機能により，ResNet50 であれば約 20MB まで圧縮可能である．モバイル回線環境では時間がかかる場合もあるが，一度 WiFi 環境でダウンロードされれば，Service Worker などの API によりキャッシュしておくことが可能である．

6.3.5 ランタイムライブラリ

Descriptor Runner は，Web ブラウザ上でロードされ，ここまでのステップで生成された Graph Descriptor をロードして実行するライブラリである．バックエンドの差異を抽象化し，シンプルなインターフェースで DNN の実行を行えるよう配慮している．

Web ブラウザ上で Descriptor Runner を利用する最小限のコード例を示す．

```
let runner;

async function init() {
  // (1) Initialize DescriptorRunner
  runner = await WebDNN.load('./model');
}

async function run() {
  // (2.1) Set input data
  runner.getInputViews()[0].set(loadImageData());
```

```
  // (2.2) Run DNN model
  await runner.run();

  // (2.3) Show result
  console.log(WebDNN.Math.argmax(
    runner.getOutputViews()[0].toActual()));
}
```

Web ブラウザにより対応している API が異なるため，異なる Graph Descriptor を利用する必要がある．HTTP アクセスするため，サーバ側で User Agent を判定する方法が考えられる．例えば iPhone SE，iOS での Safari ブラウザは"Mozilla/5.0 (iPhone; CPU iPhone OS 12_3_1 like Mac OS X) AppleWebKit/605.1.15 (KHTML, like Gecko) Version/12.1.1 Mobile/15E148 Safari/604.1"，Windows での Chrome ブラウザは"Mozilla/5.0 (Windows NT 10.0; Win64; x64) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/75.0.3770.100 Safari/537.36"という HTTP ヘッダを送信する．これを文字列処理することにより，ブラウザごとに提供するファイルを変えることが可能である．実際に，旧バージョンの Web ブラウザが対応していない API を補完する JavaScript プログラムを提供するサービス [4] ではこの手法が使われている．しかし，無料または安価で借りることができる Web サーバにおいて，ヘッダによりクライアントに送信するファイルを分岐することができるものは少ない．そのため，クライアント側に共通のプログラムを読み込ませ，その中で判定する方式を採用した．

`WebDNN.load()` において，まず実行環境を判定し，どのバックエンドを利用するか決定する．次に，そのバックエンドに対応した Graph Descriptor を動的にダウンロードする．サーバ側で User Agent を識別するのではなく，Web ブラウザ側で判定の上 `kernel_webmetal.bin`，`kernel_webassembly.bin` のようにダウンロードする Graph Descriptor のファイル名を変化させるため，Web サーバ側に特別な設定は不要である．`runner.getInputViews()` は，DNN への入力データを転送するインターフェースを返す．このインターフェースにデータを与えると，バックエンドに応じて，GPU へのデータ転送等が行われる．`runner.run()` により DNN の推論が実行される．`runner.getOutputViews()` により，DNN の出力データを CPU に転送し，JavaScript 上のオブジェクトとして取得するインターフェースを返す．

以上のように，バックエンドの差異は抽象化され，ユーザはアプリケーション固有の入出力を実装するだけでよい．

図 6.6 に，カメラから取得された画像を Neural Style Transfer モデル [86] でリアルタイムに加工するアプリケーションの実行例を示す．画面サイズの考慮は必要であるが，共通のコードでデスクトップ，スマートフォン両方で動作するアプリケーションが実現できる．

6.4 実験

WebDNN による DNN モデルの実行速度を評価した．比較対象として，Web ブラウザ上で DNN モデルを実行できるフレームワーク Keras.js [90] を用いた．Keras.js は WebGL により GPU を

[4] https://polyfill.io/v3/

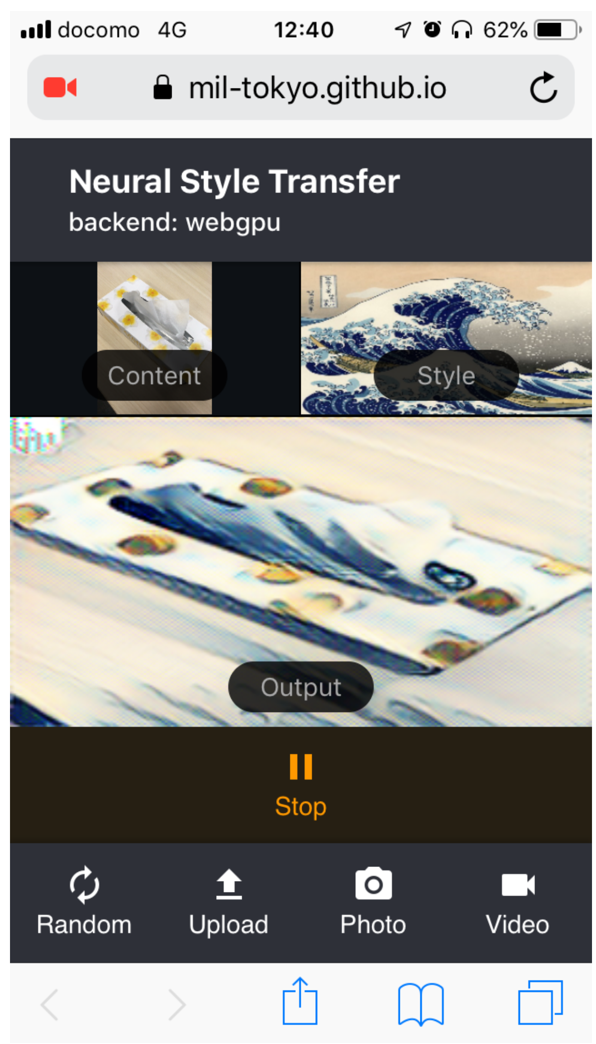


図 6.6 スマートフォン上で，カメラから取得された画像を Neural Style Transfer モデルでリアルタイムに加工するアプリケーションの実行例．

利用する．画像分類 DNN モデルとして広く用いられている VGG16 [16]，ResNet50 [14] を各フレームワークを用いて Web ブラウザ上で実行し，画像 1 枚あたりの処理時間を計測した．DNN の学習におけるベンチマークではバッチサイズを数十から数百に設定してスループットを評価するが，本評価ではバッチサイズは 1 とした．インタラクティブなアプリケーションを想定した場合，複数枚の画像を同時に入力することはまれであると考えられるためである．ハードウェアは MacBook Pro 2017，Intel Core i5 2.3GHz CPU，16GB Memory，Intel Iris Plus Graphics 640 GPU である．Web ブラウザは Safari，Chrome を利用した．Safari は WebMetal・WebGL1 に対応しているものの，WebGL2 に対応していない．Keras.js は WebGL2 向けの最適化が施されているため，WebGL2 に対応している Chrome での評価も行なっている．

実験結果を図 6.7 に示す．WebDNN において WebMetal バックエンドを利用しモデル最適化 (Optimize) を有効にした場合，Keras.js で WebGL2 を利用した場合と比較し VGG16 で 8.5

倍速，ResNet50 で 6.7 倍速の処理速度を実現できた．同一ハードウェア・同一バックエンドでも Web ブラウザごとに API の実装が異なるため，性能差が生じる．

また，スマートフォン iPhone 8 を用いた場合の実験結果を図 6.8 に示す．Web ブラウザは Safari である．iPhone へは Chrome ブラウザも提供されているが，内部で JavaScript を処理するエンジンは Safari と同じである．ResNet50 モデルが 215ms で動作しており，リアルタイムな物体認識アプリケーションとして十分な速度で処理することが可能と言える．

さらに，オフィスで用いられる市販のノート PC 環境としてもっとも標準的な Windows 10 と標準搭載の Web ブラウザ Internet Explorer 11 の組み合わせにおいても速度を計測した．ハードウェアは，Intel Core i5-3337U 1.80GHz CPU，Intel HD Graphics 4000 GPU である．Internet Explorer は WebMetal，WebAssembly には対応しておらず，asm.js コードを通常の JavaScript として解釈して動作する．ResNet50 の画像 1 枚あたりの処理時間を計測した結果，WebDNN から CPU を利用した場合では 9,495ms，Keras.js から GPU を利用した場合で 8,115ms，Keras.js から CPU を利用した場合で 27,070ms となった．端末の GPU，CPU の性能バランスによっては，WebDNN において CPU のみで計算を行わせた場合に Keras.js で GPU を用いた場合と同等のパフォーマンスが得られる場合も存在するといえる．

いずれのモデル，バックエンド，演算デバイス条件においても，WebDNN は既存フレームワークより優れた実行速度を得られた．この結果は，WebDNN が WebMetal へ対応したことおよび，Web ブラウザ上での DNN の推論フェーズに特化した最適化戦略によるものと考えられる．

Intel Iris Plus Graphics 640 GPU の性能理論値は単精度浮動小数点演算において 729GFLOPS である．一方，VGG16 モデルの実行において畳み込み層・全結合層の演算回数の合計は 3.0×10^{10} であり (積和演算は 2 個の計算とみなす)，所要時間 0.31 秒と比較すると実効速度は 97GFLOPS となっている．なお，最適化において畳み込み層・全結合層の計算量自体を削減する処理は含まれていない．この差は，CPU・GPU 間メモリ転送，行列積以外のレイヤーの処理のほか，行列積内部のメモリアクセスにかかる時間が考えられる．

計算量が大きい行列積単体の速度を測定するため，入出力行列の数・サイズを固定し，行列積のみを連続して行う DNN モデルを構築し，行列積の回数と所要時間を比較した．512×512 の行列 2 つの積を 1 回計算した場合の平均所要時間が 14.23ms，11 回計算した場合の平均所要時間が 34.16ms となった．これらは CPU・GPU 間の転送データ量は同じであり，差の 19.93ms は行列積 10 回の計算にかかる時間だとみなすことができる．ここから，行列積単体の実行速度は 135GFLOPS と見積もられる．ハードウェアの性能理論値は 729GFLOPS であり，まだ開きがある．Lavin らは，NVIDIA Maxwell GPU における畳み込み演算の最適化を行い，ハードウェア性能の 96.3%の実行速度を得ることに成功した [92]．この研究では，NVIDIA GPU 全般に用いることができる CUDA C 言語ではなく，NVIDIA Maxwell アーキテクチャ専用のアセンブラ言語を用いて最高性能が得られるようチューニングを行っている．Web ブラウザ上の API として，特定の GPU アーキテクチャ専用のアセンブラ言語を利用できるようにすることはハードウェア依存性が極めて高くなる点およびセキュリティ上の観点から困難である．また高級言語の範囲でデバイスに応じたチューニングを行うことについても，JavaScript 上での時

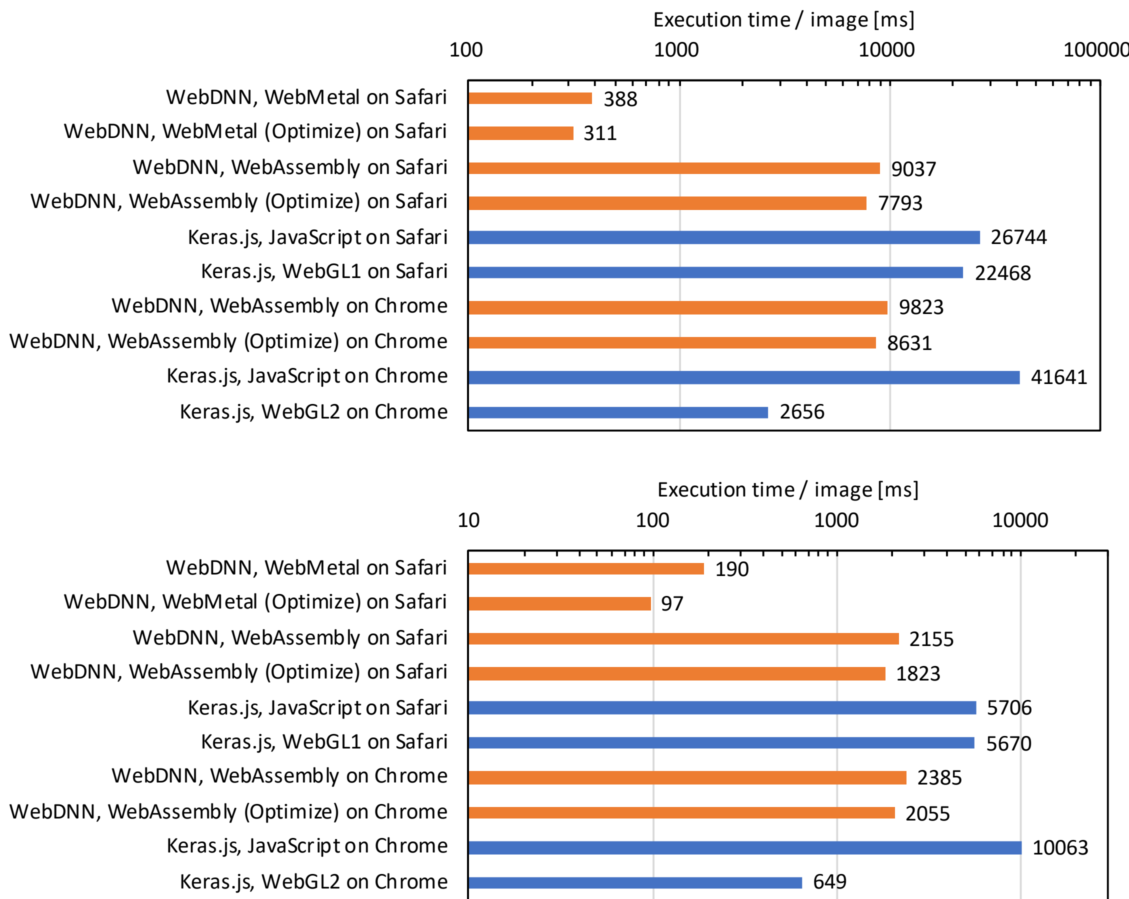


図 6.7 WebDNN のベンチマーク (MacBook, 上: VGG16 モデル, 下: ResNet50 モデル). 画像分類 DNN における画像 1 枚あたりの処理時間を示す. 短い方が良い.

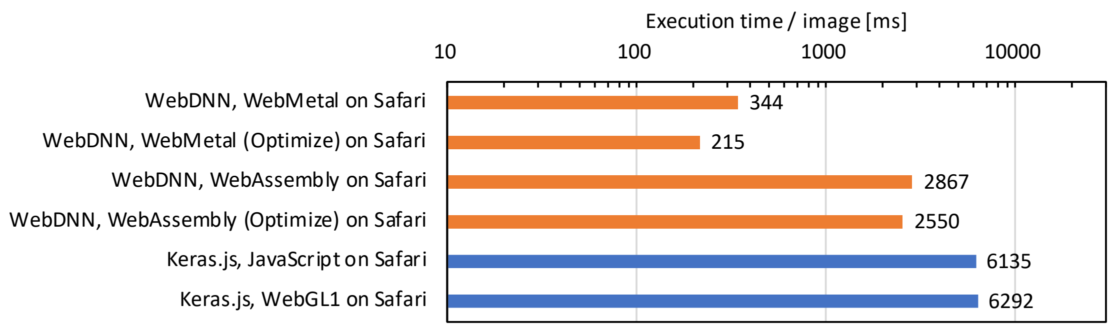


図 6.8 WebDNN のベンチマーク (iPhone, ResNet50 モデル)

間測定の解像度を恣意的に下げる [5] などの変更が行われている実情であり，この方向性は限度

[5] https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Date/now#Reduced_time_precision

がある．このような変更は広告のためにユーザのトラッキングを行う手法として端末の個体識別を試みる Fingerprinting[93][94] への耐性を高めるためになされており，今後もデバイスを特定するような情報を得ることが難しいようなプライバシー重視の API 設計がなされるものと思われる．現実的な方策として，行列積のような典型的かつ高負荷な演算に対し，ハードウェアごとにチューニングされた演算カーネルを OS または Web ブラウザに搭載し，それを呼び出す API を提供するというアーキテクチャが考えられる．さらに，GPU だけでなく AI に特化した NPU (Neural network Processing Unit) がスマートフォンに搭載される事例 [6] もあり，これらの機能を呼び出す API を提供することも有用であると考えられる．一方で，計算量の比較的少ない活性化関数においては ReLU 以降，Leaky ReLU，Parametric ReLU [95]，Scaled Exponential Linear Units [96] 等が開発され続けており，DNN のあらゆる部分を規格化・専用ハードウェア化することはできない．このような部分の処理を実装するためには GPGPU で任意の計算を行える API が引き続き有用であり，これを効率よく実装可能な WebMetal のような API の普及が必要である．

6.5 オープンソースソフトウェアとしての WebDNN

WebDNN はアルゴリズムの具体的な実装および実験で使用したモデルの変換を行うだけでなく，第三者が独自のモデルを用いて実用的なソフトウェア開発に利用できるよう整備し，Github 上でオープンソースソフトウェアとして公開・サポートを行った [7]．

利用例として，各深層学習フレームワークにおいて MNIST を識別する CNN の学習・変換を行うもの，学習済みの画像分類モデル ResNet, SqueezeNet を変換するもの，Neural Style Transfer モデルを変換し Web カメラから取り込んだ画像をリアルタイムに変換するもの，JavaScript から反復的に Descriptor Runner を実行して繰り返しが必要な自然言語生成モデルを実行するもののほか，サーバ設定上の課題を解消するために比較的サイズが大きいモデルの重みファイルを別のサーバに設置するものなど，モデル変換とアプリケーションとしての配布手段の両側面について事例を提供し各種利用に備えた．

アプリケーション開発者が大きなアプリケーションに WebDNN を組み込みやすくするため，各言語標準のパッケージマネージャへ対応した．Python で記述された Graph Transpiler は pip により，JavaScript で記述された Descriptor Runner は npm によりインストールが可能である．また，インストール方法・基本的な利用例についてドキュメントを備えた．

WebDNN を利用していると報告があったアプリケーションの事例を解説する．MakeGirls.moe [8] [97] は，Generative Adversarial Network を用いてアニメのキャラクター風の画像を自動生成する．キャラクターのパーツの属性を入力として様々な画像を生成できるモデルとなっており，ユーザによりインタラクティブな画像生成を行えるサイトにおいて WebDNN が利用されている．AZ.js [9] は，DNN による強力な思考エンジンを搭載した囲碁プログラムである．MorphCast

[6] https://pc.watch.impress.co.jp/docs/news/event/1078883.html
[7] https://github.com/mil-tokyo/webdnn/
[8] https://make.girls.moe/#/
[9] https://new3rs.github.io/AZ.js/index.html

表 6.1 Github に寄せられた WebDNN に関するバグ・質問の集計

分類	件数
レイヤー未実装による動作不可	22
クラッシュ	19
OS・ライブラリバージョンによる問題	13
使用法の誤り	9
動作結果の誤り	7
未対応深層学習フレームワークへの対応依頼	6
ドキュメントの修正	3
利便性向上提案	1
その他	6

[10] は，カメラから入力されたユーザの映像から感情を認識するライブラリであり，WebDNN を内包している．nsfw-webdnn [11] は，不適切な画像をフィルタリングする Open NSFW (Not Suitable for Work) モデルのデモンストレーションを行う．これらのプロジェクトはいずれも一般的な Web ブラウザですぐ利用できるものであり，ユーザにとって利便性が高い形で DNN モデルによる成果を提供することに WebDNN が役立っている．なお，これらはいずれも第三者により自発的に利用がなされたものであり，WebDNN プロジェクトとして提携を行っているものではない．

Github 上では利用者からのバグ報告，利用法の質問が多数なされた．これらを分類した結果は表 6.1 のようになる．

実行速度の向上を要望する意見は見られなかった．動作不具合を除くと，様々なレイヤーを持った DNN モデルを動作させることやさらに多くの深層学習フレームワークをサポートする点に需要が大きいと考えられる．

[10]https://www.morphcast.com/
[11]https://github.com/zaghaghi/nsfw-webdnn

Chapter 7

結論と今後の展望

7.1 結論

本研究では，ヘテロジニアス計算環境において Deep Neural Network モデルの学習および推論を効率的に実施することを目指し，多くのデバイスを統一的に扱うことができる Web ブラウザをプラットフォームとした効率的な計算基盤の構築を行った．

DNN の計算基盤として，DNN の実行制御，行列計算，通信の各機構が必要であり，各機構における課題を解決することによりヘテロジニアス計算環境における計算を高速化できる．計算資源として潜在的に大きな計算能力を持つスマートフォン等の情報端末を計算に用いることは計算負荷の大きい DNN の活用範囲を広げるために重要であり，端末上で DNN の計算を行うにあたり生じる課題を解決することが本研究の主な課題であった．消費者が持つ様々な機種・動作環境の端末を統一的に DNN の計算資源として利用するためには，オペレーティングシステム，計算性能，通信環境の差異の吸収および，ソフトウェアのインストールの負担の解消が必要となる．Web ブラウザはアプリケーションをインストールなしに実行できるクロスプラットフォームな環境であり，オペレーティングシステムの差異やソフトウェアのインストールにおける課題を大きく軽減する一方，科学技術計算へ応用するためには計算速度上の課題を解決することが重要である．現在のところ，端末の性能を効率的に活用するためのアクセラレータ API については環境依存性が強く，この抽象化は Web ブラウザというプラットフォームのメリットを活用するために必要不可欠なものである．

第 1 章では，DNN 計算に一般的に用いられるクラウド環境ではなくスマートフォン端末や IoT デバイスなどヘテロジニアス計算環境で科学技術計算を行う意義を述べ，端末上でのアプリケーション実行プラットフォームとして Web ブラウザが良い性質を持つことを示した．第 2 章では計算対象となる Deep Neural Network の概要と性質について述べた．第 3 章ではヘテロジニアス計算環境における計算基盤の必要条件および，その具体的な実装環境である Web ブラウザにおいて，処理を実装および高速化するために用いることができる機能について述べた．そして第 4 章では DNN の学習，第 5・6 章では DNN の推論にフォーカスを当て，実現方法を提案した．

7.1.1 DNNの分散学習 (第4章)

第4章では，深層学習を複数の端末により分散計算させる課題に取り組んだ．深層学習基盤における，行列計算でのアクセラレータの利用，通信での学習時の計算ノード差へ対応する．深層学習に必要な行列演算の種類が多く，複数のアクセラレータAPIに対応したコードを手動で記述することが大きな手間となるため，Python言語のサブセットにより記述した単一の行列演算コードを各アクセラレータAPI用のコードへと変換する機構により，広く普及しているAPIへの対応とともに，新しいAPIへの対応を少ない記述量で実現した．新しいAPIへの対応によって，従来のAPIに対して手動でチューニングされたシェーダコードよりも高速な計算を実現した．分散計算によるDNNの学習では計算ノード間の同期が必要であり，性能の異なる複数の端末を有効に活用するためには，その差異を明示的に吸収するアルゴリズムが必要となる．従来の分散計算で一般的である単純なmapreduce方式をDNNに応用する際の非効率な点を指摘し，端末の速度差を考慮した動的にバッチサイズおよび学習率を調整する機構を提案し，様々な機種の端末を用いた効率的な学習を実現した．

7.1.2 学習済みDNNの圧縮 (第5章)

アプリケーションがインストール不要であることはユーザにとって操作が簡略化される反面，DNNモデルのダウンロードの間ユーザを待たせてしまうデメリットが生じる．通信環境によりダウンロードにかかる時間は大きく変動するため，精度と待ち時間のバランスをとる手段が必要である．本研究では，DNNを利用したWebアプリケーションにおけるモデルのロードを最適化するための圧縮手法を提案した．深層学習基盤における，推論時の通信環境差に対応する．提案手法では学習済みモデルを圧縮し，ベースモデルを生成するとともに，圧縮過程で失われた情報を補完する複数のパッチを生成する．アプリケーションはベースモデルを読み込んだ時点でDNNの計算を行えるため，実行開始タイミングを早めることが可能となる．パッチを全て読み込み終えると，圧縮前のモデルと同等の精度で計算を行えるようになる．データ量に対する精度を高めるため，学習済みモデルを初期値とし，これが持つ情報を少ないサイズオーバーヘッドでベースモデルとパッチに分離するための量子化ビット数削減とチャンネル単位のプルーニングを提案した．また，精度を維持するためにバッチ正規化レイヤーのみをファインチューニングするスケールファインチューニングを提案した．ビット幅の小さいパッチデータを効率的に符号化するため，従来よく用いられていたHuffman codeではなくrange encoderの利用を提案した．実験では画像分類モデルでこれらの手法の有効性を確認し，また画像生成モデルへの応用例を示した．

7.1.3 学習済みDNNの推論高速化 (第6章)

第5章では，学習済みDNNモデルを単一端末上で高速に実行する課題に取り組んだ．深層学習基盤における，DNN実行制御での多様なモデルへの対応・最適化，行列計算でのアクセラレータの利用に対応する．様々な深層学習フレームワークにより生成されたDNNモデルを統一的に処理するための中間表現を開発し，中間表現上で推論に特化した最適化を施した．Web

ブラウザ上で GPU を効率的に利用できる新しい API である WebMetal を初めて科学技術計算へ応用した．WebMetal が利用できない環境においても JavaScript による記述よりも高速化するため，CPU 上での効率的な計算が可能となる WebAssembly API を利用する実装を行った．その結果，スマートフォン上で著名な画像分類モデルをインタラクティブなアプリケーションのために十分な速度で動作させることに成功した．速度の追求だけでなく，アプリケーション開発者が利用しやすいフレームワークとして整備を行い，オープンソースソフトウェアとして公開した．その結果，複数のアプリケーションに実際に利用され，また我々の想定以上に様々なモデルを動作させたいとの需要があることがうかがえた．

本研究全体を通して得られた知見を挙げる．

- ヘテロジニアス計算環境におけるハードウェアアクセラレーションの利用においては，DNN のレイヤーレベルでの抽象化または GPU シェーダ言語の動的生成の枠組みによりアプリケーション開発を容易にできることを示した．GPU シェーダ言語の動的生成においては，DNN に必要な多次元テンソルの処理に特化し構文を絞り込むことで可読性の高い記述を行うことができた．このシステムは C 言語ベースのシェーダ言語であれば Web ブラウザ上の API でなくても応用でき，クラウド環境で広く用いられている NVIDIA 社の CUDA 環境において新しい種類のレイヤーを実装することへ応用可能である．
- DNN の学習において計算資源の台数や性能が動的に変化する場合には，バッチサイズを変動させ，同時に学習率も変更することによって効率よく学習を進められることを示した．消費者向けの端末だけでなく，異なる性能のサーバ群，IoT デバイスの余剰計算能力を用いた分散計算への応用が考えられる．
- DNN の圧縮において情報を削除する方法および符号化の方法を工夫することにより，回線速度が予測できない環境においても精度と読み込み時間の両立を行えることを示した．連続的にモデルの精度と情報量が変化する特性から，演算のビット幅を低下と引き換えに高速な演算が可能なハードウェアが利用可能な環境では，システムの負荷状況に応じて精度とスループットを制御するような応用が期待できる．
- 深層学習フレームワーク間には様々な差異があるが，計算グラフのレベルで抽象化することによりモデルを統一的に扱うことが可能となる．さらに，DNN における各計算の性質を属性として表現して最適化に活用することにより，推論の速度が向上することを示した．数学的な計算量を低下させることや GPU の呼び出し回数を削減することはハードウェアによらず普遍的な高速化手段となる．実際，2017 年 10 月に Amazon により DNN の中間言語に対する最適化を行い各種ハードウェアでの高速実行を目指す NNVM コンパイラが発表されている．また，2017 年 9 月に Microsoft と Facebook により深層学習フレームワーク間でのモデル相互運用のための共通規格 ONNX が発表され，2019 年現在多くの深層学習フレームワークが対応している．

本研究で提案したすべてのソフトウェアシステムを統合することにより，クラウドサービス等による高性能なサーバシステムに依存せず，深層学習を用いたアプリケーションを開発，提供する枠組みが構築可能となる．

7.2 今後の課題

ブラウザコンピューティング環境の発展

本研究においては，Web ブラウザに搭載された API は所与のものとした一方，その上で動作する科学技術計算に関するアルゴリズムのほぼ全てを独自に実装する必要があった．今後のブラウザコンピューティングの発展のためには，以下のような方向性で環境を改良していく必要があると考えられる．

DNN を効率よく実行するという観点において，ハードウェアアクセラレーションのための API が今後持つべき性質を挙げる．DNN において計算量の大部分を占めるのは行列積である．メモリアクセスパターンの考慮が重要であり，効率的な計算にはキャッシュサイズ等低レイヤーの部分の差異を考慮することが必要となるが，ハードウェアの詳細を Web アプリケーションから取得可能とすることはプライバシー面から難しい情勢である．行列積は非常に典型的な演算であり，特化された実装が各ハードウェアごとに存在している．これを呼び出す機構を粗粒度の API として提供することが望ましい方向性である．一方，DNN の最適化を改善する活性化関数や，新たなタスクを実現するメカニズムが日々発表されるため，行列の要素単位で任意の演算を記述可能な GPGPU 指向の API の普及も引き続き必要となる．

より具体的な API 仕様として，以下のような事項が DNN の実装上重要である．

- DNN の学習においては，同一の実装であってもバッチサイズを極力大きくすることがハードウェア性能の活用につながる．バッチサイズの制約はメモリサイズからくるものであり，この制限を取得できる機能が必要である．
- 大きなモデル・バッチサイズの DNN モデルの処理においては，一つの行列あたりのメモリ容量も大きく取れる必要がある．WebGL 規格に準拠したモバイルデバイスの多くで 4096×4096 要素が最大となる一方で，典型的な DNN モデルでもこれを超える容量を必要とする場合があり実装の支障となる．同時に，行列内のインデックスを表す整数値を操作するにあたり，32 ビット浮動小数点数を整数として利用した場合の 23 ビット精度では不足がある．整数として 32 ビット精度が保証される環境が望ましい．
- CPU 上で計算を行う API として，マルチスレッドの考慮がなされるべきである．スマートフォンにおいてもマルチコア CPU が普及しており，数倍の高速化が見込める．

Web ブラウザにおけるハードウェアアクセラレーション技術は変化が激しい時期にある．新規格の提案や実装が頻繁に行われ，逆にセキュリティ上の理由で廃止される場合もある．Python 言語においては numpy ライブラリが行列を扱う科学技術計算アプリケーションの共通基盤として確立し，ライブラリを更新すればアプリケーションから新しい CPU 命令を用いた演算が自動的に活用できる状況にある．さらには numpy と互換性のある cupy ライブラリを用いれば GPU 上での計算も行うことができ，ハードウェアアクセラレーション技術にかかわらず上位レベルのプログラミングに集中することができる．Web ブラウザ上でも科学技術計算に対して適切な粒度での抽象化を行ったエコシステムを普及させることにより，継続的に速度向上を実現することができ科学技術計算の発展につながる．

分散計算におけるネットワーク環境の改良

深層学習の分散計算においては，各計算ノードの性能向上とともに通信がボトルネックになってくる．科学技術計算向けのコンピュータクラスタにおいては効率的なネットワークトポロジが構成されているが，インターネット上では様々な通信環境の計算ノードが混在している．全計算ノードの同期を単一のサーバとの通信によって行うのではなく，エッジサーバ等との協調によってネットワークトポロジを考慮し階層的に行うことが有用と考えられる．このようなシステムの実現には，行列データを転送・加工する処理に関するプロトコルの標準化が必要となる．

通信環境の改善と DNN 学習のアルゴリズムの発展により，世界中に散らばる計算資源を取りまとめて大規模なモデルが学習できるようになれば，囲碁の新戦法を示した AlphaGo [98] のようなブレークスルーを実社会の様々な分野において引き起こすことが可能になると考えられる．

References

[1] 総務省, “平成 30 年版　情報通信白書のポイント,” 2018. http://www.soumu.go.jp/johotsusintokei/whitepaper/ja/h30/html/nd252110.html.

[2] A. Janowczyk and A. Madabhushi, “Deep learning for digital pathology image analysis: A comprehensive tutorial with selected use cases,” *Journal of pathology informatics*, vol. 7, 2016.

[3] 齋藤彰, “デジタル・パソロジー 現状とその将来,” 東京医科大学雑誌, vol. 74, no. 4, pp. 396–408, 2016.

[4] B. Chu, V. Madhavan, O. Beijbom, J. Hoffman, and T. Darrell, “Best practices for fine-tuning visual classifiers to new domains,” in *ECCV Workshop*, 2016.

[5] Y. Ganin, E. Ustinova, H. Ajakan, P. Germain, H. Larochelle, F. Laviolette, M. March, and V. Lempitsky, “Domain-adversarial training of neural networks,” *Journal of Machine Learning Research*, vol. 17, no. 59, pp. 1–35, 2016.

[6] Y. LeCun, L. Bottou, Y. Bengio, and P. Haffner, “Gradient-based learning applied to document recognition,” *Proceedings of the IEEE*, vol. 86, 1998.

[7] D. G. Lowe, “Distinctive image features from scale-invariant keypoints,” *International journal of computer vision*, vol. 60, no. 2, pp. 91–110, 2004.

[8] A. Krizhevsky, I. Sutskever, and G. E. Hinton, “Imagenet classification with deep convolutional neural networks,” in *NIPS*, 2012.

[9] J. Deng, W. Dong, R. Socher, L.-J. Li, K. Li, and L. Fei-Fei, “ImageNet: A Large-Scale Hierarchical Image Database,” in *CVPR*, 2009.

[10] D.-A. Clevert, T. Unterthiner, and S. Hochreiter, “Fast and Accurate Deep Network Learning by Exponential Linear Units (ELUs),” *arXiv preprint arXiv:1511.07289*, 2015.

[11] S. Ioffe and C. Szegedy, “Batch normalization: Accelerating deep network training by reducing internal covariate shift,” in *ICML*, 2015.

[12] G. Hinton, N. Srivastava, and K. Swersky, “Neural networks for machine learning, lecture 6a, overview of mini-batch gradient descent,” 2012. http://www.cs.toronto.edu/~tijmen/csc321/slides/lecture_slides_lec6.pdf.

[13] D. P. Kingma and J. Ba, “Adam: A method for stochastic optimization,” in *ICLR*, 2015.

[14] K. He, X. Zhang, S. Ren, and J. Sun, “Deep residual learning for image recognition,” in *CVPR*, 2015.

[15] P. Goyal, P. Dollár, R. B. Girshick, P. Noordhuis, L. Wesolowski, A. Kyrola, A. Tulloch, Y. Jia, and K. He, “Accurate, large minibatch SGD: training imagenet in 1 hour,” *arXiv preprint arXiv:1706.02677*, 2017.

[16] K. Simonyan and A. Zisserman, "Very deep convolutional networks for large-scale image recognition," in *ICLR*, 2014.

[17] Y. Jia, E. Shelhamer, J. Donahue, S. Karayev, J. Long, R. Girshick, S. Guadarrama, and T. Darrell, "Caffe: Convolutional architecture for fast feature embedding," in *ACMMM*, 2014.

[18] S. Tokui, K. Oono, S. Hido, and J. Clayton, "Chainer: a next-generation open source framework for deep learning," in *LearningSys, NIPS Workshop*, 2015.

[19] M. Abadi, A. Agarwal, P. Barham, E. Brevdo, Z. Chen, C. Citro, G. S. Corrado, A. Davis, J. Dean, M. Devin, S. Ghemawat, I. J. Goodfellow, A. Harp, G. Irving, M. Isard, Y. Jia, R. Józefowicz, L. Kaiser, M. Kudlur, J. Levenberg, D. Mané, R. Monga, S. Moore, D. G. Murray, C. Olah, M. Schuster, J. Shlens, B. Steiner, I. Sutskever, K. Talwar, P. A. Tucker, V. Vanhoucke, V. Vasudevan, F. B. Viégas, O. Vinyals, P. Warden, M. Wattenberg, M. Wicke, Y. Yu, and X. Zheng, "Tensorflow: Large-scale machine learning on heterogeneous distributed systems," *arXiv preprint arXiv:1603.04467*, 2016.

[20] 吉村俊哉, "Iot 時代に注目される「エッジコンピューティング」," 2018. https://www.nttdata.com/jp/ja/data-insight/2018/1122/.

[21] K. Kanai, K. Imagane, and J. Katto, "[invited paper] overview of multimedia mobile edge computing," *ITE Transactions on Media Technology and Applications*, vol. 6, no. 1, pp. 46–52, 2018.

[22] D. P. Anderson, J. Cobb, E. Korpela, M. Lebofsky, and D. Werthimer, "SETI@home: an experiment in public-resource computing," *Communications of the ACM*, vol. 45, pp. 56–61, 2002.

[23] 高田雅美，柴山智子，渡辺知恵美，庄野逸，城和貴, "i アプリを用いた数値計算の可能性," 情報処理学会論文誌：数理モデル化と応用, vol. 46, no. SIG 2(TOM 11), pp. 47–55, 2005.

[24] T. Fabisiak and A. Danilecki, "Browser-based harnessing of voluntary computational power," *Foundations of Computing and Decision Sciences*, vol. 42, no. 1, pp. 3 – 42, 2017.

[25] E. Lavoie, L. J. Hendren, F. Desprez, and M. Correia, "Pando: a volunteer computing platform for the web," *arXiv preprint arXiv:1803.08426*, 2018.

[26] 高木省吾，渡邊寛，福士将，天野憲樹，舩曳信生，中西透, "Web ブラウザを用いたボランティアコンピューティングプラットフォームの提案," ハイパフォーマンスコンピューティング研究報告, vol. 2014, pp. 1–8, feb 2014.

[27] J. Konecný, H. B. McMahan, F. X. Yu, P. Richtárik, A. T. Suresh, and D. Bacon, "Federated learning: Strategies for improving communication efficiency," *arXiv preprint arXiv:1610.05492*, 2016.

[28] F. Konishi, M. Ishii, S. Ohki, R. Umestu, and A. Konagaya, "Rabc: A conceptual design of pervasive infrastructure for browser computing based on ajax technologies," in *Seventh IEEE International Symposium on Cluster Computing and the Grid (CCGrid '07)*, pp. 661–672, May 2007.

[29] E. Meeds, R. Hendriks, S. al Faraby, M. Bruntink, and M. Welling, "MLitB: Machine Learning in the Browser," *arXiv preprint arXiv:1412.2432*, 2014.

[30] 川崎寛文，丸山広，高嶋章雄，中村太一, "Web ブラウザを用いたグリッドコンピューティングフレームワークの研究," in 情報処理学会第 *74* 回全国大会講演論文集, no. 1, pp. 165–166, mar 2012.

[31] A. Baratloo, M. Karaul, Z. Kedem, and P. Wijckoff, “Charlotte: Metacomputing on the web,” *Future Generation Computer Systems*, vol. 15, no. 5, pp. 559 – 570, 1999.

[32] Oracle, “Java client roadmap update - an oracle white paper march 2018,” 2018. http://www.oracle.com/technetwork/java/javase/javaclientroadmapupdate2018mar-4414431.pdf.

[33] Node.js Foundation, “Node.js,” 2009. https://nodejs.org/en/.

[34] Microsoft, “Typescript,” 2012. https://www.typescriptlang.org/.

[35] D. Jackson, “Webgpu api proposal,” 2017. https://webkit.org/wp-content/uploads/webgpu-api-proposal.html.

[36] GPU for the Web Community Group, “Gpu for the web community group,” 2017. https://www.w3.org/community/gpu/.

[37] T. Schuster, “Sharedarraybuffer and timing attacks (meltdown and spectre),” 2018. https://github.com/tc39/security/issues/3.

[38] syzer, “Js-spark,” 2012. https://github.com/syzer/JS-Spark.

[39] K. Miura and T. Harada, “Implementation of a practical distributed calculation system with browsers and javascript, and application to distributed deep learning,” *arXiv preprint arXiv:1503.05743*, 2015.

[40] T. Akiba, K. Fukuda, and S. Suzuki, “ChainerMN: Scalable Distributed Deep Learning Framework,” in *ML Systems, NIPS Workshop*, 2017.

[41] A. Karpathy, “Convnetjs,” 2014. https://cs.stanford.edu/people/karpathy/convnetjs/.

[42] M. Hidaka, K. Miura, and T. Harada, “Development of javascript-based deep learning platform and application to distributed training,” in *ICLR Workshop*, 2017.

[43] W. Flinn, “weblas,” 2016. https://github.com/waylonflinn/weblas.

[44] gpu.js Team, “gpu.js,” 2017. https://github.com/gpujs/gpu.js.

[45] J. Mao, X. Chen, K. W. Nixon, C. Krieger, and Y. Chen, “Modnn: Local distributed mobile computing system for deep neural network,” in *Design, Automation Test in Europe Conference Exhibition (DATE), 2017*, pp. 1396–1401, March 2017.

[46] E. Yang, S. Kim, T. Kim, M. Jeon, S. Park, and C. Youn, “An adaptive batch-orchestration algorithm for the heterogeneous gpu cluster environment in distributed deep learning system,” in *2018 IEEE International Conference on Big Data and Smart Computing (BigComp)*, pp. 725–728, Jan 2018.

[47] C. Chen, Q. Weng, W. Wang, B. Li, and B. Li, “Fast distributed deep learning via worker-adaptive batch sizing,” *arXiv preprint arXiv:1806.02508*, 2018.

[48] J. Dean and L. A. Barroso, “The Tail at Scale,” *Communications of the ACM*, vol. 56, no. 2, pp. 74–80, 2013.

[49] J. Chen, R. Monga, S. Bengio, and R. Józefowicz, “Revisiting distributed synchronous SGD,” *arXiv preprint arXiv:1604.00981*, 2016.

[50] E. Ilyushin and D. Namiot, “On source-to-source compilers,” *International Journal of Open Information Technologies*, vol. 4, no. 5, 2016.

[51] T. Dettmers, "8-bit approximations for parallelism in deep learning," in *ICLR*, 2016.

[52] L. Luo, J. Nelson, L. Ceze, A. Phanishayee, and A. Krishnamurthy, "Parameter hub: A rack-scale parameter server for distributed deep neural network training," in *Proceedings of the ACM Symposium on Cloud Computing*, SoCC '18, (New York, NY, USA), pp. 41–54, ACM, 2018.

[53] T. Akiba, S. Suzuki, and K. Fukuda, "Extremely large minibatch SGD: training resnet-50 on imagenet in 15 minutes," *arXiv preprint arXiv:1711.04325*, 2017.

[54] S. L. Smith, P. Kindermans, and Q. V. Le, "Don't decay the learning rate, increase the batch size," in *ICLR*, 2017.

[55] A. Krizhevsky, "Learning multiple layers of features from tiny images," 2009. Technical Report TR-2009, University of Toronto, Toronto. https://www.cs.toronto.edu/~kriz/learning-features-2009-TR.pdf.

[56] J. Coglan, "Sylvester," 2012. http://sylvester.jcoglan.com/.

[57] M. Sandler, A. G. Howard, M. Zhu, A. Zhmoginov, and L. Chen, "Inverted residuals and linear bottlenecks: Mobile networks for classification, detection and segmentation," *arXiv preprint arXiv:1801.04381*, 2018.

[58] Ookla, "Speedtest global index," 2019. https://www.speedtest.net/global-index.

[59] Akamai, "A 100-millisecond delay in website load time can hurt conversion rates by 7 percent," 2017. https://www.akamai.com/uk/en/about/news/press/2017-press/akamai-releases-spring-2017-state-of-online-retail-performance-report.jsp.

[60] Y. Cheng, D. Wang, P. Zhou, and T. Zhang, "A survey of model compression and acceleration for deep neural networks," *arXiv preprint arXiv:1710.09282*, 2017.

[61] Y. L. Cun, J. S. Denker, and S. A. Solla, "Optimal brain damage," in *NIPS*, 1990.

[62] S. Han, J. Pool, J. Tran, and W. Dally, "Learning both weights and connections for efficient neural network," in *NIPS*, 2015.

[63] R. Yu, A. Li, C.-F. Chen, J.-H. Lai, V. I. Morariu, X. Han, M. Gao, C.-Y. Lin, and L. S. Davis, "Nisp: Pruning networks using neuron importance score propagation," in *CVPR*, 2018.

[64] H. Li, A. Kadav, I. Durdanovic, H. Samet, and H. P. Graf, "Pruning filters for efficient convnets," in *ICLR*, 2017.

[65] H. Hu, R. Peng, Y. Tai, and C. Tang, "Network trimming: A data-driven neuron pruning approach towards efficient deep architectures," *arXiv preprint arXiv:1607.03250*, 2016.

[66] Y. He, X. Zhang, and J. Sun, "Channel pruning for accelerating very deep neural networks," in *ICCV*, 2017.

[67] C. Chen, F. Tung, N. Vedula, and G. Mori, "Constraint-aware deep neural network compression," in *ECCV*, 2018.

[68] Y. He, J. Lin, Z. Liu, H. Wang, L.-J. Li, and S. Han, "Amc: Automl for model compression and acceleration on mobile devices," in *ECCV*, 2018.

[69] W. Sung, S. Shin, and K. Hwang, "Resiliency of deep neural networks under quantization," *arXiv preprint arXiv:1511.06488*, 2015.

[70] D. Miyashita, E. H. Lee, and B. Murmann, "Convolutional neural networks using logarithmic data representation," *arXiv preprint arXiv:1603.01025*, 2016.

[71] E. Park, J. Ahn, and S. Yoo, "Weighted-entropy-based quantization for deep neural networks," in *CVPR*, 2017.

[72] E. Park, S. Yoo, and P. Vajda, "Value-aware quantization for training and inference of neural networks," in *ECCV*, 2018.

[73] W. Chen, J. T. Wilson, S. Tyree, K. Q. Weinberger, and Y. Chen, "Compressing neural networks with the hashing trick," in *ICML*, 2015.

[74] M. Rastegari, V. Ordonez, J. Redmon, and A. Farhadi, "Xnor-net: Imagenet classification using binary convolutional neural networks," in *ECCV*, 2016.

[75] Y. Gong, L. Liu, M. Yang, and L. D. Bourdev, "Compressing deep convolutional networks using vector quantization," *arXiv preprint arXiv:1412.6115*, 2014.

[76] S. Son, S. Nah, and K. M. Lee, "Clustering convolutional kernels to compress deep neural networks," in *ECCV*, 2018.

[77] X. Wang and J. Liang, "Scalable compression of deep neural networks," in *ACMMM*, 2016.

[78] S. Han, H. Mao, and W. J. Dally, "Deep compression: compressing deep neural networks with pruning, trained quantization and huffman coding," in *ICLR*, 2016.

[79] F. Tung and G. Mori, "Clip-q : Deep network compression learning by in-parallel," in *CVPR*, 2018.

[80] M. Courbariaux and Y. Bengio, "Binarynet: Training deep neural networks with weights and activations constrained to +1 or -1," *arXiv preprint arXiv:1602.02830*, 2016.

[81] J. Qiu, J. Wang, S. Yao, K. Guo, B. Li, E. Zhou, J. Yu, T. Tang, N. Xu, S. Song, Y. Wang, and H. Yang, "Going deeper with embedded fpga platform for convolutional neural network," in *FPGA*, 2016.

[82] Y. Li, N. Wang, J. Shi, J. Liu, and X. Hou, "Revisiting batch normalization for practical domain adaptation," in *ICLR*, 2017.

[83] J. van Leeuwen, "On the construction of huffman trees," in *ICALP*, 1976.

[84] G. Martin, "Range encoding: an algorithm for removing redundancy from a digitised message," in *the Video & Data Recording Conference*, 1979.

[85] F. Chollet, "Xception: Deep learning with depthwise separable convolutions," in *CVPR*, 2017.

[86] J. Johnson, A. Alahi, and L. Fei-Fei, "Perceptual losses for real-time style transfer and super-resolution," in *ECCV*, 2016.

[87] L. A. Gatys, A. S. Ecker, and M. Bethge, "A neural algorithm of artistic style," *arXiv preprint arXiv:1508.06576*, 2015.

[88] T.-Y. Lin, M. Maire, S. Belongie, J. Hays, P. Perona, D. Ramanan, P. Dollár, and C. L. Zitnick, "Microsoft coco: Common objects in context," in *ECCV*, 2014.

[89] 木倉悠一郎, "Web ブラウザ上での高速な深層ニューラルネットワークの推論手法に関する研究," Master's thesis, 東京大学情報理工学系研究科創造情報学専攻, 2018.

[90] L. Chen, "Keras.js," 2016. https://github.com/transcranial/keras-js.

[91] Theano Development Team, "Theano: A Python framework for fast computation of mathematical expressions," *arXiv preprint arXiv:1605.02688*, 2016.

[92] A. Lavin, "maxDNN: An Efficient Convolution Kernel for Deep Learning with Maxwell GPUs," *arXiv preprint arXiv:1501.06633*, 2015.

[93] P. Laperdrix, N. Bielova, B. Baudry, and G. Avoine, "Browser fingerprinting: A survey," *arXiv preprint arXiv:1905.01051*, 2019.

[94] I. Sanchez-Rola, I. Santos, and D. Balzarotti, "Clock around the clock: Time-based device fingerprinting," in *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*, CCS '18, (New York, NY, USA), pp. 1502–1514, ACM, 2018.

[95] K. He, X. Zhang, S. Ren, and J. Sun, "Delving deep into rectifiers: Surpassing human-level performance on imagenet classification," *arXiv preprint arXiv:1502.01852*, 2015.

[96] G. Klambauer, T. Unterthiner, A. Mayr, and S. Hochreiter, "Self-normalizing neural networks," in *NIPS*, 2017.

[97] Y. Jin, J. Zhang, M. Li, Y. Tian, and H. Zhu, "Towards the high-quality anime characters generation with generative adversarial networks," in *Machine Learning for Creativity and Design, NIPS 2017 Workshop*, 2017.

[98] D. Silver, A. Huang, C. J. Maddison, A. Guez, L. Sifre, G. Van Den Driessche, J. Schrittwieser, I. Antonoglou, V. Panneershelvam, M. Lanctot, *et al.*, "Mastering the game of go with deep neural networks and tree search," *nature*, vol. 529, no. 7587, p. 484, 2016.

Publications

Reviewed Conference

1. Masatoshi Hidaka, Yuichiro Kikura, Yoshitaka Ushiku and Tatsuya Harada. WebDNN: Fastest DNN execution framework on web browser. ACM International Conference on Multimedia (ACMMM) Open Source Software Competition, California, United States, October 2017.

2. Masatoshi Hidaka, Ken Miura and Tatsuya Harada. Development of JavaScript-based deep learning platform and application to distributed training. International Conference on Learning Representations (ICLR) Workshop, Toulon, France, April 2017.

3. Katsunori Ohnishi, Masatoshi Hidaka and Tatsuya Harada. Improved Dense Trajectory with Cross Streams. ACM International Conference on Multimedia (ACMMM), Amsterdam, The Netherlands, pp.257-261, October 2016.

4. Yoshitaka Ushiku, Masatoshi Hidaka and Tatsuya Harada. Three Guidelines of Online Learning for Large-Scale Visual Recognition. Computer Vision and Pattern Recognition (CVPR), pp.3574-3581, Ohio, United States, June 2014.

Un-reviewed Conference

1. 日高雅俊, 三浦拳, 原田達也. JavaScript による環境非依存かつ分散計算可能な Deep Convolutional Neural Network フレームワークの開発. コンピュータビジョンとイメージメディア研究会 (CVIM), 神戸, 2015 年 11 月.

2. 日高雅俊, 郡司直之, 原田達也. ラベル間の階層構造を考慮した Web 画像アノテーション手法に関する研究. パターン認識・メディア理解研究会 (PRMU), pp.201-206, 鳥取, 2013 年 9 月.

Others

1. (解説記事) 日高雅俊, 木倉悠一郎, 牛久祥孝, 原田達也. アプリ不要の高速ディープニューラルネットワーク. 画像ラボ 2018 年 6 月号, 日本工業出版.

2. **(Award)** Masatoshi Hidaka, Yuichiro Kikura, Yoshitaka Ushiku and Tatsuya Harada. WebDNN: Fastest DNN execution framework on web browser. Honorable Mention in ACM International Conference on Multimedia (ACMMM) Open Source Software Competition, California, United States, October 2017.

3. **(Invited Talk)** Masatoshi Hidaka, Naoyuki Gunji, and Tatsuya Harada. MIL at ImageCLEF 2013: Scalable System for Image Annotation. Conference and Labs of the Evaluation Forum (CLEF), Valencia, Spain, September 2013.

4. **(Award)** Masatoshi Hidaka, Naoyuki Gunji, and Tatsuya Harada. The 2nd place in the ImageCLEF Scalable Concept Image Annotation 2013, September 2013.