C-015

ディープラーニングのための双方向循環パイプラインプロセッサ
Bidirectionally Circular Pipelining Processor Specialized for Deep Learning

奥村 拓生‡　　平田 博章‡　　布目 淳‡
Takumi Okumura　Hiroaki Hirata　Atushi Nunome

1. はじめに

ディープラーニングは機械学習の手法の 1 つで，機械に大量のデータを入力して繰り返し学習させることで，データのパターンや特徴の抽出を可能とする．人間と同じような判断や動作をさせることができるので，画像認識や音声認識を必要とする分野や自動運転，医療などで使われている．

ディープラーニングの学習には多大な時間を要するので，それを短縮するために専用のハードウェア機構によるサポートが考えられている．学習の処理は主に行列計算からなるので，GPU(Graphics Processing Unit)を用いたり，あるいは行列計算に特化した TPU(Tensor Processing Unit)が提案されている．しかし，大きいサイズの行列を小さいサイズの行列に分割して計算するので，途中の計算結果をメモリに格納する必要があり，そのためのメモリアクセスが大きなオーバヘッドになっている．

そこで，本稿では，途中の計算結果のメモリへの格納を回避することによってメモリアクセス量を抑え，ディープラーニングの高速化を目指す専用プロセッサのアーキテクチャを提案する．

2. ディープラーニング

2.1 DNN (Deep Neural Network)

ディープラーニングでは，使用目的によって DNN(Deep Neural Network)，CNN(Convolutional Neural Network)，RNN(Recurrent Neural Network)などのニューラルネットワークを用いるが，本稿の双方向循環パイプラインプロセッサは DNN を対象とする．ここでは，まず，DNN を用いた計算手順[1]を説明する．

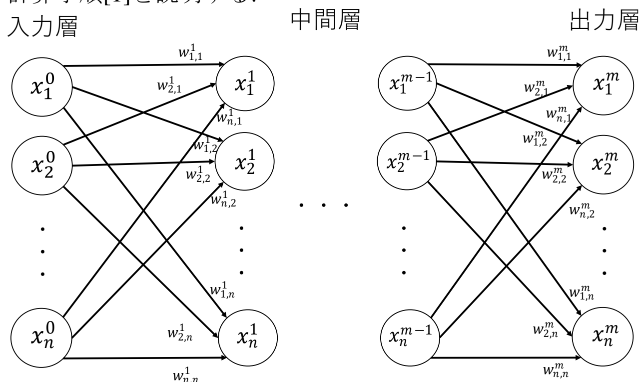

図 1　DNN

DNN を図 1 に示す．DNN は入力層，中間層，出力層の 3 種類の層で構成される．入力層は入力データが得られる層である．中間層は入力層と出力層の間にあり，1 つ以上の層で構成される．出力層は計算結果を出力する層である．本稿では，DNN の層の数を$m+1$とする．

図 1 の丸印はニューロンを，矢印はエッジをそれぞれ表しており，各層のニューロンの数をnとする．i層目$(0 \le i \le m)$の(上から)k番目$(1 \le k \le n)$のニューロンの値をx_k^iで表す．

エッジには重みが付与されており，$i-1$層目のj番目のニューロンからi層目のk番目のニューロンへのエッジの重みを$w_{j,k}^i$で表す．また，各ニューロンにはバイアスがあり，i層目のk番目のニューロンのバイアスをb_k^iで表す．

また，入力データのセット$[x_1^0 \quad x_2^0 \quad \cdots \quad x_n^0]$をいくつ用いて計算するかを決める必要がある．このデータセットの数をバッチサイズといい，本稿ではhで表す．上記では各ニューロンの値を単にx_k^iとしたが，g番目のデータセットを入力したときのx_k^iを$x_{g,k}^i$と表すことにする．

DNN の学習には，入力層から出力層までのニューロンの値を計算する順伝播，DNN の出力を確率分布に変換するソフトマックス関数の計算，ニューロンの重みやバイアスを更新するための逆伝播，の 3 段階の計算過程が必要になる．

2.2 順伝播

順伝播の処理において，i層目のニューロンの値$x_{g,k}^i$を求めるには，まず，$i-1$層目のニューロンの値とエッジの重み，バイアスを用いて式(2.1)の計算を行う．

$$\sum_{l=1}^{n} x_{g,l}^{i-1} w_{l,k}^i + b_k^i \tag{2.1}$$

i層目のニューロンの値の行列をX^i，i層目の重みの行列をW^i，i層目のバイアスの行列をB^iとして，それぞれ以下のように定義する．

$$X^i = \begin{bmatrix} x_{1,1}^i & \cdots & x_{1,n}^i \\ \vdots & \vdots & \vdots \\ x_{h,1}^i & \cdots & x_{h,n}^i \end{bmatrix}$$

$$W^i = \begin{bmatrix} w_{1,1}^i & \cdots & w_{1,n}^i \\ \vdots & \vdots & \vdots \\ w_{n,1}^i & \cdots & w_{n,n}^i \end{bmatrix}, B^i = [b_1^i \quad \cdots \quad b_n^i]$$

このとき，式(2.1)は式(2.2)のように表せる．

$$X^{i-1}W^i + B^i \tag{2.2}$$

X^iは，式(2.2)の結果に活性化関数を適用することで求めることができる．双方向循環パイプラインプロセッサでは，活性化関数に ReLU 関数を用いる．ReLU 関数は入力が 0 を超えていれば入力した値をそのまま出力し，0 以下ならば 0 を出力する関数である．行列Aの各要素に ReLU 関数を適用する関数を$f(A)$と表すことにすると，$X^i (1 \le i < m)$

‡京都工芸繊維大学情報工学専攻　Department of Information Science, Kyoto Institute of Technology

Copyright © 2024 by
The Institute of Electronics, Information and Communication Engineers and
Information Processing Society of Japan All rights reserved.

は式(2.3)で求められる．

$$X^i = f\left(X^{i-1}W^i + B^i\right) \tag{2.3}$$

X^{i-1}を用いてX^iを求めるので，中間層から出力層までは式(2.3)の計算を繰り返すことで，各層のニューロンの値が求まる．ただし，出力層では ReLU 関数を使わず，式(2.4)に示すように，式(2.2)の結果をそのままX^mとする．

$$X^m = X^{m-1}W^m + B^m \tag{2.4}$$

2.3 ソフトマックス関数の計算

順伝播で求めた出力層のニューロンの値にソフトマックス関数を適用する．適用後の行列Yは以下のように表せる．式(2.5)において，c_gは行列X^mのg行目の値の中の最大値を表す．

$$Y = \begin{bmatrix} y_{1,1} & \cdots & y_{1,n} \\ \vdots & \vdots & \vdots \\ y_{h,1} & \cdots & y_{h,n} \end{bmatrix}$$

$$y_{g,k} = \frac{e^{x^{\mathrm{m}}_{g,k} - c_g}}{\sum_{l=1}^{\mathrm{n}} e^{x^{\mathrm{m}}_{g,l} - c_g}} \tag{2.5}$$

式(2.5)では指数関数の値を求める必要があるが，これはe^xをマクローリン展開した式(2.6)で近似する．

$$e^x \approx 1 + x + \frac{x^2}{2!} + \frac{x^3}{3!} + \cdot\cdot\cdot + \frac{x^p}{p!} \tag{2.6}$$

ソフトマックス関数の計算も，双方向循環パイプラインプロセッサ自身が行う．16 ビットのデータに対しては，$p = 10$で十分な精度が得られる．したがって，1 から 10 までの自然数の逆数を定数として用意しておけば，演算セル内の積和演算機構を活かして指数関数の値を求めることができる．

2.4 逆伝播

2.4.1 損失関数の計算

ソフトマックス関数で出力された結果に損失関数を適用する．損失関数とは，予測結果$y_{g,k}$と実際の値(教師データ)とのズレを計算するための関数であり，本稿では損失関数に交差エントロピー誤差を用いる．

ニューラルネットワークで求めた予測結果$y_{g,k}$に対して実際の値を示す教師データを$t_{g,k}$とし，その行列をTで表す．ニューラルネットワークの推論の結果と教師データとの誤差を表す交差エントロピー誤差Lは式(2.7)で求められる．

$$T = \begin{bmatrix} t_{1,1} & \cdots & t_{1,n} \\ \vdots & \vdots & \vdots \\ t_{h,1} & \cdots & t_{h,n} \end{bmatrix}$$

$$L = -\frac{1}{h}\sum_{g=1}^{h}\sum_{k=1}^{n} t_{g,k} \log y_{g,k} \tag{2.7}$$

この誤差Lが小さくなるように，ニューラルネットワークの重みとバイアスを更新する．その方法として，本稿では，誤差逆伝播法と勾配降下法を用いる．

2.4.2 勾配降下法

勾配降下法では，式(2.8)，(2.9)によって重みやバイアスを更新する．ここで，w^t，b^tが更新前の重みとバイアスで，w^{t+1}，b^{t+1}が更新後の重みとバイアスを表す．

$$w^{t+1} = w^t - r \times \frac{\partial L}{\partial w} \tag{2.8}$$

$$b^{t+1} = b^t - r \times \frac{\partial L}{\partial b} \tag{2.9}$$

rは学習率であり，重みやバイアスを一度にどの程度変化させるかを表すメタパラメータである．$\frac{\partial L}{\partial w}$，$\frac{\partial L}{\partial b}$は重みとバイアスの勾配であり，これらの勾配を求めるために誤差逆伝播法を用いる．

2.4.3 誤差逆伝播法

誤差逆伝播法では，まず，損失関数により出力層からの出力値の誤差を求める．この誤差を元に，重みやバイアスの勾配を求め，中間層のニューロンの値の誤差を求めていく．

各層のニューロンの値$x^i_{g,k}$に対する誤差を$d^i_{g,k}$とする．i層目のニューロンの値の誤差の行列をD^iと表す．出力層のニューロンの値の誤差の行列D^mは，式(2.10)で求まる．

$$D^i = \begin{bmatrix} d^i_{1,1} & \cdots & d^i_{1,n} \\ \vdots & \vdots & \vdots \\ d^i_{h,1} & \cdots & d^i_{h,n} \end{bmatrix}$$

$$D^m = \frac{\partial L}{\partial Y} = \frac{1}{h}(Y - T) \tag{2.10}$$

次にi層目の誤差の行列D^iを求める．$i+1$層目のニューロンの誤差の行列と重みの行列とから，i層目の各ニューロンがどれだけ誤差に寄与しているかを計算する．

ここで，逆伝播の計算には，ReLU 関数の導関数を用いる．行列Aの各要素に ReLU 関数の導関数を適用する関数を$f'(A)$と表すことにする．$f'(A)$は行列Aの各要素に対して，それが 0 より大きければ 1 を出力し，0 以下ならば 0 を出力する．

i層目$(1 \le i < m)$の誤差の行列D^iは式(2.11)によって求められる．計算は順伝播と逆向きになるので，W^{i+1}を転置して計算に用いる．$\odot$は要素毎に積をとるアダマール積を表す．

$$D^i = \left(D^{i+1}{W^{i+1}}^{\mathsf{T}}\right) \odot f'(X^{m-1}) \tag{2.11}$$

i層目の重みの行列W^iとバイアスの行列B^iの勾配$\frac{\partial L}{\partial W^i}$，$\frac{\partial L}{\partial B^i}$を，式(2.12)，(2.13)により求める．

$$\frac{\partial L}{\partial W^i} = \frac{\partial Y}{\partial W^i}\frac{\partial L}{\partial Y} = {X^{i-1}}^{\mathsf{T}} D^i \tag{2.12}$$

$$\frac{\partial L}{\partial B^i} = \frac{\partial L}{\partial Y} = \left[\sum_l d^i_{1,l} \quad \cdots \quad \sum_l d^i_{n,l}\right] \tag{2.13}$$

以上のようにi層目の誤差を求めて，i層目の重みとバイアスの勾配を求める．これらを用いて，式(2.8)，(2.9)により重みとバイアスを更新する．以上の計算を出力層から入力層まで順番に行い，全ての層の重みとバイアスを更新する．

3. 関連研究

ディープラーニングの計算はそのほとんどが行列の積和演算であり，これを高速化したり，計算時のデータ通信を最適化する様々な専用ハードウェアが提案されている．

Copyright © 2024 by
The Institute of Electronics, Information and Communication Engineers and
Information Processing Society of Japan All rights reserved.

シストリックアレイによって行列計算を効率よく行う専用ハードウェアに TPU[2]や Automated Systolic Array[3]がある．シストリックアレイは，規則的に配置された演算器で構成され，これらの演算器が一定の時間間隔でデータを横方向や縦方向に同時に受け渡しながら，並列に処理を行う．TPU は 256×256 個の演算器を備え，専用のローカルメモリに重みのデータを格納する．Automated Systolic Array は CNN を FPGA(Field Programmable Gate Array)を用いて実装しており，シストリックアレイによって効率的な畳み込み演算を行う．適切なデータを特定の演算器で利用できるように，計算負荷やデータの依存関係を解析することで適切なデータフローを決定し，横方向と縦方向でデータを受け渡して計算する．

ディープラーニングの処理をパイプライン化する専用ハードウェアに PipeLayer[4]と Minerva[5]がある．PipeLayer は CNN 用の ReRAM(Resistive Random Access Memory)ベースの PIM(Processing in Memory)アクセラレータであり，層間並列処理を活用する効率的なパイプライン処理を提案している．Minerva は入力データや重み用のメモリを持つ複数の演算器で構成されている．それぞれの演算器は独立して動作し，計算をパイプライン化している．ディープラーニングの計算で重要でない操作を予測し，その操作をパイプライン中でスキップすることで，電力効率を高める手法を提案している．

特徴的なデータフローによって CNN の計算を効率よく行う専用ハードウェアに AtomLayer[6]や Eyeriss[7]がある．AtomLayer は効率的な CNN の計算を行う ReRAM ベースのアクセラレータである．Rotating Crossbars という独自のフィルターマッピングと演算器間でのデータ再利用により，CNN の計算を効率よく行う．また，Eyeriss は 168 個の処理要素を備え，Row Stationary というデータフローに基づいて特定の行のデータを再利用することで，CNN の計算を効率よく行う．

他にも，上記とは異なる手法を用いた ReRAM ベースのアクセラレータ[8][9]や，電気信号ではなく光電子ハイブリッドフォトニクスによる計算を行っているアクセラレータ[10][11]，GPU を用いた分散学習による高速化[12]など，様々な手法が提案されている．

4. 双方向循環パイプラインプロセッサ

3.で述べたように，行列計算を高速化する様々な専用アーキテクチャが提案されている．しかし，行列計算の場合，小さいサイズの行列に分割して計算しなければならない．例えば，式(4.1)の行列計算を考える．行列$\begin{bmatrix} A_{11} & A_{12} & A_{13} \\ A_{21} & A_{22} & A_{23} \end{bmatrix}$は行列演算機構で 1 回に計算できるサイズを超えた大きなサイズの行列であり，したがって，行列演算機構で扱えるサイズの行列A_{11}，A_{12}，A_{13}，A_{21}，A_{22}，A_{23}に分割して計算する．

$$\begin{bmatrix} A_{11} & A_{12} & A_{13} \\ A_{21} & A_{22} & A_{23} \end{bmatrix} \begin{bmatrix} B_1 \\ B_2 \\ B_3 \end{bmatrix} = \begin{bmatrix} A_{11}B_1 + A_{12}B_2 + A_{13}B_3 \\ A_{21}B_1 + A_{22}B_2 + A_{23}B_3 \end{bmatrix} \quad (4.1)$$

このとき，途中の計算結果$A_{11}B_1$，$A_{12}B_2$，$A_{13}B_3$，$A_{21}B_1$，$A_{22}B_2$，$A_{23}B_3$をメモリに格納するので，メモリアクセスが増えてしまう．より大きな行列では，アクセス回数が増え，それが大きなオーバヘッドになる．

そこで，本稿では，データを循環させるための通信路を設けてパイプライン処理することで，途中の計算結果をメモリに一時的に格納することを回避し，メモリへのアクセス量を抑える専用プロセッサを提案する．

4.1 システム構成

図 1 の DNN を対象とする双方向循環パイプラインプロセッサの構成を図 2 に示す．各演算セルには加算器と乗算器のほかにレジスタとローカルメモリを備える．ローカルメモリには，計算前に DNN の重みとバイアス，教師データを格納しておく．ニューロンの数nに合わせて，演算セルの数もn個とする．以降，左からq番目の演算セルをP_qと呼ぶことにする．

矢印は単方向のデータの通信路を表す．マルチプレクサは，外部からのデータをP_1に入力するのか，P_1の出力データを再びP_1の入力データとしてフィードバックするのかを選択する．

また，入力データのバッチサイズは，後述する理由により 2 とする．

演算セルP_qのローカルメモリには，重み$w_{j,q}^i$とバイアスb_q^iを格納する．$w_{j,q}^i$は$i-1$層目のj番目のニューロンからi層目のq番目のニューロンへのエッジの重みである．

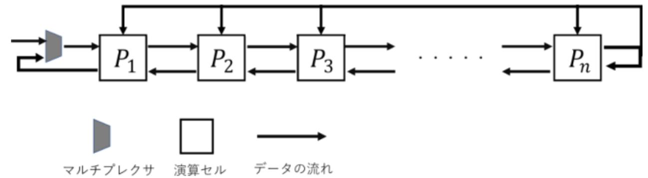

図 2　双方向循環パイプラインプロセッサの構成

4.2 順伝播

入力層から中間層までの順伝播時の演算セルの処理について述べる．まず，バッチサイズを 1 として考える．演算セル間のデータの流れを図 3 に示す．図 3 はX^0W^1の計算時の演算セル間のデータの流れを描いており，バイアスB^1の加算時や ReLU 関数の適用時は演算セル間でのデータの受け渡しは必要ない．

それぞれの演算セルP_qでは，以下の処理を行う．ここで，$L.in$は左隣の演算セルから受け取る値を表し，$L.out$は左隣の演算セルへ渡す値，$R.in$は右隣の演算セルから受け取る値，$R.out$は右隣の演算セルへ渡す値をそれぞれ表す．また，$reg()$は括弧内のデータをレジスタに記憶することを示し，$local()$は括弧内のデータをローカルメモリに記憶することを示す．

$$L.in: x_k^0$$

$$reg(sum) = reg(sum) + L.in \times w_{k,q}^1$$

$$R.out = L.in$$

$$local(x_k^0) = L.in$$

演算セル間のデータの流れは，図 3 に示すように，演算セルP_1に入力データ$X^0 = [x_1^0 \ \cdots \ x_n^0]$を順に入力する．各

Copyright © 2024 by
The Institute of Electronics, Information and Communication Engineers and
Information Processing Society of Japan All rights reserved.

演算セルは左から右へと，順次，入力データx_k^0を転送する．その結果，演算セルP_qでは，最終的に式(4.2)の積和演算の結果sumをレジスタに得ることができる．

$$x_1^0 \times w_{1,q}^1 + x_2^0 \times w_{2,q}^1 + \cdots + x_n^0 \times w_{n,q}^1 \qquad (4.2)$$

その後，sumにb_q^1を足し，ReLU 関数を適用する．そうすることで，演算セルP_qでx_q^1を求めることができ，全体として，$X^1 = f(X^0W^1 + B^1)$の計算結果を得ることができる．

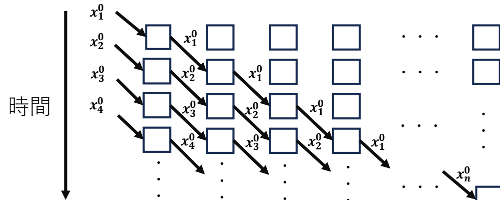

図 3　順伝播時(入力層から中間層まで)のデータの流れ

次に中間層から出力層までの演算セルの処理について述べる．全体のデータの流れを図 4 に示す．図 4 では簡単のため$n = 5$としている．

演算セルP_qでは，まず，求めたx_q^1を用いて，以下の処理を行う．$x_q^1 \times w_{q,q}^2$を計算し，左隣の演算セルにx_q^1を渡す．

$$local(x_q^1) = reg(sum)$$
$$reg(sum) = x_q^1 \times w_{q,q}^2$$
$$L.out = x_q^1$$

次に，演算セルP_qでは以下の処理を行う．

$$R.in: x_k^1$$
$$L.in: x_j^1$$
$$reg(sum) = reg(sum) + R.in \times w_{k,q}^2$$
$$L.out = R.in$$
$$R.out = L.in$$
$$local(x_k^1) = R.in$$

x_k^1は，小さいkから時間的に先に求まる．つまり，左端の演算セルから順に計算が終了する．x_k^1の計算に引き続いてx_k^2を求める計算を開始するためには，x_k^1を左方向に転送しなければならない．また，必ずしも$x_1^1 \times w_{1,q}^2$の項から順に計算するのではなく，演算セルP_qでは$x_q^1 \times w_{q,q}^2$の項から順に加算することで，データ待ちのオーバヘッドが生じるのを回避する．結局，図 4 に示すように，x_k^1は，当初は左方向に転送され，左端の演算セルP_1に達すると，今度は右方向に転送される．そして右端の演算セルP_nで再び折り返して左方向に転送される．左方向に転送されているときにのみ演算セルで積和演算に使用され，右方向に転送されているときは単に演算セル間を移動するだけである．このようにして演算セルP_qでは，最終的に式(4.3)の結果がsumに格納される．

$$\sum_{l=q}^{n} x_l^1 w_{l,q}^2 + \sum_{l=1}^{q-1} x_l^1 w_{l,q}^2 \qquad (4.3)$$

その後，sumにb_q^2を足し，ReLU 関数を適用する．そうすることで，演算セルP_qで，x_q^2を求めることができ，全体として，$X^2 = f(X^1W^2 + B^2)$の計算結果を得ることができる．

以上が，X^1からX^2を求める処理内容であり，同様の処理を層ごとに中間層から出力層まで繰り返す．また，出力層の値X^mを計算するときは ReLU 関数を使わず，演算セルP_qで，x_q^mを求め，ローカルメモリに格納する．

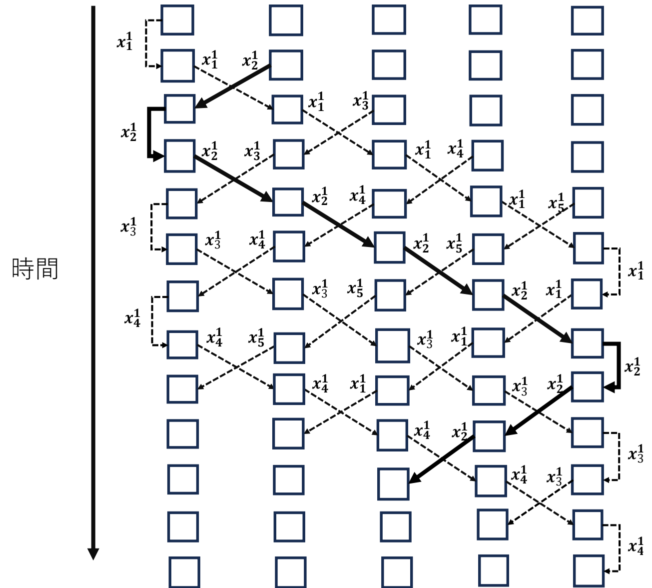

図 4　順伝播時(中間層から出力層まで)のデータの流れ

さて，図 4 では，左隣の演算セルP_{q-1}から受け取ったデータを右隣の演算セルP_{q+1}へ転送するとき，演算セルP_qの演算器は働いていない．そこで，バッチサイズを 2 とすることで，演算資源の遊びをなくす．

バッチサイズが 2 のデータを$X^1 = \begin{bmatrix} x_{1,1}^1 & x_{1,2}^1 & \cdots & x_{1,n}^1 \\ x_{2,1}^1 & x_{2,2}^1 & \cdots & x_{2,n}^1 \end{bmatrix}$と表す．図 5 に示すように，バッチサイズが 1 の時のデータの流れを 2 つ重ね合わせたように処理することで，全ての演算セル内の演算器を稼働状態に保つことができる．

演算セルP_qでは，式(4.4)，(4.5)の計算をインタリーブして見かけ上同時に実行する．

$$sum_1 = \sum_{l=q}^{n} x_{1,l}^1 w_{l,q}^2 + \sum_{l=1}^{q-1} x_{1,l}^1 w_{l,q}^2 \qquad (4.4)$$

$$sum_2 = \sum_{l=q}^{n} x_{2,l}^1 w_{l,q}^2 + \sum_{l=1}^{q-1} x_{2,l}^1 w_{l,q}^2 \qquad (4.5)$$

なお，図 3 や以降の説明では簡単のためにバッチサイズを 1 として説明するが，全ての処理においてインタリーブにより 2 つのデータセットに対する処理を見かけ上同時に行う．

4.3 ソフトマックス関数の計算

まず，$x_q^{\mathrm{m}} - c_1$を計算する時の演算セルの処理について述べる．全体のデータの流れを図 6 に示す．

演算セルP_qでは，以下の処理を行う．

$$L.in: M_{q-1}$$
$$R.out = compare(L.in,\ x_q^{\mathrm{m}})$$

Copyright © 2024 by
The Institute of Electronics, Information and Communication Engineers and
Information Processing Society of Japan All rights reserved.

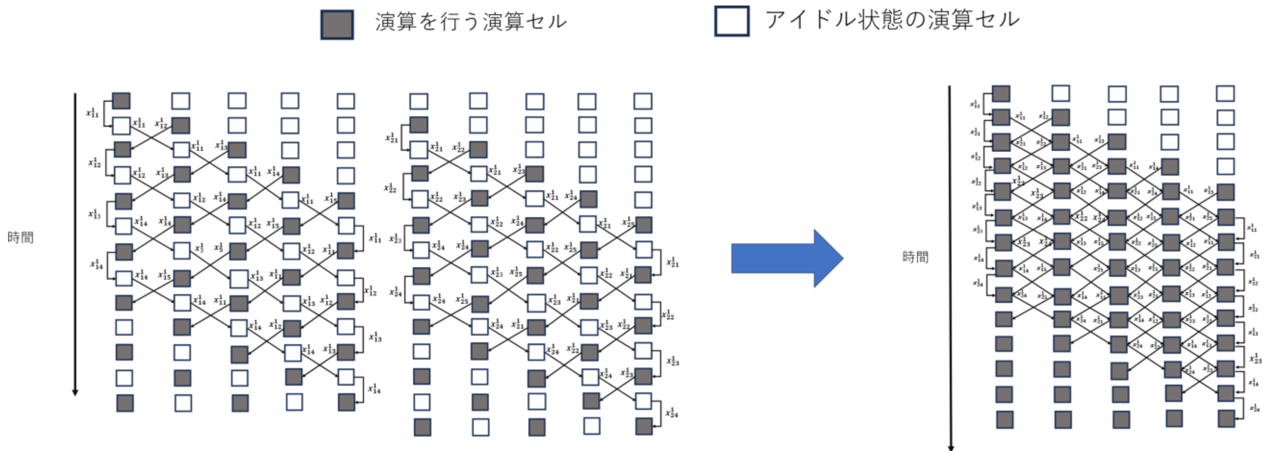

図５　インタリーブによる稼働状態の改善

$compare(a,b)$はaとbの値の大きい方を選択する処理を表す．左隣の演算セルから入力するM_{q-1}はx_1^{m}からx_{q-1}^{m}までの中の最大値を表し，x_q^{m}までの中での最大値M_qを右隣の演算セルに出力する．

このようにして，図 6 に示すように，右端の演算セルP_nの右側から最大値c_1が出力される．そこで，このc_1を左端の演算セルP_1に渡し，以降は，順にc_1を全ての演算セルに配布する．このとき，演算セルP_qは以下の処理を行う．

$$L.in: c_1$$
$$reg(exp) = x_q^{\mathrm{m}} - L.in$$
$$R.out = L.in$$
$$local(exp) = reg(exp)$$

これで全ての演算セルP_qで$x_q^{\mathrm{m}} - c_1$が求まる．

次に，指数関数$e^{x_q^{\mathrm{m}}-c_1}$の値を求め，ソフトマックス関数の計算を行う．このときの全体のデータの流れを図 7 に示す．

まず，演算セルP_qでは，ローカルメモリに格納した$x_q^m - c_1$を用いて，式(2.6)により指数関数$e^{x_q^{\mathrm{m}}-c_1}$の値を求め，ローカルメモリに格納する．$e^{x_q^{\mathrm{m}}-c_1}$の計算には，演算セル間でデータの受け渡しを行う必要はない．このようにして，全ての演算セルでそれぞれ独立に指数関数を計算する．

次に演算セルP_qでは，以下の処理を行う．ここで，S_{q-1}は$\sum_{l=1}^{q-1} e^{x_l^{\mathrm{m}}-c_1}$を表し，$S_q$を右側から出力する．

$$L.in: S_{q-1}$$
$$R.out = L.in + e^{x_q^{\mathrm{m}}-c_1}$$

このようにして，図 7 に示すように，右端の演算セルP_nから指数関数の合計値$S_n = \sum_{l=1}^{\mathrm{n}} e^{x_l^{\mathrm{m}}-c_1}$が出力され，これを左端の演算セル$P_1$に渡す．

最後に，演算セルP_qでは，以下の処理を行う．

$$L.in: S_n$$
$$reg(frac) = \frac{e^{x_q^{\mathrm{m}}-c_1}}{L.in}$$
$$R.out = L.in$$
$$local(y_q) = reg(frac)$$

このように各演算セルでy_qの値を求め，それぞれのローカルメモリに格納する．

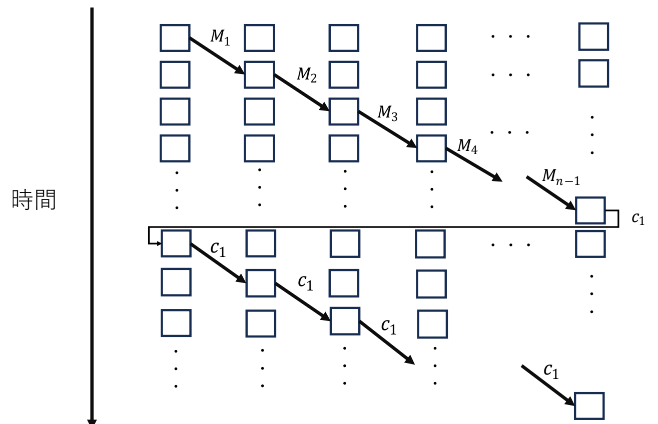

図６　最大値を求める時のデータの流れ

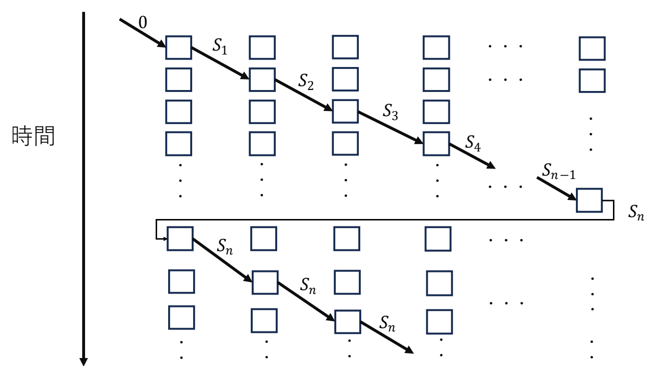

図７　ソフトマックス関数の計算時のデータの流れ

4.4 逆伝播

逆伝播時の演算セルの処理について述べる．全体のデータの流れを図 8 に示す．

Copyright © 2024 by
The Institute of Electronics, Information and Communication Engineers and
Information Processing Society of Japan All rights reserved.

まず，出力層のニューロンの値の誤差d_q^mを求める．y_qと教師データt_qは演算セルP_qのローカルメモリに保存しているので，演算セル間でデータの受け渡しをすることなく，演算セルP_qで$d_q^m = (y_q - t_q)$を計算してローカルメモリに格納する．

次に，$\mathrm{m}-1$層目の誤差の行列$D^{\mathrm{m}-1}$を求める．演算セルP_qは，以下の処理を行う．$G_{k,q-1}$は$\sum_{l=1}^{q-1} d_l^m \times w_{k,l}^m$を表し，$G_{k,q}$を右隣の演算セルへ出力する．

$$L.in: G_{k,q-1}$$
$$R.out = L.in + d_q^m \times w_{k,q}^m$$

このようにして，右端の演算セルP_nから$G_{k,n}$の値が出力される.その値を左からk番目の演算セルP_kに渡し，それぞれの演算セルで ReLU 関数の導関数に通して，$\mathrm{m}-1$層目のニューロンの値の誤差d_q^{m-1}を求め，ローカルメモリに格納する．

次に，W^{m}とB^{m}を更新する．ローカルメモリに必要なデータは全て格納されているので，演算セルP_qでは，他の演算セルと通信する事なく，重みの勾配$\frac{\partial L}{\partial w_{k,q}^m} = x_k^{m-1} \times d_q^m$とバイアスの勾配$\frac{\partial L}{\partial b_q^m} = d_q^m$を計算する．これらの勾配から$w_{k,q}^m$と$b_q^m$を更新し，ローカルメモリに格納する．

以上がニューロンの値の誤差から重みとバイアスを更新するまでの処理内容であり，これらの処理を出力層から入力層まで繰り返す．

重みとバイアスの値を更新し終わると，別のデータを入力して順伝播の処理を開始する．

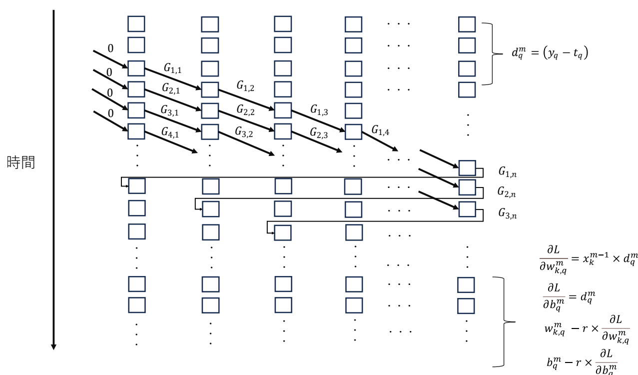

図 8　逆伝播時のデータの流れ

5. まとめ

本稿では，途中の計算結果をメモリに格納することを回避することによってアクセス量を抑え，ディープラーニングの高速化を図る双方向循環パイプラインプロセッサを提案した．プロセッサ内でデータが循環するように通信路を設け，計算結果を隣の演算セルに渡す．こうすることで，計算に必要なデータをメモリにアクセスせずに受け取ることができるので，メモリアクセス量を抑えることができる．

今後の課題としては，性能およびメモリアクセス量の評価と，DNN 以外の CNN や RNN への対応に関する検討，などが挙げられる．

参考文献

[1] 斎藤 康毅, “ゼロから作る Deep Learning”,株式会社オライリー・ジャパン(2016).

[2] Norman P.Jouppi, Cliff Young, Nishant Patil, David Patterson, Gaurav Agrawal, Raminder Bajwa, et al., "In-Datacenter Performance Analysis of a Tensor Processing Unit.", 44th Annual International Symposium on Computer Architecture, 1–12 (2017).

[3] Xuechao Wei, Cody Hao Yu, Peng Zhang, Youxiang Chen, Yuxin Wang, Han Hu, Yun Liang, and Jason Cong, "Automated systolic array architecture synthesis for high throughput CNN inference on FPGAs.", 54th Annual Design Automation Conference, No.:29, 1–6 (2017).

[4] Linghao Xuehai Qian, Hai Li, Yiran Chen. "PipeLayer: A pipelined ReRAM-based accelerator for deep learning.", IEEE International Symposium on High Performance Computer Architecture, 541–552 (2017).

[5] Brandon Reagen, Paul Whatmough, Robert Adolf, Saketh Rama, Hyunkwang Lee, Sae Kyu Lee, José Miguel, Hernández-Lobato, Gu-Yeon Wei, David Brooks, "Minerva:Enabling low-power, highly-accurate deep neural network accelerators.", 43rd Annual International Symposium on Computer Architecture, 267–278 (2016).

[6] Ximing Qiao, Xiong Cao, Huanrui Yang, Linghao Song, Hai Li, "AtomLayer: A Universal ReRAM-Based CNN Accelerator with Atomic Layer Computation", 55th ACM/ESDA/IEEE Design Automation Conference, 1-6 (2018).

[7] Yu-Hsin Chen et al., "Eyeriss: An energy-efficient reconfigurable accelerator for deep convolutional neural networks.", IEEE Journal of Solid-State Circuits, Volume: 52, Issue: 1, 127–138 (2017).

[8] Ali Shafiee, Anirban Nag, Naveen Muralimanohar, Rajeev Balasubramonian, John Paul Strachan, Miao Hu, R. Stanley Williams, Vivek Srikumar, “ISAAC: A Convolutional Neural Network Accelerator with In-Situ Analog Arithmetic in Crossbars", 43rd Annual International Symposium on Computer Architecture, 14–26 (2016).

[9] Ping Chi, Shuangchen Li, Cong Xu, Tao Zhang, Jishen Zhao, Yongpan Liu, Yu Wang, Yuan Xie, "PRIME: A Novel Processing-in-memory Architecture for Neural Network Computation in ReRAM-based Main Memory", 43rd Annual International Symposium on Computer Architecture, 27–39 (2016).

[10] Jiaxin Peng, Yousra Alkabani, Krunal Puri, Shuai Sun, Volker Sorger, Tarek El-Ghazawi, "DNNARA: A Deep Neural Network Accelerator using Residue Arithmetic and Integrated Photonics", 49th International Conference on Parallel Processing, 1–11 (2020).

[11] Jiaxin Peng, Yousra Alkabani, Krunal Puri, Xiaoxuan Ma, Volker Sorger, Tarek El-Ghazawi, "A Deep Neural Network Accelerator using Residue Arithmetic in a Hybrid Optoelectronic System" ACM Journal on Emerging Technologies in Computing System, Volume 18, Issue 4, Article No.:81, 1–26 (2022).

[12] 本田 巧, 山崎 雅文, 笠置 明彦, 田渕 晶大, 福本 尚人, 田原 司睦, 池 敦, 中島 耕太, "大規模 GPU クラスタにおける ResNet-50/ImageNet 学習の高速化", 情報処理学会第 170 回 HPC 研究会報告, Vol.2019-HPC-170, No.6, 1–7 (2019).

Copyright © 2024 by
The Institute of Electronics, Information and Communication Engineers and
Information Processing Society of Japan All rights reserved.