

GAN を用いた新たなアバター生成

藤間裕大 小野佳祐 土居遼太郎 美谷佳寛
千葉大学院 融合理工学府 基幹工学専攻 修士1年

1 概要

今年度のLSIデザインコンテストにおける設計課題は「Generative Adversarial Networks (GAN)」である。GANは、画像生成、超解像、スタイル変換、異常検知などに用いられる生成ネットワークである。我々は、画像生成の応用として、様々な髪型、髪色、目、服のアバターの画像を学習されることで、データセットには含まれない新たなアバターを生成するGANの回路設計および実装を行った。

本システムの特徴を以下に挙げる。

1. ランダムノイズ (100次元) の入力 z に対し、新たなアバター画像を生成するGeneratorネットワーク (CNN) のFPGA実装
2. ランダムノイズ z に対し、100次元のランダムベクトル v_1, v_2 を用いて、 z はR画像の学習、 $z + v_1$ はG画像の学習、 $z + v_2$ はB画像の学習を行うことで、一つのGeneratorネットワークでカラー画像の出力を実現
3. 大規模なCNNモデルの実装を目的とした、PSとPLが連携する協調型アーキテクチャ
4. 今後の高速化、高解像度化が可能な拡張性のある構成

2章では、本システムの設計に取り組む背景を示す。3章および4章では、準備として生成モデルの中でのGANの位置づけと、GANの学習アルゴリズムについて説明する。5章では、ソフトウェア実装における本システムの特徴を示す。6章では、CNNの特徴と本システムでの利用方法について述べる。7章では、ハードウェア設計に向けて、エミュレータ側での固定小数点の検討について示す。そして、8章

では提案するシステム構築に用いる使用機器についてまとめる。9章では、ハードウェア構成 (PS/PL協調・AXI通信) を説明し、10章では、Generatorの実装についてまとめる。11章では、FPGA実装にあたっての工夫点をまとめている。12章、13章ではそれぞれシミュレーション動作、実機動作について記載し、14章で本システムの今後の展望をまとめた。15章でまとめを、16章で謝辞を述べる。

2 背景

近年、オンライン空間を活用したサービスが広く普及し、情報発信や交流の場として利用される機会が増加している[1]。SNS、ゲーム、メタバース、遠隔コミュニケーションなど、物理的な距離に依存しないコミュニケーション手段が社会に定着しつつある。

このようなオンライン空間において、人をどのように表現するかは重要な要素である。現実空間とは異なり、オンライン上では外見や振る舞いを直接的に共有することが難しく、その代替として視覚的な表現手段が用いられてきた。その代表例として、オンライン空間上で人の存在を表す手段としてアバターが広く利用されている(図1)。アバターは単なる装飾ではなく、利用者の個性や属性を反映する表現手段として機能しており、オンライン上のコミュニケーションにおいて重要な役割を果たしている[2]。

今回は、このようなアバターを対象とし、アバター画像を生成するシステムの設計および実装を行う。具体的には、既存のアバター画像をもとに、新たな外観を持つアバター画像を生成することを目的とする。

また、画像生成処理をFPGA上に実装することで、生成モデルを用いた画像生成をハードウェアレベルで実現する構成について検討する。



図1 オンライン空間におけるアバター利用のイメージ図 (Gemini を用いて作成)

3 教師あり学習とGANによる画像生成

本章では、画像生成に用いられる代表的な学習手法として、教師あり学習およびGANによる画像生成の考え方を整理する。

3.1 教師あり学習による画像生成

教師あり学習による画像生成では、入力データと対応する正解画像の組を用いて学習が行われる。学習過程においては、生成画像と正解画像との差を損失関数として定義し、その値が最小となるようにモデルのパラメータが更新される。このとき、画素ごとの値の差分を評価する損失関数が用いられることが多く、モデルは正解画像の画素値を再現する方向に学習が進む。

このような学習方法は、入力と出力の対応関係が明確に定義されているタスクにおいて有効であり、既存画像の再構成などの用途で広く用いられてきた。学習データに含まれる入力に対しては、正解画像に近い出力を安定して得ることができる点が特徴である。

一方で、教師あり学習では学習時に与えられた正解画像を基準としてパラメータが更新されるため、学習データに含まれない新規の

入力に対しては、明確な正解画像が存在しない。この場合、モデルは複数の妥当な出力を同時に満たそうとする挙動となり、結果として生成画像が平均的な外観となることがある。

3.2 GANによる画像生成

GANでは、生成器と識別器の二つのネットワークを用いた学習が行われる。生成器は入力から画像を生成し、識別器はその画像が正解データに由来するものかどうかを判別する。学習過程において、生成器は識別器の判別結果をもとにパラメータを更新する。

この学習では、生成画像が特定の正解画像と画素ごとに一致しているかどうかではなく、正解データとして与えられた画像群に含まれる特徴を持っているかどうかが評価基準となる。したがって、生成器は正解画像そのものを再現するのではなく、正解データに共通する外観の特徴を反映した画像の生成方法を学習する。

このため、新規の入力が与えられた場合においても、生成器は正解データらしい外観を持つ画像を出力することが可能となる。教師あり学習とGANは、いずれも画像生成に用いられる手法であるが、学習の目的および評価の基準が異なっており、それぞれ異なる特性を持つ手法として位置づけられる。

4 準備

本章では、4.1において生成モデルにおけるGANの位置づけを示し、4.2においてGANの学習アルゴリズムについて説明する。

4.1 生成モデル

生成モデルは未知の真の分布 $p_{data}(x)$ をモデル分布 p_θ で近似することを目標とする。これは、モデルから画像 x が生成される確率である周辺尤度 $p_\theta(x) = \int p_\theta(z)p_\theta(x|z)dz$ を最大化することに等しいが、高次元空間において、この $p_\theta(x)$ を直接的に計算することは難しい。この問題に対処したアルゴリズムとしてVAEとGANがある。

4.1.1 VAE

画像データ x の特徴を潜在変数として次元の圧縮を行い、その潜在変数に基づいて画像を生成する仕組みをオートエンコーダー(AE)という。AEでは潜在変数と出力画像は一対一対応だが、潜在変数を確率分布として未知データの出力を可能とした技術としてVAE(Variational Auto Encoder)[3]がある。(図2)

VAEでは $p_\theta(x)$ を直接的に計算することは難しいという問題に対して、潜在変数 z を介して、推論モデル $q_\phi(z|x)$ および生成モデル $p_\theta(x|z)$ を導入することで対処している。

VAEの推論モデル $q_\phi(z|x)$ は、入力データ x に対して潜在変数の分布を近似するものであり、ニューラルネットワークを用いて平均 μ と分散 σ^2 を出力することで、 z が従うガウス分布 $\mathcal{N}(\mu, \sigma^2)$ を定義する。生成モデル $p_\theta(x|z)$ では、潜在変数 z を事前分布 $p(z)$ からサンプリングし、それに基づいて新しい画像を生成することが可能となる。

VAEでは $p(x|z)$ をガウス分布としてモデル化するため、再構成誤差は二乗誤差に一致す

る。しかしながら、二乗誤差は多峰的な分布に対して平均化を促す性質がある。その結果、高周波成分を正確に再現する鋭い画像よりも、平滑化された曖昧な画像の方が誤差が小さくなりやすく、VAEでは生成画像がぼやけやすいというアルゴリズム上の課題が存在する。

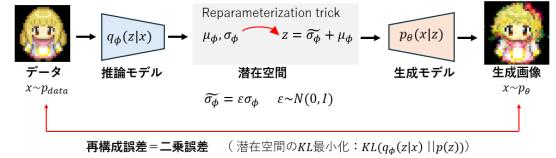


図2 VAEの概要図

4.1.2 GAN

VAEと異なり、GAN[4]は明示的な尤度関数 $p_\theta(x)$ や潜在変数の事後分布 $q(z|x)$ を定義せず、識別モデルを介して生成分布とデータ分布との差異を評価することで、分布全体を暗黙的に近似する生成モデルである。(図3)

具体的には、Generatorとよばれる生成モデル $G(z)$ とDiscriminatorと呼ばれる識別モデル $D(x)$ を用いて、これらを敵対的に学習させることで画像のデータ分布を学習する。

生成モデル $G(z)$ は潜在変数 z を画像空間に写像するパラメータを学習し、識別モデル $D(x)$ は入力画像 x が正解データか $G(z)$ かを判別する関数として機能する。VAEとは異なり、 p_{data}, p_θ のJSダイバージェンスを最小化するように学習が進むため、高周波成分も再現したシャープな画像が生成されやすい。

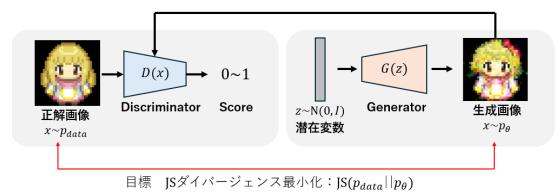


図3 GANの概要図 (VAE 比較)

4.2 GAN (Generative Adversarial Networks) の学習アルゴリズム

GANの目標は画像データ x に対する生成モデルの分布 p_g を学習することである。そのために、まずノイズ変数 z に対する事前分布 $p_z(z)$ を定義し、これをデータ空間へ移す写像として $G(z; \theta_g)$ を学習する。 θ_g はNNのパラメータである。これにより、単純な事前分布 $z \sim p_z(z)$ を関数 $G(z)$ に入力することで、新たなデータ分布 $x = p_g$ を生成できる。

次に、单一のスカラー値を出力するペアプロトロン $D(x; \theta_d)$ を定義する。 $D(x)$ は、入力 x が生成分布 p_g からではなく、実データ分布から得られたものである確率を表す。

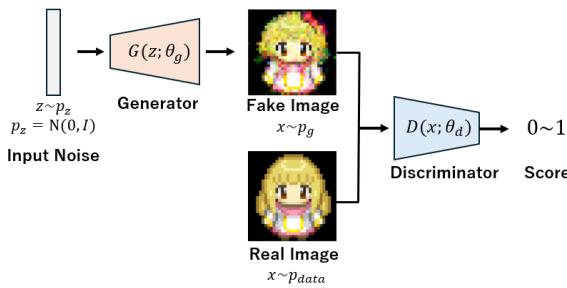


図4 GANの概要図

GANでは、式(1)に示すminimax問題を解くことにより、 $G(z)$ はデータ分布を学習し、 D は正解画像のデータ分布と p_g の判別方法を学習する。

$$\begin{aligned} \min_G \max_D V(D, G) &= \mathbb{E}_{x \sim p_{\text{data}}(x)} [\log D(x)] \\ &+ \mathbb{E}_{z \sim p_z(z)} [\log(1 - D(G(z)))] \end{aligned} \quad (1)$$

ここで、先に D の最適化を完全に進めてしまうと、 $D(G(z)) \approx 0$ となってしまい、 G の損失関数 $L_G = \mathbb{E}_{z \sim p_z(z)} [\log(1 - D(G(z)))]$ の勾配は $\partial \mathcal{L}_G / \partial \theta_g \approx 0$ となる。その結果、 G の有効な更新ができず、生成器 G が $p_z(z)$ から p_g へと変換する過程を学習できなくなってしまう。そのため、 D と G を交互に更新すること

により、 D の過度な最適化を避け、 G の学習を有効に進めることができる。

4.2.1、4.2.2にてDiscriminatorおよびGeneratorの最適解について、4.2.3にてGAN全体の学習の流れについて詳細を説明する。

4.2.1 Discriminator 最適解

期待値 E は確率変数 x に対して関数 $f(x)$ の平均をとることで求まるから、離散の確率密度関数 $P(x)$ に対して $E[f(x)] = \sum f(x)P(x)$ 、連続の確率密度関数 $p(x)$ 場合は $E[f(x)] = \int f(x)p(x)dx$ と記述できる。

識別器 D では、式(1)の最大化を目指すため、式(1)を変形すると式(2)のようになる。

$$\begin{aligned} V(D, G) &= \mathbb{E}_{x \sim p_{\text{data}}(x)} [\log D(x)] + \mathbb{E}_{z \sim p_z(z)} [\log(1 - D(G(z)))] \\ &= \int dx [p_{\text{data}}(x) \log D(x) + p_g(x) \log(1 - D(x))] \end{aligned} \quad (2)$$

ここで、関数 $y = a \log y + b \log(1 - y)$ は $y = a/(a+b)$ で最大値をとることから、式(2)は式(3)で最大値をとる。

$$D_G^*(x) = \frac{p_{\text{data}}(x)}{p_{\text{data}}(x) + p_g(x)} \quad (3)$$

よって、識別器の最適解は $D_G^*(x)$ と分かる。GANの学習が理想的に進行した場合、生成分布は実データ分布に一致し $p_{\text{data}} = p_g$ となる。このとき、識別器の最適解は $D_G^*(x) = 1/2$ である。

4.2.2 Generator の最適解

KL ダイバージェンス

確率分布の類似性を測る指標としてKLダイバージェンスがあり、真の確率分布 $p(x)$ を $q(x|\theta)$ で表現することは、式(4)に示すKLダイバージェンスを最小化する問題に帰着することができる。

$$D_{KL}(p||q) = \int p(x) \log \frac{p(x)}{q(x)} dx \quad (4)$$

JS ダイバージェンス

KLダイバージェンスは非対称性を持ち、 $D_{KL}(P||Q) \neq D_{KL}(Q||P)$ であるため距離として扱うことはできない。そこで、対称性を持つための指標としてJSダイバージェンスがあり、式(5)で示す。

$$D_{JS}(p||q) = \frac{D_{KL}(p||m) + D_{KL}(q||m)}{2} \quad (5)$$

生成器の学習では、式(2)の $V(D, G)$ の最小化を考える。式(2)の $V(D, G)$ に対して、式(3)の最適演算子 D^* および式(4)、式(5)を適用すると式(6)になる。

$$\begin{aligned} V(G) &= \int p_{\text{data}}(x) \log \frac{p_{\text{data}}}{p_{\text{data}} + p_g} dx \\ &\quad + \int p_g(x) \log \left(1 - \frac{p_{\text{data}}}{p_{\text{data}} + p_g} \right) dx \\ &= D_{KL}(p_{\text{data}} || (p_{\text{data}} + p_g)) + \\ &\quad D_{KL}(p_g || (p_{\text{data}} + p_g)) - \log 4 \\ &= 2D_{JS}(p_{\text{data}} || p_g) - \log 4 \end{aligned} \quad (6)$$

式(6)から、生成器の学習はJSダイバージェンスの最小化問題に帰着できることが分かる。JSダイバージェンスは非負性を持つため、 $D_{JS} \geq 0$ である。また、等式が成り立つのは確率分布が一致するときであるから、 $p_g = p_{\text{data}}$ のときであり、このときの $V(G)$ の最小値は $-\log 4$ である。

以上の内容から、識別器を最適解 $D_G^*(x) = 1/2$ の近傍で保った状態で、生成器に対してJSダイバージェンスの最小化を目指すことで、唯一の最適解 $p_g = p_{\text{data}}$ に収束することが分かる。

4.2.3 GAN の学習の流れ

式(1)をもとに、Discriminator および Generator の損失関数を L_D, L_G とすると、式(7)、式(8)のようになる。

$$L_D = \frac{1}{m} \sum_{i=1}^m \left[\log D(x) + \log(1 - D(G(z))) \right] \quad (7)$$

$$L_G = \frac{1}{m} \sum_{i=1}^m \log(1 - D(G(z))) \quad (8)$$

BCELoss(Binary Cross Entropy Loss)

交差エントロピー損失(BCELoss)は式(9)で定義される。

$$BCE(x, y) = -(y \log x + (1 - y) \log(1 - x)) \quad (9)$$

Discriminator の学習

Discriminator の学習時の概要図を図5に示す。Discriminator 学習時は Generator のパラメータ θ_g は固定する。

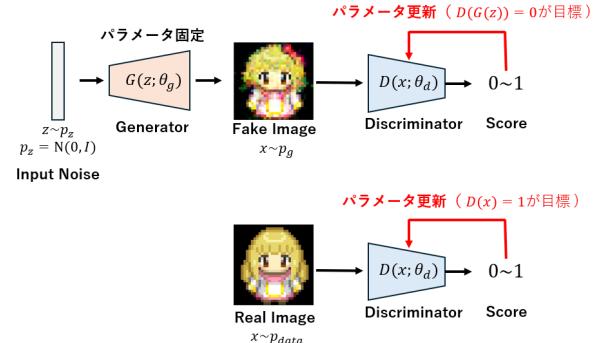


図5 Discriminator の学習時の概要図

Discriminator の目標は p_{data} と p_g を判別することであるから、学習時のラベルとして $\text{Fake}=0, \text{True}=1$ とした場合、 $D(G(z)) = 0, D(x) = 1$ となる。

式(9)を参考にすると、式(7)の損失は次式

で表せる。

$$\begin{aligned} BCE(D(x \sim p_{\text{data}}), 1) &= -\log(D(x)) \\ BCE(D(G(z)), 0) &= -\log(1 - D(G(z))) \\ L_D &= |BCE(D(x), 1) + BCE(D(G(z)), 0)| \end{aligned} \quad (10)$$

Generator の学習

Generator の学習時の概要図を図5に示す。Generator 学習時は Discriminator のパラメータ θ_d は固定する。

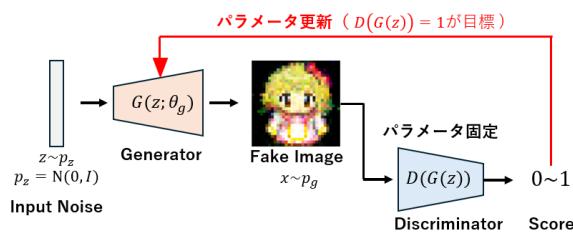


図 6 Generator の学習時の概要図

Generator の目標は $p_{\text{data}} = p_g$ として、Discriminator をだます分布を生成することであるから $D(G(z)) = 1$ を目指す。式(9)を参考にすると、式(8)の損失は次式で表せる。

$$\begin{aligned} BCE(G(z), 1) &= -\log(D(x)) \\ L_G &= |BCE(G(z), 1)| \end{aligned} \quad (11)$$

学習アルゴリズム

GANにおける学習アルゴリズムをAlgorithm1に示す。

Algorithm 1 Training of GAN

- 1: **Input:** correct Image $p_{\text{data}}(x)$, noise prior $z \sim \mathcal{N}(0, I)$, minibatch size m
 - 2: **Input:** learning rates η_D, η_G
 - 3: Initialize parameters θ_d, θ_g
 - 4: **for** iteration = 1, 2, ... **do**
 - 5: Sample minibatch $\{x^{(i)}\}_{i=1}^m \sim p_{\text{data}}(x)$
 - 6: Sample minibatch $\{z^{(i)}\}_{i=1}^m \sim p_z(z)$
 - 7: $\tilde{x}^{(i)} \leftarrow G(z^{(i)}; \theta_g)$
 - 8: $g_d \leftarrow \nabla_{\theta_d} \frac{1}{m} \sum_{i=1}^m [\log D(x^{(i)}; \theta_d) + \log(1 - D(\tilde{x}^{(i)}; \theta_d))]$
 - 9: $\theta_d \leftarrow \theta_d + \eta_D g_d$
 - 10: Sample minibatch $\{z^{(i)}\}_{i=1}^m \sim p_z(z)$
 - 11: $\tilde{x}^{(i)} \leftarrow G(z^{(i)}; \theta_g)$
 - 12: $g_g \leftarrow \nabla_{\theta_g} \frac{1}{m} \sum_{i=1}^m (1 - \log D(\tilde{x}^{(i)}; \theta_d))$
 - 13: $\theta_g \leftarrow \theta_g - \eta_G g_g$
 - 14: **end for**
-

5 本システムの構成

100次元のランダムノイズ z を入力とし、 32×32 の画像を出力する Generator を学習する。ここで、一つのネットワークで R、G、B の画像空間を学習させるために、値が固定の 100 次元のランダムノイズ v_1, v_2 を用いる。具体的には、基本となる潜在空間 z に対して、 $z + v_1, z + v_2$ の平行移動した潜在空間を用意し、 z は R、 $z + v_1$ は G、 $z + v_2$ は B の学習入力に対応付ける。固定ベクトルを用いた、新たな空間生成の理論については、補足資料1にて検証を行う。

Discriminator では、 32×32 の画像を見てそれが偽物 (G が生成) か本物 (正解画像) かの確率を 0~1 で出力する (本物:1、偽物:0)。本システムでは、カラー画像の判別精度向上のため、2種類の Discriminator を用意した。1つ目の Discriminator では、Generator の出力するグレースケール画像 (R,G,B) に対し、グレースケールの正解画像と比較を行うことでアバターの形としての正しさを評価する。2つ目の Discriminator では、「 $z, z + v_1, z + v_2$

の入力で得たR・G・B画像を用いて作成した偽物のカラー画像」と「正解カラー画像」との比較で正誤の確率を出力する。このように、Discriminatorの役割をアバターの形と色に分担を行うことで精度の向上を図った。

5.1 FPGA 実装に向けたソフトウェアの工夫

FPGA 実装では Generator ネットワークのみを実装し、100 次元の入力に対して 32×32 の画像生成を行うことを目指す。カラー画像生成を行うには R・G・B 各成分に対応した学習が必要であるが、色ごとに別々のネットワークを用いる場合、3つの重みを個別に保持・転送する必要が生じ、実装負荷が大きい。そこで、本システムでは、固定値の 100 次元ベクトル v_1, v_2 を導入し、入力 z に対して $z, z + v_1, z + v_2$ の3種の潜在空間を作成する。そして、 z をR、 $z + v_1$ をG、 $z + v_2$ をB の学習入力に対応付けることで、単一の Generator で 3 色成分の生成を学習させる(図7)。

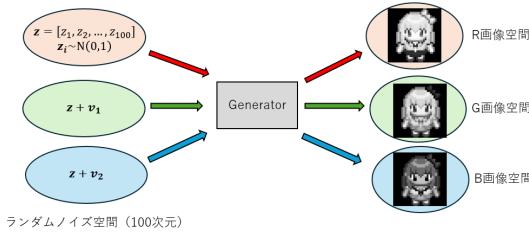


図7 固定ベクトルを用いたRGB学習

学習後は、Generator のパラメータおよび固定ベクトル v_1, v_2 を FPGA 上に実装することで、任意の 100 次元入力 z に対し、FPGA 内部で $z, z + v_1, z + v_2$ の演算を行い、それぞれ R・G・B 成分に対応した画像を生成できる(図8)。

任意の z 入力に対し、FPGA 内部で $z, z + v_1, z + v_2$ を生成することで、一つの入力および一つのネットワークからカラー画像を生成できる点が、本システムの強みである。

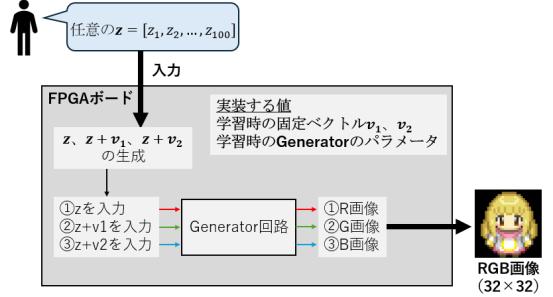


図8 FPGA 実装時のカラー化演算イメージ

5.2 データセット

キャラメル(CharaMEL0.9.0)[?]というフリーソフトを用いて、 32×32 のアバターを生成した。本ソフトでは、アバターの目、髪型、髪色、服装、アクセサリーなどの要素を任意にカスタマイズすることが可能である。本実験では、図9に示すような多様な外観を持つアバターを 275 種類生成し、これらを学習用データセットとして用いた。



図9 入力データの一部

5.3 ネットワーク構成

本システムでは、表1に示す3つのネットワークを用いて学習を行う。各ネットワークの構造については、5.3.1～5.3.3 節で詳細に説明する。

表1 各ネットワークの入出力構成

ネットワーク	入力	出力
Generator	100 次元ランダムノイズ	32×32 グレー画像 (R, G, B)
Discriminator _{grey}	32×32 グレー画像 (R, G, B)	本物・偽物の判別確率 (0~1)
Discriminator _{color}	$32 \times 32 \times 3$ カラー画像 (RGB)	本物・偽物の判別確率 (0~1)

5.3.1 Generator

Generatorの構成は、DCGAN[5]のネットワークを参考とした。Generatorのネットワーク構造を図10に示す。本ネットワークは、入力として100次元の潜在変数 z を受け取り、逆畳み込み層(ConvTranspose2d)を用いて段階的に空間解像度を拡大する構造を有する。具体的には、 1×1 の潜在特徴マップを 4×4 、 8×8 、 16×16 と段階的にアップサンプリングし、最終的に 32×32 の画像を生成する。各逆畳み込み層の後にはReLU関数を適用し、非線形性を導入することで表現能力を高めている。最終層ではTanh関数を用い、出力値を $[-1,1]$ の範囲に正規化する。

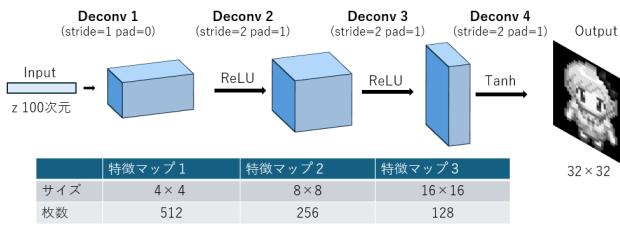


図 10 Generator のネットワーク構造

5.3.2 Discriminator(grey)

Discriminator(grey)は、 32×32 (1ch) のグレースケール画像を入力とし、畳み込み層により空間解像度を段階的に縮小しながら特徴量を抽出する。最終的に、全結合層の出力をsigmoid関数に通することで0~1のスコア（本物らしさ）を出力する。本ネットワークの構成を図11に示す。ここで、Generatorとは異なりLeaky ReLU関数およびBatch正規化を用いているが、これは DCGAN における学習安定性および性能向上を目的としたものであり、参考文献[6]に基づいている。

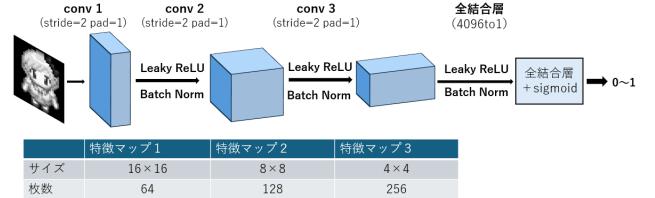


図 11 Discriminator のネットワーク構造

5.3.3 Discriminator(color)

本ネットワークは、Discriminator(grey)の入力をカラー画像(3ch)に変更することで構成される。Discriminator(grey)では、アバターの形状に注目して判別を行うのに対し、Discriminator(color)ではアバターの色の相関関係に注目して判別を行うことで、Discriminatorの判別性能を向上することを目的として構成した。

5.4 訓練パラメータ

学習に用いたパラメータを表2に示す。

表 2 学習パラメータ設定

項目	設定値
学習回数	10000
学習率(生成器)	1.0×10^{-4}
学習率(識別器)	1.0×10^{-6}
バッチサイズ	64
Adam β_1	0.5
Adam β_2	0.999
入力ノイズ次元	100

5.5 学習結果 (ソフトウェア)

学習済みGeneratorに新たなランダムノイズを入力したときの出力結果を図12に示す。図12より、Generatorはアバターの特徴を学習しており、髪型や服装の異なる多様なアバターを生成できていることが分かる。図12中の2番および5番は正解画像に近い生成結果となっている。一方で、1番および4番の服装に

見られる黄色いラインは正解データセットには存在しない。また、3番の水色のシャツにネクタイを着用したような画像も正解データセット内には見られない。これらの例から、Generatorは単なる学習画像の再現にとどまらず、アバターの特徴を保持しつつ新規性を含む画像を生成できていると考えられる。なお、6番の画像では服装がぼやけたように見えるが、これはDiscriminatorの識別性能が十分でなく、6番のようなぼやけを含む画像も正解として判別されてしまったと考える。

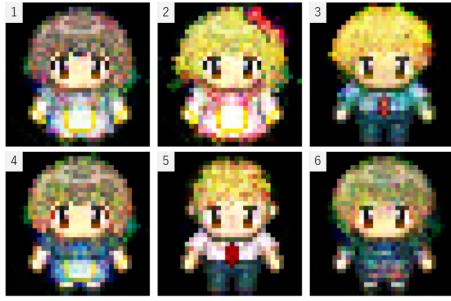


図 12 6 種のランダム入力に対する出力結果

5.5.1 ランダム空間の学習の確認

100次元ランダム入力 z は、平均0、分散1の独立な正規分布 $\mathcal{N}(0, 1)$ に従う乱数として生成される。新たにサンプリングされる z_1, z_2 も同様の正規分布から生成されている。つまり、Generatorはこの連続な潜在空間 z から画像空間への連続写像 $G(z)$ を学習することで、多様なアバター画像を生成していると解釈できる。

これを確かめるために、ランダムな2つの入力ベクトル z_1, z_2 に対し、その間を線形補間した3点の入力ベクトルから生成した画像を図13に示す。線形補間ににより得られるベクトル $z(t)$ は次式で定義される($N = 3$)。

$$z(t_k) = (1 - t_k) \cdot z_1 + t_k \cdot z_2, \quad (12)$$

$$t_k = \frac{k}{N+1}, \quad k = 1, 2, \dots, N. \quad (13)$$

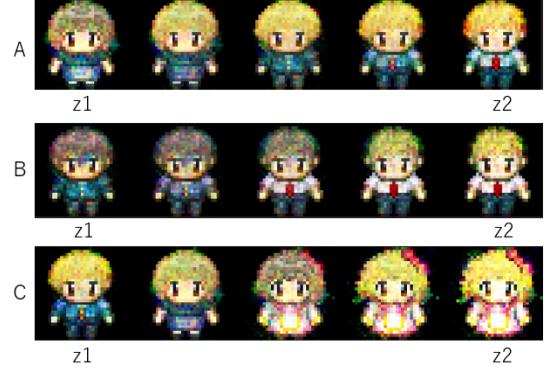


図 13 3 種のランダム入力セットに対する補間結果

図13の結果から、GANでは入力 z に対して正解画像を学習しているのではなく、 z という連続空間に対してアバターらしい画像を学習していることが分かる。また、隣接する画像を比較すると形状が似ているものが多く、画像を確率分布として学習していることが分かる。

画素単位の一致を目的とした教師あり学習では、任意の2入力における補間点に対応する出力は、それぞれの教師画像の画素値を平均化したような結果になりやすく、視覚的にぼやけた画像が生成される傾向がある。一方、GANは画像を確率分布としてモデル化し、潜在空間 z から画像空間への写像 $G(z)$ を学習するため、任意の2つの潜在変数間の補間点においても、意味的に一貫性を保った新しい画像を生成することが可能である。この性質により、GANは単なる画素補間では得られない多様な画像を生成でき、データセット拡張の観点において優位性を有するといえる。

6 CNN

本章では、GeneratorおよびDiscriminatorのネットワークとして使用したCNN(Convolutional Neural Network)について説明する。基礎となるニューラルネットワーク、誤差逆伝播法については補足資料2に記述した。

6.1 CNN の概要

CNNは画像認識のタスクにおいて優れた性能を示すことから注目される深層学習アルゴリズムである。CNNの概要図を図14に示す。画像には隣接するピクセル間に関係性があるため、CNNでは、「畠み込み層」「プーリング層」を用いて画像の局所的な特徴量（エッジや色の変化）を抽出する。分類問題では、最後に全結合層を用いることでスコアを算出する。

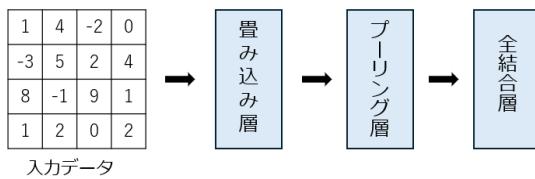


図14 CNN の概要図

畠み込み層

畠み込み層では、カーネルと呼ばれる小さなフィルタを入力データに対して畠み込むことで、局所的な特徴を抽出する。図15に示すように、複数種類のカーネルを入力データに適用することで、エッジや模様などの多様な画像特徴を捉えることができる。strideを1とした場合、カーネルは入力上を1マスクづつライドしながら、各位置において要素積の計算を行う。

プーリング層

プーリング層では、畠み込み層によって得られた特徴マップを空間的に縮小する処理を行う。代表的な手法としてマックスプーリングがあり、局所領域内の最大値を出力として

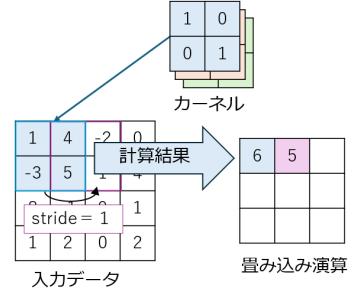


図15 畠み込み演算

採用する。(図16)

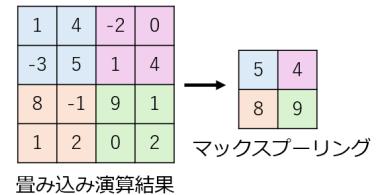


図16 マックスプーリング

6.2 逆畠み込み演算

CNNでは、情報を抽出するため畠み込み層を通過するごとに画像サイズは小さくなる。その一方で、逆畠み込み演算を用いることにより、画像を拡大していくことが可能となる。今回、Generatorは100次元の潜在変数ベクトルに対し、逆畠み込みを行うことで32×32の画像を生成する。逆畠み込みは以下の4つのステップに基づく。

1. strideに応じたデータ拡張
2. ゼロパディング
3. paddingに応じた、余白の削除
4. 畠み込み演算

STEP1：strideに応じたデータ拡張

図17に示すように、strideで指定した行数分だけ入力データのピクセル間に0を追加する。

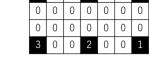
		
stride=1	stride=2	stride=3

図 17 stride に応じたデータ拡張

STEP2：ゼロパディング

STEP2では、カーネルのサイズよりも1行少ない数だけ、余白の追加を行う。

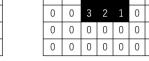
		
Kernel size 1 × 1	Kernel size 2 × 2	Kernel size 3 × 3

図 18 ゼロパディング

STEP3：padding に応じた、余白の削除

paddingで指定した行数分の余白を削除する。

		
Padding = 0 (入力データそのまま)	Padding = 1	Padding = 2

図 19 padding に応じた、余白の削除

STEP4：畳み込み演算

図15と同様の、通常の畳み込み演算を行う。

6.3 本システムの逆畳み込み回路

本システムの逆畳み込み回路は、図20のような構成をしており、 4×4 のサイズのカーネルに対して、各パラメータに基づいた演算を行うことにより、 $1 \times 1 \rightarrow 4 \times 4 \rightarrow 8 \times 8 \rightarrow 16 \times 16 \rightarrow 32 \times 32$ とスケールアップしていく。

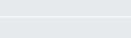
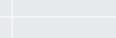
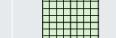
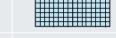
層	各層の入出力関係 (特徴マップの数)	特徴マップ サイズ	パラメータ
Deconv1	 ... 	1 × 1	stride=0 padding=1 kernel数 100×512
	 ... 	4 × 4	stride=2 padding=1 kernel数 512×256
Deconv2	 ... 	8 × 8	stride=2 padding=1 kernel数 256×128
	 ... 	16 × 16	stride=2 padding=1 kernel数 128×1
Deconv3	 ... 	32 × 32	
Deconv4	 ... 		

図 20 逆畳み込み回路の入出力関係

一つのカーネルと特徴マップの逆畳み込み計算

例として、Deconv2における $4 \times 4 \rightarrow 8 \times 8$ の計算を図21に示す。カーネルサイズは 4×4 である。まず、stride=2より図17のデータ拡張を行うことで 7×7 になる。次に、STEP2の余白の追加は「カーネル数-1 = 3」の0が追加される。STEP3ではpadding=1より、1行分の0が削除され、 11×11 となる。STEP4で通常の畳み込み演算を行うことにより、 8×8 の特徴マップが生成できる。

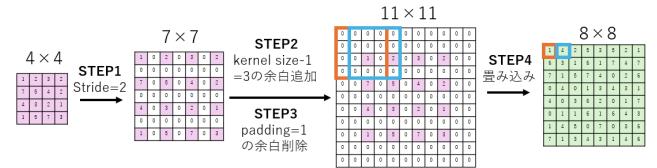


図 21 Deconv2 における $4 \times 4 \rightarrow 8 \times 8$ の計算例

一層ごとの逆畳み込み演算

本節では、一層ごとの逆畳み込み層ではどのような演算が行われているかを説明する。本システムにおける各層ごとの入出力およびカーネル行列の関係を図22に示す。

図22の計算関係から分かるように、各層では入力input[1][i]とカーネルkernel[i][k]の行列

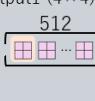
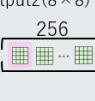
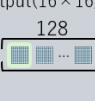
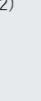
層	入力	カーネル	出力
Deconv1	input1 1 [□ □ ... □]	\ast^T 	output1 (4×4) 1 [■ ■ ... ■] = 1 [■ ■ ... ■]
Deconv2	input2 (=output1) 1 [■ ■ ... ■]	\ast^T 	output2 (8×8) 1 [■ ■ ... ■] = 1 [■ ■ ... ■]
Deconv3	input3 (=output2) 1 [■ ■ ... ■]	\ast^T 	output (16×16) 1 [■ ■ ... ■] = 1 [■ ■ ... ■]
Deconv4	input4 (=output3) 1 [■ ■ ... ■]	\ast^T 	tanh = 

図 22 各層ごとの入出力およびカーネル行列の関係

積を計算することで、出力 $output[k]$ を得ている。 j 番目の出力を得るには、 j 列のカーネル $kernel[i][j]$ と入力 $input[1][i]$ の行列積をとる。つまり、出力の一要素を計算したい場合は、カーネルは該当する一列分の要素のみが必要となる。具体的な計算を次節で説明する。

Deconv1 の逆畳み込み演算例

本節では、Deconv1の逆畳み込み演算を例として、各層の演算の流れを示す。Deconv1では、100次元の入力 $input[1][i](i = 1, 2, \dots, 100)$ に対して、512次元の $output[k](k = 1, 2, \dots, 512)$ を出力する。各 $output$ に対する計算は列ごとに行うため、1列目の計算を例にする。その概要図を図23に示す。

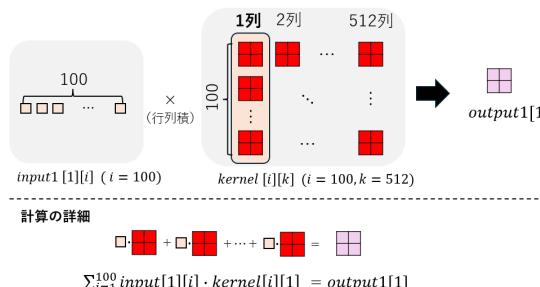


図 23 Deconv1 の 1 列目の計算例

図23のように、一列目の計算は式(14)となる。

$$\sum_{i=1}^{100} input[1][i] \cdot kernel[1][1] = output1[1] \quad (14)$$

これで1列分の計算が完了するため、同様の計算を残りの511列に対して行う(図24)。つまり、Deconv1層における畳み込み演算は式(15)となる。

$$\sum_{k=1}^{512} [\sum_{i=1}^{100} input[1][i] \cdot kernel[1][k]] = output1[k] \quad (15)$$

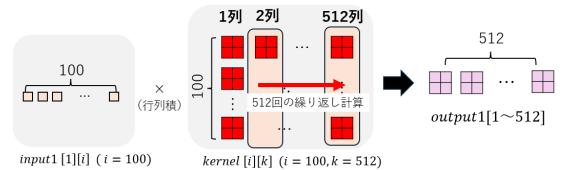


図 24 Deconv1 の全列の計算例

7 エミュレータにおける、固定小数点の検討

本章では、FPGA実装を想定し、固定小数点表現におけるビット幅設計の工夫により、精度を維持しつつビット幅を最小化したエミュレータの構築手法について検討する。

7.1 カーネル(重み)の固定小数点の検討

表3に各層のカーネル(重み)の最大値・最小値を示す。表3から、整数部はないことが分かるから、符号部1bit、小数部7bitと考え、int8(Q0.7)で重みの保存を行う。Qa.bは整数部のビット数がa、小数部のビット数はbであることを示す。Q0.7によって表現できる範囲は $-1 \sim 0.992(1 - 2^{-7})$ であり、分解能は1/128である。この範囲がint8の-128～127にマッピングされる。具体的に、実数値で0.6046は $77.38 \approx 78$ (四捨五入)に対応する。以後の計算では、16bitとしてint16を用いるが、重みパラメータはリソースの確保および計算精度を担保できることから8bitで表現した。

表3 各層カーネルの最小値・最大値

Layer	Shape	Min	Max
net0	(100, 512, 4, 4)	-0.7249	0.6046
net1	(512, 256, 4, 4)	-0.7105	0.2585
net2	(256, 128, 4, 4)	-0.4607	0.2120
net3	(128, 1, 4, 4)	-0.2351	0.1872

7.2 各層の出力に対する固定小数点の検討

任意のランダム入力に対する、各層の出力(output1～4)の実数値の最大値・最小値を表4に示す。表4から整数部を6bit持たせると、-64.000～63.998まで表せるため十分な範囲であるといえる。しかし、今回の値は任意の入力に対するものであるから、頑健性を考慮して、Q7.8(int16)での実装を行った。Q7.8の実数の表現範囲は-128.000～127.996であり、分解能は1/256である。これをint16

表4 各層出力の最小値・最大値

Layer	Min	Max
output0	-13.46	12.44
output2	-47.34	20.94
output3	-47.62	14.70
output4	-8.46	11.23

の-32768～32767の範囲にマッピングする。int8(Q0.7)からint16(Q7.8)の変換については、小数部が一桁増加することから、int8の値に 2^1 を掛けることでint16の値を得た。例として、実数値で0.6046はint8で $77.38 \approx 78$ であり、これに2を掛けると156(int16)である。これを実数値に戻すには、 $2^8 = 256$ で割ると0.609となり元の実数値に近い値となることが分かる。

7.3 各逆畳み込み演算の乗算における、固定小数点の検討

前節で示したように、計算時には基本的にint16(Q7.8)が用いられる。しかし、畳み込み演算の際に乗算が含まれるため、Q7.8 × Q7.8 = Q14.16となりint16の範囲ではオーバーフローしてしまう。そこで、掛け算の結果のみint32(Q14.16)で保存し、計算後の結果に対し 2^8 で割った値をint16(Q7.8)に格納した。実際に、「 $0.6046 \times 0.6046 = 0.36554116$ 」の計算を例に挙げると、int16では「 $155 \times 155 = 24025$ 」となる。24025の値を小数部16bitをもつ値とするとQ7.8の精度に戻すには 2^8 で割る必要がある。よって、「 $24025 \div 256 = 93.84 \approx 94$ 」となる。int16(Q7.8)で94であるから、これを実数に戻すと「 $94 \div 256 = 0.3671875$ 」であり、実数で計算した値に近い値となることが分かる。

7.4 tanh テーブルにおける固定小数点の検討

output4の値はtanhを掛けた後、256階調にして画像として表示する。このtanhの入力を16bitにする場合、高精度にはなるがテーブル作成成分のメモリ消費が大きくなる。そこで、

8bitの入力でtanhテーブルを作成する方法を考えた。

まず、output4の出力は表4より、 ± 10 の範囲程度であるから、整数部を5bit、小数部を2bitとしてQ5.2を用いると8bitで表現できることが分かる。Q5.2の範囲は、 $-32 \sim 31.75$ で分解能は $1/4$ である。Q7.8からQ5.2に変換するには、int16の値を 2^6 で割ればよい。これによって、ここまで計算結果をint8(-128～127)の範囲にマッピングできる。

tanhは入力された値を $-1 \sim 1$ の範囲に非線形変換するが、今回int8を用いるため、-128～127の範囲にマッピングするようなtanhテーブルを作成する。int8の入力に対し、int8の出力を返すtanhテーブルをプロットすると図25となる。このtanhの出力に対し、+128を足し合わせることで0～255の256階調となる。今回は演算数を減らす目的で、図25の値に+128を足したものとtanhテーブルとして実装した。

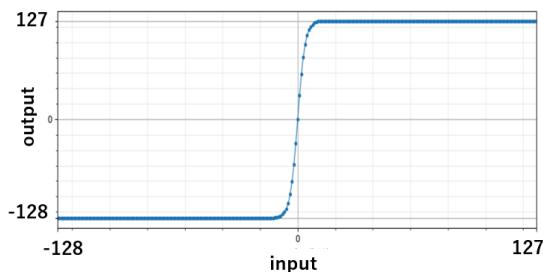


図 25 tanh テーブル (int8 対応)

7.5 固定小数点の検討のまとめ

以上の内容をまとめた概要図を図26に示す。ポイントは、リソース確保のため、重みパラメータをint8で保存したこと。基本的な演算はint16(Q7.8)だが、乗算部分はint32(Q14.16)で確保したこと。tanhテーブルの出力を-128～127ではなく、0～255にすることにより、画像変換までを一気通貫で行うテーブルを作成したことである。

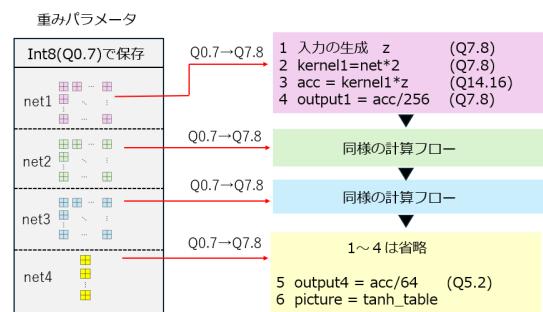


図 26 固定小数点の検討のまとめ

8 使用機器について

本システムのハードウェア実装には、AMD ZynqTM UltraScale+TM MPSoC ZCU104 Evaluation Kitを用いた[7]。また、AMDの提供するPython productivity for Zynq(PYNQ)[8]と呼ばれるオープンソースプロジェクトを活用し、ブラウザ上からJupyter Notebookを通じてFPGAボードを動作させるシステムを構築した。本節では使用機器についての解説と仕様説明を行う。

8.1 Zynq UltraScale MPSoC の概要

Zynq UltraScale MPSoCとは、ARMプロセッサ(PS)とFPGA(PL)を1チップに統合したSoCである。PS部、PL部それぞれの仕様は表5, 6のようになっている。本ボードの最大の特徴は、ソフトウェアとハードウェアを同一チップ上で高速かつ低遅延に動作させられる点である。今回路ではPL部とPS部間の通信にAXIインターフェースの中でも扱いやすいAXI4-Liteを用いており、APUとFPGA間の通信が可能となっている。

表5 PL部の仕様

System Logic Cells	504,000
CLB Flip-Flops	460,800
CLB LUTs	230,400
Block RAM Blocks	312
Block RAM	11 [Mb]
DSP Slices	1,728
Max. HP I/O	416
Max. HD I/O	48

8.2 PYNQ の概要

Python productivity for Zynq(PYNQ)は、Linux上で動作するPython APIを用いてZynq SoCのPL部を制御・利用するためのフレームワークである。本フレームワークを用いるこ

表6 PS部の仕様

APU	Dual-core Arm Cortex-A53
Architecture	Arm v8-A
OS	PYNQ Linux
Framework	PYNQ
Language	Python

とにより、以下のような利点がある。

- Libraryが豊富なPythonを利用して、PL部の制御を行える
- Linux環境下で動作させられるため、柔軟性のある開発環境が実現可能
- Webブラウザ上でJupyter Notebookを利用した直感的な操作が可能

このように、PYNQを用いることで迅速な開発やPythonを利用できる幅広いユーザに対して製品を提供することが可能となる。また、評価ボードをインターネットに接続することで、別PCからJupyter Notebookを介してアクセスすることが可能であり、評価ボードの遠隔操作やファイル転送を容易に行える。そのため、本フレームワークはIoTシステムとしての実用化にも適している。このことから本システムにおいても、システムの実用化に向けてPYNQを利用した開発を行う。

9 システム構成と動作説明

本章では設計したシステムのアーキテクチャからPS部、PL部それぞれの構成、およびその動作について説明する。

9.1 システムのアーキテクチャ

本システムのアーキテクチャは図28のようになっている。Block RAM等のリソース量を考慮し、エミュレータで実装した学習したパラメータを使用して生成部のみの実装を行った。8章でも記述した通り、本システムは大きく分けてPS部とPL部に分かれている。PS

部は、保存された入力データおよび重みをPL部に送る役割を担う。またPL部の動作を制御する制御信号を送ることで動作モードを遷移させ、PL部の動作を処理の進行に合わせて適切に変化させる役割も担っている。一方、PL部はPSとPL間の通信を担うAXI-LITE、PS側から送られたデータをGeneratorに適切な形、タイミングで転送するモジュール、そしてGeneratorの大きく分けて3つの回路で構成されている。これらのシステム全体の動作概要は以下の通りである。

1. AXI-LITE形式でPS側からPL側に図中 Input Vecotrで示される100次元入力を送信する
2. AXI-LITE形式でGeneratorの入力用RAMにデータ転送する開始信号 (Control signal) を送る
3. AXI-LITE形式でPS側からPL側に推論に必要な重みのうち、1層分のさらに1出力要素を得るために必要な重み (Weight of 1-4 layersの一部) を送信する
4. AXI-LITE形式でGeneratorの重み用RAMにデータ転送する開始信号を送る、PSは演算が終了するまで待機する
5. 1出力を得るために必要な入力 x および重み w を取得した後、Generatorは計算を開始する
6. RAMに保存された重みを使い切ると、Generatorが1出力の演算終了信号を送信し、図中end_signalとしてAXI-LITE形式でPS側が受け取る
7. 3 - 6の手順を繰り返し、1層分の出力を得る。この時、1層分の演算終了信号がPS側に送信される
8. さらに3 - 7の手順を繰り返すことで4層

分の演算を行う。この時、演算終了信号が送信される。

9. 4層分の演算終了後、AXI-LITE形式でPL側が出力したアバター画像をPS側に送信する

特に手順3-6については、図27で補足説明を行う。エミュレータで実装したCNNモデルのうち、特に1層目、2層目は図27のようになっている。1層目では100次元の入力ベクトルに対して 4×4 のカーネルを 512×100 要素掛け合わせることで出力を獲得し、これにReLU関数を適用したものを2層目の入力として用いる。このような試行を4層分繰り返すことでアバター画像を出力することができる。本回路では特に、それぞれの層の出力の1要素を求めるために必要な重みである図27中の1paked weight (1×100 要素)のみをPS部からPL部に送信する。そして、この重みと入力の掛け算を行う試行をそれぞれの要素の出力要素分行うことで1層分の計算を終了する回路とした。本回路構成を採用した理由と解説については、11章のハードウェア上の工夫点にて詳しく解説する。

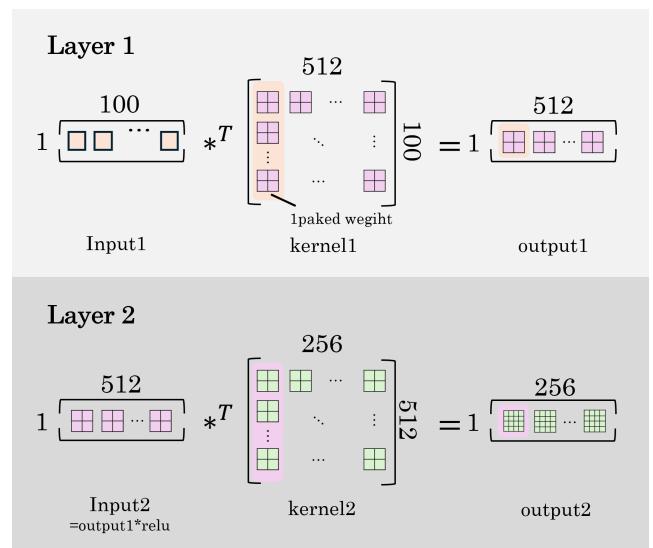


図 27 計算手順の概要

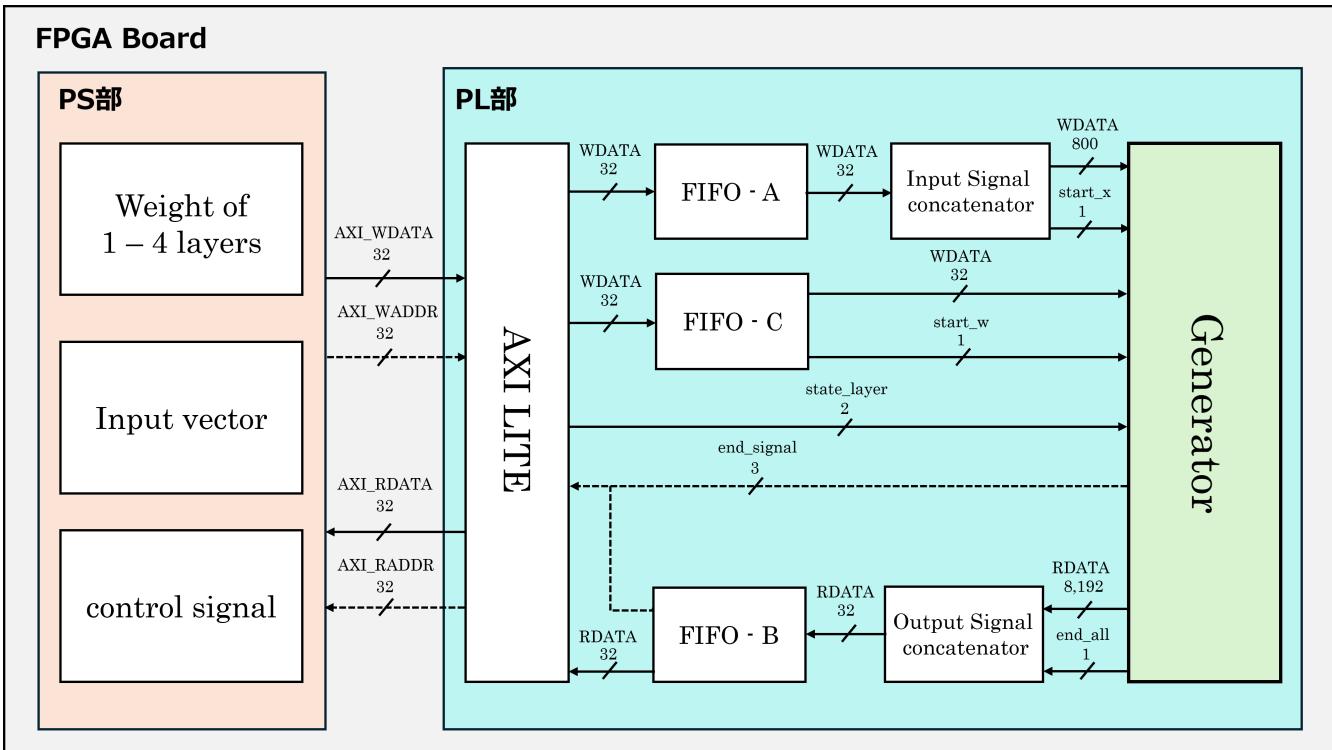


図 28 システムのアーキテクチャ

9.2 PL 側の回路設計

AXI-LITE

本回路では、AXI-4という規格の中でも AXI-LITEを用いた通信方式を採用した。AXI 規格とは、ARM社が使用を策定したバスプロトコルであり、開発者側は共通のインターフェースを用いてPSとPL間の通信を行う。特にAXI-LITE Moduleの回路構成を図29に示す。図29からわかるようにAXI-LITEはアドレス指定により異なる制御を行うことができる利点がある。本回路では特に表7のようにアドレスマップを作成し、PSからAXI-LITEを通じて細かくPLの制御を行えるような設計とした。この設計により、前述した複雑な計算手順をPythonを利用して極めて正確かつ簡単にを行うことができるような設計となっている。具体的に1つの層を例にとって、実際の動作手順を説明する。

1. 100次元入力を送信するため、slv_reg0に

00001をセットする（リセット信号はアクティブロー）

2. GeneratorのRAMに入力信号を読み込ませるため、計算開始信号を立てる（slv_reg0に00111）
3. 計算に必要な重みを送信するため、slv_reg0に00101をセットする。
4. GeneratorのRAMに重みを読み込ませるため、計算開始信号を立てる（slv_reg0に00111）
5. 重みを使い切るまでPS側は動作を止める（slv_reg1が100となるまで待機）
6. 1-5の操作を512回繰り返したとき、slv_reg1が110となり2層目の処理に移行する。

このように、PL側の動作をPS側から可視化できるようにすることでUIを用いた計算の進捗状況表示や計算手順の柔軟性が生まれるように設計を行った。本実装は、アドレス指

定により制御に幅を持たせることができるAXI-LITEならではの制御法といえる。

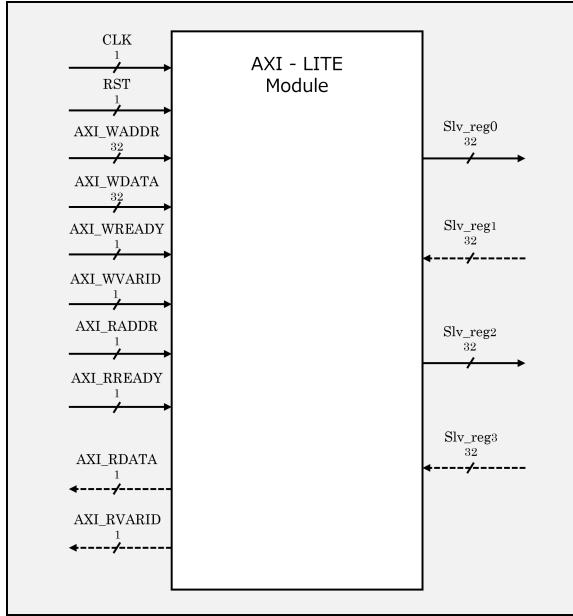


図 29 AXI-LITE module の構成

表 7 AXI-LITE の制御信号

slv_reg0	値	slv_reg1	値
リセット信号	0	全計算終了	0
計算開始信号	1	1層計算終了	1
FIFOA,C 選択	2	1出力計算終了	2
計算層選択	3-4		

Input, output Signal concatenator

Input signal concatenator, Output signal concatenator はそれぞれ図 30 のようにあらわされる計算回路である。Input Signal concatenatorでは、Generatorに対して必要な入力データを1CLKで渡すことを可能とするためにFIFO-Aから送られる32ビット幅の信号を繋げて1つの信号として送信する。一方、Output signal concatenatorではGeneratorからPS側に対して送られる $32 \times 32 \times 8$ bit の信号をそれぞれ32ビットの信号に分けることで、本回路における AXI-LITE 通信方式に

あった形でPS側にデータ送信できるように加工する回路である。

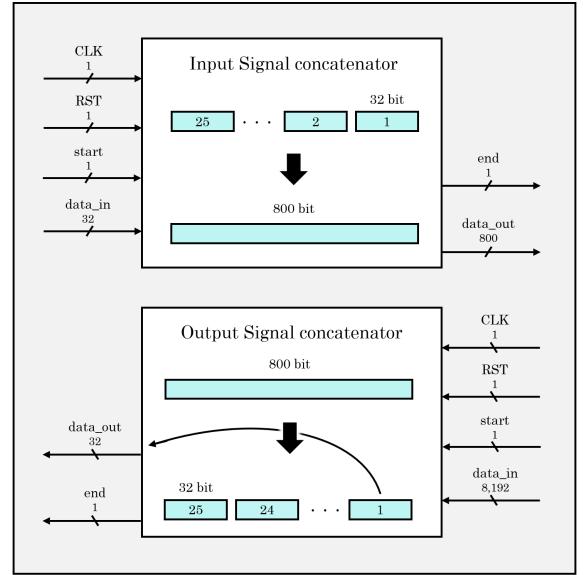


図 30 Input, output Signal concatenator の構成

Generator

エミュレータで学習を行ったパラメータと計算精度をもとにして、100次元入力から 32×32 のアバター画像を生成する計算部分である。次節にて詳しい説明を述べる。

10 計算回路の設計

本章では、提案するGANアバター生成システムの核となる計算回路のFPGA実装について、その詳細な設計内容を記述する。

10.1 計算回路の全体アーキテクチャ

本システムにおける Generator 回路は、画像生成のための大規模な転置畳み込み演算を効率的に実行するため、明確な役割分担を持つ複数のモジュールによる階層構造を採用している。計算モジュールの全体階層構造および、各メモリと演算コア間の詳細なデータフローを図31に示す。

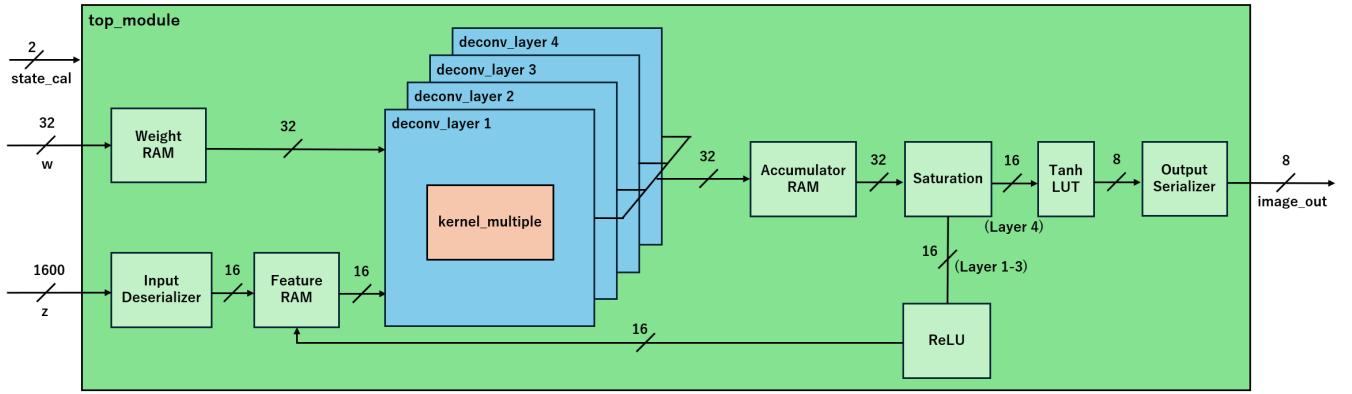


図31 計算モジュールの階層構造と詳細データフロー図

10.1.1 主要モジュールの機能

図31に示した全体のデータフローを解説する前に、本項ではシステムを構成する主要な8つのハードウェアモジュールの役割について定義する。

- Input_Deserializer:** 外部から1クロックで一括して供給される100次元のランダムデータ（16bit × 100要素）を受け取るインターフェースである。本モジュールは、1600bitの並列データを16bit単位に分割し、それぞれ演算用の16bit幅のFeature_RAMに順次転送する。
- Feature_RAM / Weight_RAM:** 演算に必要なデータを保持するメモリである。Feature_RAMは入力特徴マップ（16bit）を保持し、Weight_RAMは学習済みの重みパラメータを保持する。特にWeight_RAMでは、データ転送効率向上の観点から、 4×4 のカーネルのうち 2×2 のカーネル要素（4つ）を1つのアドレスに集約して保持する仕様にした。そのため、8bitの重みデータを4つ結合した32bitパケットとして管理する。
- deconv_layer:** 本システムの演算の中核を担うモジュールである。第1層から第4

層までの転置畳み込み演算を担当し、メモリからデータを読み出して積和演算を実行する。

- Accumulator_RAM:** 積和演算の途中経過を保持する32bit精度の累積加算用メモリである。
- Saturation:** Accumulator_RAMから読み出された32bitの演算結果に対し、飽和演算を行いながら16bit幅へ丸め処理を行うモジュールである。
- ReLU:** 中間層（第1～3層）において使用される活性化関数モジュールである。負の値を0にする非線形処理を行う。
- Tanh_LUT:** 最終層（第4層）において使用される活性化関数モジュールである。双曲線正接関数の値を格納したルックアップテーブルを参照することで、複雑な非線形計算を回避しつつ、データを画像の画素値に適した8bit範囲へ変換する。
- Output_Serializer:** 最終的な生成画像を外部へ出力するインターフェースである。並列データを1画素ずつ直列化し、image_outとして出力する。

10.1.2 全体のデータフロー

前項で定義したモジュール群は、最上位階層であるtop_moduleによって統合管理され、データは以下のフローに従って処理される。

まず、外部より供給されたランダム入力 z は、Input_Deserializerによって個別に切り分けられ、Feature_RAMに格納される。その後、重み値 w もWeight_RAMにロードされる。

演算フェーズでは、各層に対応するdeconv_layerが順次起動される。deconv_layerは、Feature_RAMおよびWeight_RAMから必要なデータを読み出し、内部演算コアkernel_multipleへ供給する。演算された積和結果は、Accumulator_RAMに対して順次加算され、畳み込み結果が形成される。

1層分の演算が完了すると、データはAccumulator_RAMから読み出され、Saturationで16bitに圧縮される。その後、中間層の場合は、データは活性化関数ReLUを経て、次層の入力特徴マップとして再びFeature_RAMへ書き戻される。一方、最終層の場合、データはTanh_LUTに入力され、画素値として整形された後にOutput.Serializerを介して画像データimage_outとして外部へ出力される。

このように、層によってデータの行き先を切り替える構成をとることで、限られた回路リソースを有効活用しながら、スムーズな処理を実現している。

10.1.3 全体制御ステートマシンの設計

本システムのtop_moduleは、入力から出力に至る一連の処理を効率的に制御するため、ステートマシンによって管理されている。主要な状態遷移と制御信号の流れを図32に示す。

本ステートマシンはアイドル状態(Idle)から始まり、Input_data_controlからの制御信号start_xの立ち上がりエッジを検出すること

で動作を開始する。動作は大きく分けて以下の手順で進行する。

1. **入力値の展開 (Load_Input)** : start_xを受信すると、Input_Deserializerが起動する。ここでは一度に受け取った並列データを、クロックに同期して順番に切り分けてFeature_RAMへ書き込む。
2. **入力値の格納待機 (Wait 1)** : Load_Inputから来た場合は、全データの格納が完了するまで待機する。Move_Dataから来た場合は、Feature_RAMに対して全データの転送が完了するまで待機する。
3. **重み値のロード (Load_Weights)** : 外部からのstart_w信号がHighになっている期間、ステートマシーンは毎クロック32bit幅の重み値をそのままWeight_RAMに格納する。
4. **重みロード待機 (Wait 2)** : start_wがLowになり、データの転送期間が終了するまで待機する。
5. **チャネル演算実行 (Compute_Channel)** : deconv_layerに対して開始信号startを発行し、1出力チャネル分の積和演算を実行する。
6. **演算待機 (Wait 3)** : deconv_layerからの完了通知doneを受信するまで待機する。
7. **チャネルループ判定 (Check_Loop 1)** : 同一層内の演算継続可否を判定する。
 - **チャネル継続**: 未計算の出力チャネルが残っている場合、start_w信号に従い、次の重みをロードする。
 - **層完了**: 層内の全チャネル計算が完了している場合、完了信号layer_doneに従い、次の判定ステートへ移行する。
8. **レイヤーループ判定 (Check_Loop 2)** :

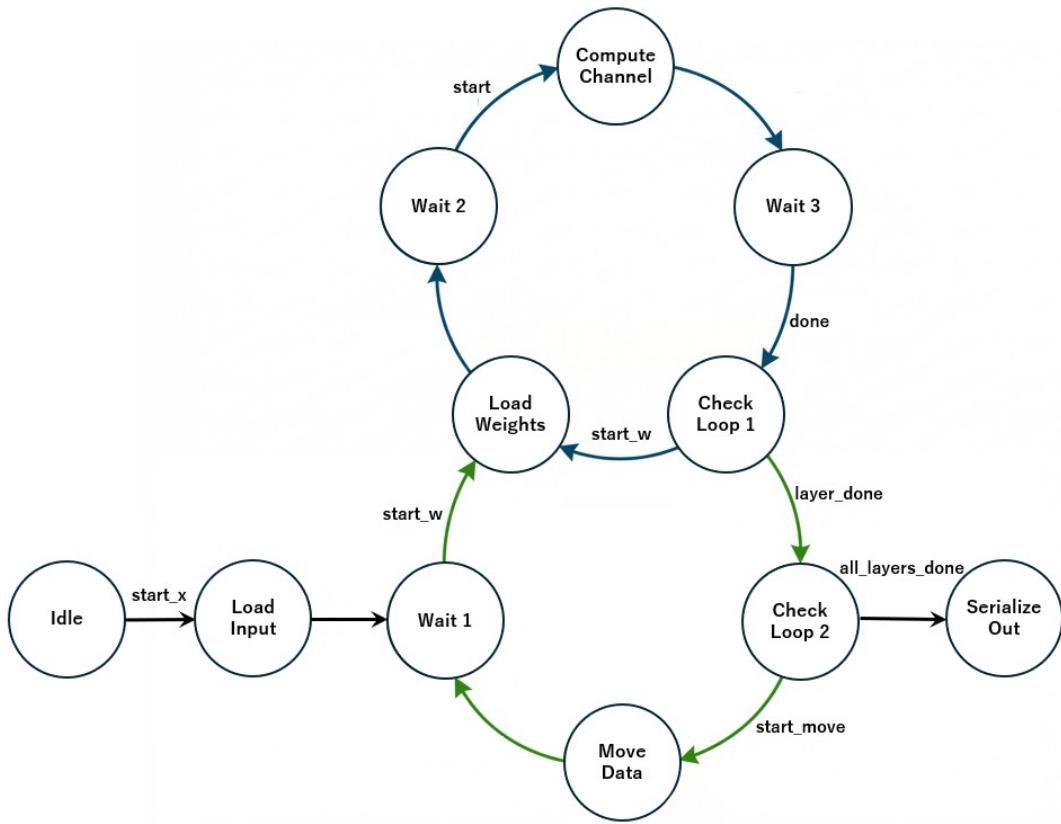


図 32 全体制御ステートマシンの簡略化状態遷移図

全層の進捗状況に応じた分岐を行う。

- **データ更新（中間層）**：最終層でない場合、信号start_moveに従い、Move_Dataへ遷移する。
 - **全完了（最終層）**：第4層の演算が完了している場合、完了信号all_layers_doneに従い、Serialize_Outへ遷移する。
9. **データ更新（Move_Data）**：Accumulator_RAMにある演算結果を1クロックごとに読み出し、SaturationおよびReLU活性化関数を通してFeature_RAMへ書き戻す。その後は、次層の重みをロードして演算を行う。
 10. **最終出力（Serialize_Out）**：Accumulator_RAMから演算結果を読み出し、SaturationとTanh_LUT、Output_Serializerを通して、出力用レジスタimage_outに格納される。全画素の格納が完了した時点で、画像データ全体がまとめて外部へ出力される。

タ全体がまとめて外部へ出力される。

10.2 転置畳み込みユニット (deconv_layer) の実装

本システムにおける転置畳み込み演算の中核を担うのがdeconv_layerモジュールである。このモジュールは、上位システムtop_moduleと演算コアkernel_multipleの間に位置するメインコントローラとして機能する。本モジュールの概略図を図33に示す。

10.2.1 各サブモジュールの機能

本モジュールは、役割の異なる4つのサブモジュールによって構成されている。

1. **Controller:** ユニット全体の動作を統括する最上位コントローラ。初期化フェーズでは、次の演算結果が格納されるAccumulator_RAMの領域を順次走査し、クロック同期で0を書き込む。演算

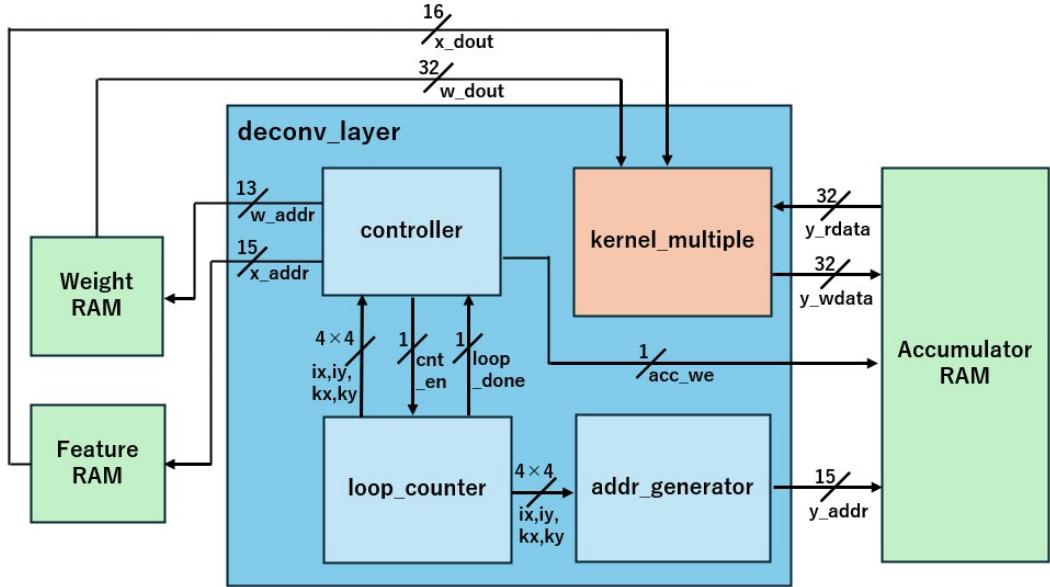


図 33 deconv_layer の内部構成

フェーズでは、loop_counterから供給される座標情報 (ix, iy, kx, ky) を、読み出しアドレス (x_addr, w_addr) へと変換してFeature_RAMとWeight_RAMへ出力する。また、kernel_multiple演算結果を確定させるタイミングでAccumulator_RAMへの書き込み許可信号 (acc_we) を発行する重要な役割を担う。1チャネルの演算が完了した際には、上位へ終了信号 doneを発行する。

2. **loop_counter:** 1つの出力チャネルの演算に必要な4重ループ（入力画像の走査およびカーネルの展開）を管理する座標生成カウンタ。Controllerからのカウント許可信号cnt_enがHighの期間のみカウントを進める設計となっており、特定のタイミングで座標出力を保持することが可能。各ループの最大値は層のパラメータに応じて設定されており、全走査が完了すると完了信号loop_doneが自動で生成される。
3. **addr_generator:** loop_counterから供給される入力画像の座標 (ix, iy) とカーネル座標 (kx, ky) を基に、転置畳み込みの

ストライド (S) およびパディング (P) を考慮した出力座標を算出し、出力画像上的一次元アドレス (y_addr) に変換する。そして、そのアドレスをAccumulator_RAMへ出力する。

4. **Accumulator_RAM:** 積和演算の途中経過を保持する32bit精度の累積加算用メモリ。書き込み許可信号 (acc_we) がHighの期間のみ書き込みが有効で、Lowの時にアドレスを送ると格納された蓄積値が読み出される。蓄積値に新しい乗算結果を足し合わせ、再び同じ場所へ上書きして保存することで、累積計算を実現する。
5. **kernel_multiple:** Feature_RAMから来る入力値 x_dout 、Weight_RAMから来る重み値 w_dout 、Accumulator_RAMから来る蓄積値 y_rdata を基に、積和演算を行う演算コア。演算結果 y_wdata はAccumulator_RAMに出力する。

10.2.2 deconv_layer 内におけるデータフロー

deconv_layer 内部では、Controllerを中心には各モジュールが双方向に信号をやり取りすることで、一連の演算プロセスを進行させる。そのステートマシンを図34に示す。

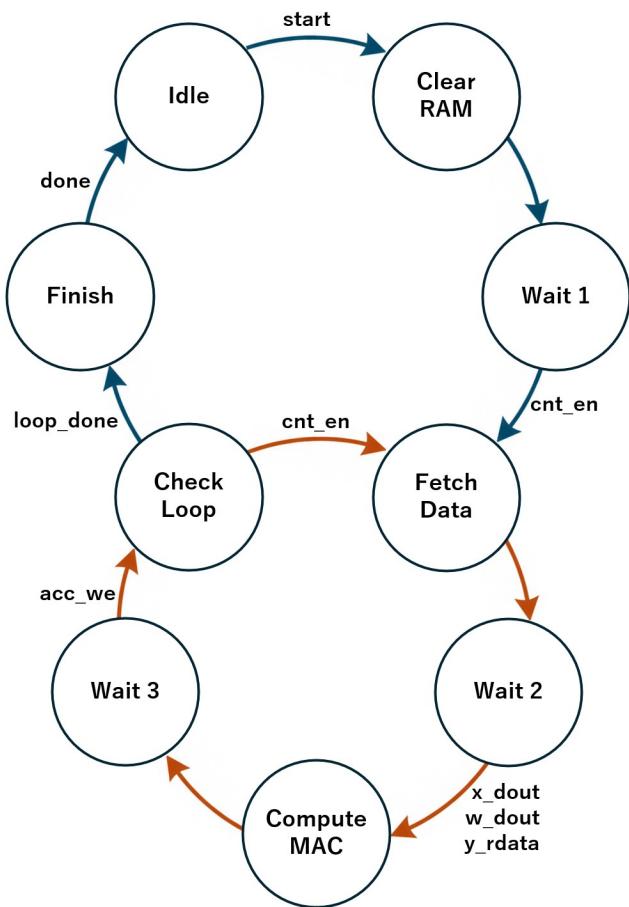


図 34 層内制御ステートマシンの状態遷移図

ステートマシンは以下の手順で動作する。

1. **待機 (Idle)** : 上位からの start 信号を待機する。
2. **初期化 (Clear_RAM)** : Controllerが主導して Accumulator_RAM の領域を初期化し、演算の準備を整える。
3. **初期化待機 (Wait 1)** : メモリのリセットが完了するまで待機する。
4. **データ読み出し (Fetch_Data)** : Controller

が cnt_en を立ち上げ、loop_counter が座標信号を生成する。生成された座標信号は二手に分かれ、一方は Controller で読み出しあドレスに変換され、Feature_RAM と Weight_RAM に出力される。もう一方は addr_generator で演算結果の格納先アドレスに変化され、Accumulator_RAM に出力される。

5. **読み出し待機 (Wait 2)** : kernel_multiple に対して、Feature_RAM と Weight_RAM からは入力値と重み値、Accumulator_RAM から蓄積値が読み出されるまで待機する。ここで cnt_en は立ち下げ、次の Fetch_Data まで loop_counter は停止する。
6. **積和演算実行 (Compute_MAC)** : 供給された入力値・重み値は kernel_multiple 内で乗算され、蓄積値と加算される。このようにして 1 座標分の積和演算が実行される。
7. **演算完了待機 (Wait 3)** : 積和演算が終わると、加算結果 y_wdata は、Controller が制御する書き込み許可信号 acc_we に同期して、再び Accumulator_RAM の同一アドレスへと書き戻される。
8. **ループ判定 (Check_Loop)** : loop_counter から完了信号 loop_done を確認する。1 チャネル内の全画素の走査が完了していれば Finish へ遷移し、未走査要素があれば Fetch_Data へ戻る。
9. **完了通知 (Finish)** : 上位へ終了信号 done を通知し、Idle へ復帰する。

10.3 演算コア (kernel_multiple) の実装

kernel_multiple は、deconv_layer の制御下で動作する演算モジュールである。その内部構成を図35に示す。

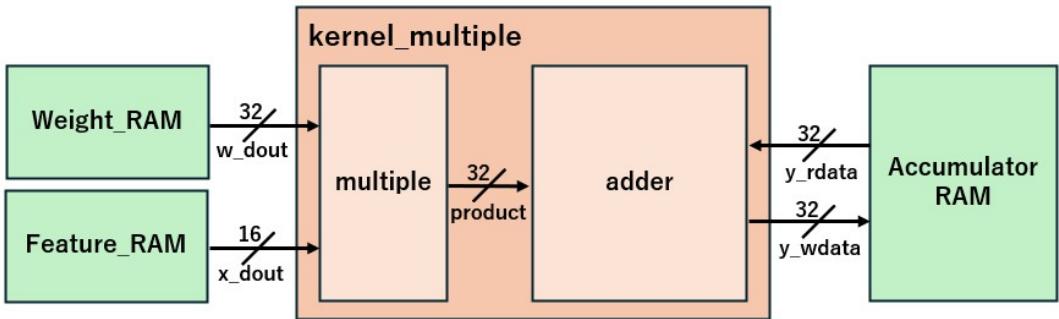


図 35 kernel_multiple の内部構成

10.3.1 各サブモジュールの機能

本モジュールは内部に以下の主要なサブモジュールをインスタンス化している。

1. **multiple:** 入力値と重み値の乗算を行う。本設計では、メモリ帯域を有効活用するため、 4×4 のカーネルのうち 2×2 のカーネル要素（4つ）に相当する 32bit 幅（8bit \times 4）の重み値が一括して供給される。そのため、座標信号に基づいたバイト選択ロジックにより、必要な 8bit の重み値を抽出した上で 16bit への精度拡張を行い、入力値との乗算を実行する。
2. **adder:** 乗算結果を Accumulator_RAM の蓄積値に累積する。32bit 幅の加算器を実装しており、読み出された蓄積データと現在の乗算結果を合算する際のオーバーフローを防ぎ、演算精度を維持する役割を持つ。

10.3.2 kernel_multiple 内におけるデータフロー

kernel_multiple 内部では、上位の Controller によって送られてきた入力値と重み値と蓄積値を用い、1 座標ごとに演算を実行する。

まず multiple で 32bit 幅の重み値から、現在の演算座標に対応する特定のバイトデータが

抽出される。この重み値と入力値による乗算結果 product は、adder において蓄積値と合算される。その合算値 y_wdata は、新たな蓄積データとして即座に元の同一アドレスへと書き戻される。

この動作を繰り返すことで、1 出力チャネル分の演算が完結する。

11 FPGA 実装の中で工夫した点

本章では特に FPGA 実装を行うにあたって工夫した点についてまとめる。

11.1 Block RAM の制限を考慮した回路構成

本システムのアーキテクチャとして、9 章で示した計算手順をとった理由について詳しい説明を行う。仮に本システムの計算に用いるパラメータや入力要素をすべて FPGA ボードの Block RAM に保存する形式をとった場合のメモリ使用量の概算は表 8 のようになる。なお、使用した ZCU104 に搭載される Block RAM 1 つにつき 36kB のメモリ量を持つことをもとにして計算を行った。この時、実装に必要な Block RAM の個数は 7,943 個となるため表 5 に示した ZCU104 に搭載されている 312 個の Block RAM では到底貯うことができないリソース量であることがわかる。そこで私たちのチームでは、PL 部に 1 層の計算の中でも特に 1 出力要素に必要な重みのみを保存し、これを随時更新することで演算を行うアーキ

キテクチャを設計した。このようなアーキテクチャをとることによって図27に示したweight-packed分の重みのみをPL側のRAMに保存すればよいこととなるため、消費するBlock RAMは表9のようになる。表9より、Block RAMのリソース消費量は347個にまで削減することが可能である。しかしながら、表5に示したようにZCU104に搭載されたBlock RAMは312個であるため、依然としてPL部に実装できないことが推測された。そこで私たちは次に説明するRAMの再利用を行うことによって更なるメモリ消費量の削減を行った。

表8 メモリの最大使用量

	入力要素 [Kb]	重み [Kb]	Block RAM
1層目	65	6,553	1,839
2層目	131	16,777	4,697
3層目	262	4,194	1238
4層目	524	16	169
総数	983	27,541	7,943

表9 各層の1出力要素当たりのメモリ使用量

	入力要素 [Kb]	重み [Kb]	Block RAM
1層目	65	12	24
2層目	131	65	61
3層目	262	32	92
4層目	524	16	169
総数	983	127	347

11.2 RAM の再利用によるリソースの最適化

限られたBlock RAMリソースでGANを実装するためには、個々のメモリのビット幅およびアドレス幅を、格納するデータ量と必要精度に即して最小限に定義する必要がある。本システムにおける各RAMの仕様について以下のように工夫を加えた。

1. **Feature_RAM / Accumulator_RAM のビット幅:** Feature_RAMのビット幅は、外部から転送されるランダム入力 z の精度

に適合する16bitを採用した。積和演算の結果を保持するAccumulator_RAMについては、累積加算過程におけるオーバーフローを防止するため、32bit幅を採用した。

2. **アドレス幅（深度）:** FPGAのBRAMは構成後に深度を動的に変更できないため、Feature_RAMおよびAccumulator_RAMのアドレス幅は、全4層の中で最大となる特徴マップの総要素数（画素数 × チャネル数）から算出した。本ネットワークで最大のデータ量を持つのは第3層の出力時であり、そのデータサイズは以下の通りとなる。

$$16(\text{px}) \times 16(\text{px}) \times 128(\text{ch}) = 32,768 \text{要素}$$

この 32,768 個の要素を一元的に管理するためには、 $2^{15} = 32,768$ であることから、アドレス幅を15bitと定義し、メモリ配置の最適化を図った。

3. **Weight_RAMのビット幅:** 重みパラメータは学習済みの8bit精度の値を基本単位としている。本設計ではデータ転送効率向上の観点から、 4×4 のカーネルのうち 2×2 のカーネル要素（4つ）を1つのアドレスに集約して保持する仕様にした。そこで、4つの重み要素（8bit × 4）を单一のアドレスに集約した32bit幅を採用した。
4. **Weight_RAMのアドレス幅:** 本システムでは全パラメータを一括で保持すると膨大なリソースを消費するため、出力を特定のチャネル数ごとに分割して演算を行う。本ネットワークにおいて入力チャネル数が最大となるのは第2層（512 ch）である。そこで、リソース制約と演算速度のバランスを考慮し、出力チャネルの分割単位を16とした。すると、重みメモリのアドレス幅は、入力チャネル数と出

力チャネルの分割単位を掛け合わせた以下の式で導出できる。

$$512(\text{入力 ch}) \times 16(\text{出力 ch}) = 8,192 \text{ カーネル}$$

この 8,192 個の要素を管理するためには $2^{13} = 8,192$ であることから、アドレス幅を 13bit と定義した。

以上の議論から、PLで用いられる Block RAM数は表10のように表される。ZCU104に搭載されたBlock RAM数は312であるので、本構成によりハードウェア実装可能である設計に落とし込んでいるといえる。

表 10 最終的な Block RAM 使用量の概算

	サイズ [Kb]	Block RAM
Feature_RAM	524	16
Weight_RAM	262	8
Accumulator_RAM	1,048	32
総数	1,874	56

12 シミュレーション

ハードウェアはVHDLを用いて設計を行い、そのシミュレーションにはXilinx社のVivadoを使用した。本章ではVivadoを用いて作成したシミュレーションをもとにPL部の動作解説を行う。

12.1 FIFO の分岐動作

入力要素 x を送信するFIFO-A、重み w を送信するFIFO-Cの切り替えについて着目する。今回路では、slv_reg0の3ビット目に示される FIFO選択信号により、AXI-LITEを通じたデータ書き込みがどのFIFOに転送されるかを分岐する仕組みとなっている。この仕組みについて、図36に示したシミュレーション波形をもとに説明する。図36はFIFO-Aに100次元入力を送信した後、FIFO-Cに1つ目の重み $4 \times 4 \times 100$ 要素を送信した波形を表したものである。PS側からFIFO-Aへのデータ転送

の様子を見ると、FIFO-A、FIFO-Cのどちらともデータ書き込み信号dinに値が入力されていることがわかる。しかしながら、書き込み可能信号wr_enはFIFO-Aのもののみ立っており、この時slv_reg0は01 (00001) から03 (00011) に変化している。同様に、FIFO-Cを利用する区間を確認するとPSからPLにデータ書き込みを行う際のwr_en信号はFIFO-Cのもののみ立っており、この時のslv_reg0は05 (00101) である。このように、PSからの制御信号によってどのFIFOにデータを書き込むかの分岐が行えていることがわかる。

12.2 Generator の動作

1 出力要素を得る手順

まず、1層目の中でも初めの出力要素を求める際の回路動作について説明を行う。図37に100次元入力 x の受信から1層目の1出力要素の取得までのシミュレーション波形を示す。図37をもとに計算の動作について確認する。まず、本波形で確認する計算過程は次の通りである。

1. PS側から100次元入力 x が送られるまで待機。取得後値をRAM-X (Feature_RAM) に格納
2. PS側から1つの出力要素を得るために必要な重み w が送られるまで待機
3. 取得後値をRAM-W (Weight_RAM) に格納し、計算開始
4. 計算終了時、計算終了信号を発信し、重み待機状態に遷移

まず手順1にて、演算に必要な100次元入力 x をGeneratorに取り込む。実際に、ram_x_din の動作からRAM-X (Feature_RAM) に入力データが書き込まれていることが確認できる。この時、動作モードはFIFO-Aを選択したうえで計算開始信号を立てることとなるた

め、slv_reg0は01から03に遷移する。

続いて手順2において、Generatorへの重み送信が開始するまでGeneratorは待機する。この時、PS側はPLに対して重み w を送信し、FIFO-Cはデータを蓄える。この際Generatorは計算を行わないため、計算回路の内部状態を表す信号stは、重み w を待つWAIT_W_REQとなっている。また、動作モードは計算開始信号が0かつFIFO-Cを選択するため05に遷移する。

手順3ではGeneratorが重み w を取得する。Generatorは計算に必要な入力 x と重み w がそろい次第計算を開始する。図より、演算結果を保存するRAM-Y (Accumulator_RAM) に出力1要素分である $4 \times 4 = 16$ 要素以上の書き込みがあるが、これは計算回路の仕様による挙動である。計算回路ではRAM-X (Feature_RAM) から入力 x のうち1変数、RAM-W (Weight_RAM) から得た重み w のうちの1変数を掛け合わせこれをRAM-Y (Accumulator_RAM) に書き込む動作を繰り返す。1出力要素(4×4)を得るためにには、100次元の入力 x と100次元の重み w の積を足し合わせる必要があるため、計算回路ではRAM-Y (Accumulator_RAM) への書き込みを行った後、次の計算結果とRAM-Y (Accumulator_RAM) の読み出しデータとの和を再び書き込む試行を繰り返す。このことから、シミュレーションのような波形となる。また、信号stは計算終了待機のWAIT_COREとなる。また、動作モードは計算開始信号が1となるため07に遷移する。

最後に、手順4では計算終了時に計算終了信号であるl1_doneを立てていることが確認できる。この信号はAXI-LITE modlueを通じてPS側に送信され、PS側は再び手順1-4を繰り返す。このことから、slv_reg0は再び05に戻る。このような操作をそれぞれの層の出力要素分行うことによって1層分の計算を完了す

ることができる。

1層分の計算を終了した際の手順

図38に1層目の計算から2層目の計算に遷移する際のシミュレーション波形を示す。1層目の計算では、 4×4 のサイズを持つ512要素の出力を得る。このことから、出力要素を示すcur_ocは0-511の値を遷移することとなる。実際にシミュレーションは波形を確認するとcur_ocが511となる区間があり、1層目の計算が終了していることが確認できる。ここで、1層分の計算を終了した際の手順を、1層目から2層目に遷移する際を例に以下で説明する。

1. 1層目の最後である512番目の出力要素を計算し、計算終了時にl1_doneを立てる
2. 内部状態がMOVE_DATAに遷移し、RAM-Y (Accumulator_RAM) のデータをReLU関数にとした後、RAM-X (Feature_RAM) に格納される
3. RAM-X (Feature_RAM) への格納が終了した際に、end_moveが立つ。RAM-X (Feature_RAM) を入力とした計算を行うために、次の層の計算で用いる重み w のロードを開始する

まず、手順1について「1出力を得る手順」でも説明したように1出力要素を得る計算が終了した際にl1_done信号が立つ。この時、cur_ocが計算を行う層を指定するstate_calによって定まる出力要素と一致しているとき、重み w の受付を終了しデータ移行状態のMOVE_DATAに遷移する。実際にシミュレーション波形より、手順1までの計算は重みを格納したRAM-W (Weight_RAM) の出力および入力要素のRAM-X (Feature_RAM) の出力を用いた演算結果を、RAM-Y (Accumulator_RAM) に書き込むことで完結している。

続いて、手順 2 では入力 x と重み w の積が保存された RAM-Y (Accumulator_RAM) の値に ReLU 関数を適用し、RAM-X (Feature_RAM) に保存する。実際に、シミュレーション波形では手順 2 において ram_x_din が受け付けられていることがわかる。

手順 3 では、RAM-X (Feature_RAM) へのデータ移行が終了した際に end_move 信号を PS 側に送信する。PS 側がこの信号を受け付けることにより PS は 1 層目の計算が終了したことを確認する。実際に、計算する層の層数を指定する state_cal は end_move 信号が立つとともに 1 に遷移していることがわかる。また、 slv_reg0 の値は 0d (01101) に遷移していることがわかるため、state_cal も変化していることが裏付けられる。このように、1 層分の計算が終了した際には出力結果を ReLU 関数を用いて整形して 2 層目の入力として用いるような回路動作が行えていることが確認できる。

計算終了時の手順

図39に計算終了時のGeneratorにおけるシミュレーション波形を示す。このシミュレーション波形は、3 層目の計算が終了した後 4 層目の計算を行い、計算終了信号を送るまでを表した図である。実際に、計算される層を表す state_cal 信号は 2 から 3 に遷移しており、4 層目の計算を行っていることが確認できる。計算終了までの手順を 3 層目から 4 層目の計算を行う部分を例に説明すると以下の通りとなる。

1. 1 層分の計算を終了した際の手順で示したように 3 層目の計算を終了し、RAM-X (Feature_RAM) に入力データを書き込む
2. 4 層目の計算に必要な重みを受け取ったのち、計算を開始する
3. 4 層目の計算終了した後、RAM-Y

(Accumulator_RAM) に格納された 32×32 要素に Tanh を適用し、1 つの信号として繋げたものを end_all と共に出力として送る

まず、手順 1 についてシミュレーション波形からも end_move が立っていることから正常動作していることが確認できる。続いて手順 2 について、今までと同様に重みの送信と計算開始信号 start_w を送ることで計算開始していることがわかる。また、slv_reg0 は 1d (11101) で重みのデータ送信、1f (11111) で計算開始となっていることも確認できる。最後に手順 3 にて、RAM-Y (Accumulator_RAM) に格納した値に Tanh とデータを連結する処理を加えた値を out_img として出力する。実際に、図39の拡大波形より、l4_done から 計算終了信号 end_all が立つまでの間に 出力信号 out_img の更新が行われており、この間に RAM-Y (Accumulator_RAM) の値に対する Tanh の適用とデータ連結を行っていることがわかる。このことから、Generator から出力される値は $32 \times 32 \times 8$ bit であり、この信号が 1clk で出力されることがわかる。

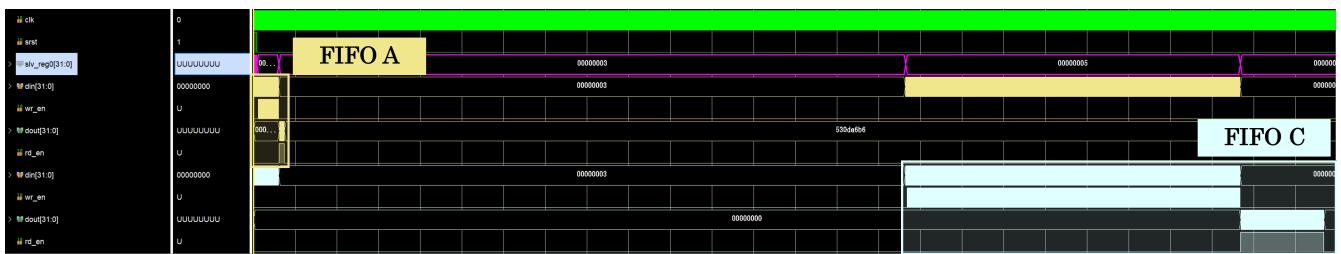


図 36 FIFO の分岐動作に関するシミュレーション波形

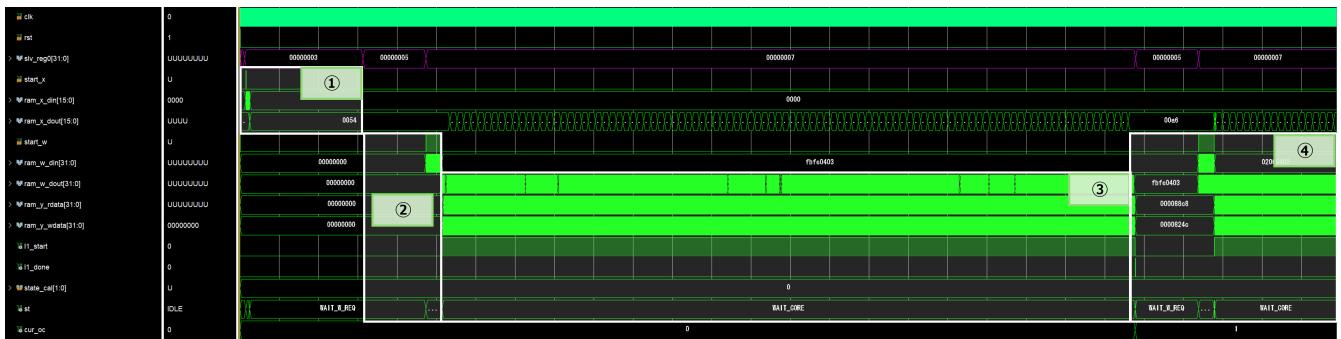


図 37 1出力要素分のシミュレーション波形



図 38 1層目から2層目の計算に変化する際のシミュレーション波形

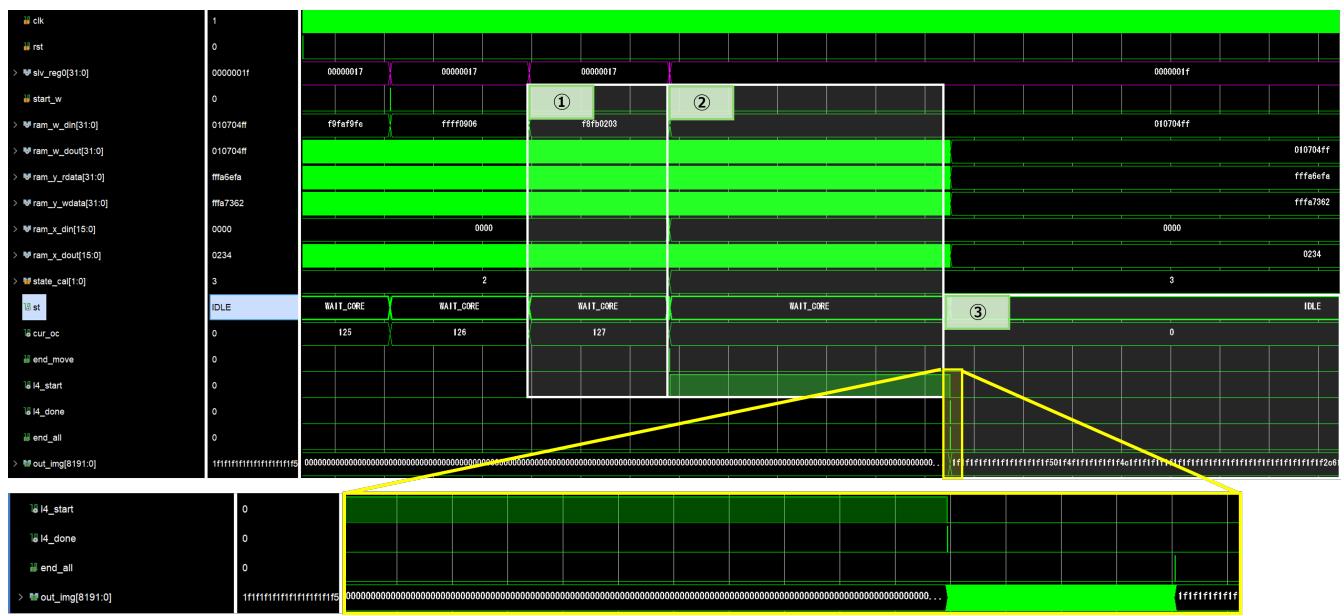


図 39 Generator 計算終了時のシミュレーション波形

13 実機動作

本章では、実際にシステムをFPGAボードで実機実装した際の結果をまとめます。

13.1 回路規模と動作周波数

本システムのうち、PL部が使用したリソースと最大周波数を表11に示します。なお、最大動作周波数 f_{max} の計算には、Vivadoで配置配線を行った際に得たWNS (Worst Negative

Slack) をもとに以下の式によって算出した。

$$f_{max} = \frac{1}{1/f - WNS} \quad (16)$$

現時点での実装を終えた回路では、1から4層目までのそれぞれの回路で1つの乗算器を利用している。このことから、使用リソースのうちDSP sliceは4つ利用していることがわかる。また、BRAMの使用量は「FPGA実装の中で工夫した点」で示したものとおおむね予測通りになっていることも確認できる。少し少なくなっている点に関しては、論理合成を行った際に1つのBRAMにtanhやFIFOといった容量の少ないRAMやROMを合わせて格納することでリソース消費量を削減していると考えられる。

表 11 使用リソースと最大動作周波数

	Used resources	Utilization [%]
LUT	13454	5.84
LUTRAM	86	0.08
FF	19464	4.22
BRAM	52	16.67
URAM	1	1.04
DSP	4	0.23
BUFG	3	0.55
Maximum Frequency[MHz]		118.04

13.2 CPUとの性能比較

今回実装したシステムと同様のモデルを表12に示す環境で動作させた際の性能比較を行う。

表 12 エミュレータPCの仕様

CPU	Intel Core i7-1165G7 2.80GHz
RAM	32 GB
OS	Windows 11 home
Language	Python 3.11.14

100次元入力を入力し、1つのグレースケール画像を得るまでにかかった計算時間はCPU

とFPGAとでそれぞれ表13のように計測された。表13より、FPGAの計算時間はCPUのものに比べて2.28倍の高速化を達成していることが確認できる。

表 13 CPU と FPGA の計算時間比較

CPU	FPGA
304.26[s]	133.32[s]

また、ランダム入力に対する画像出力についてエミュレータから得られたものとFPGAから得られたものを比較したものを図40に示す。図40より、エミュレータとFPGAの出力結果は概ね同様になっていることがわかる。このことから、エミュレータ環境と同等のシステムをFPGAボードに実装することができたといえる。

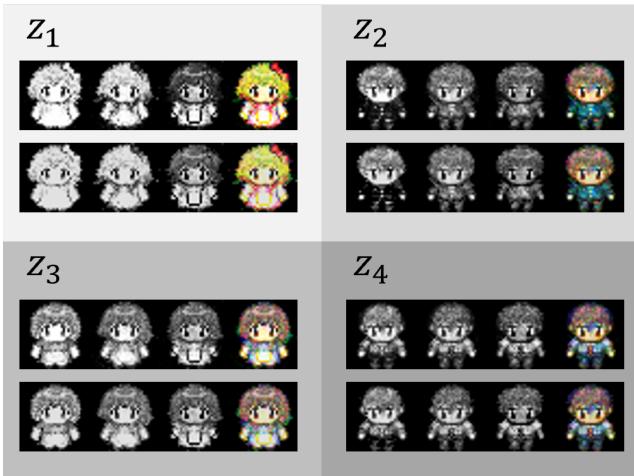


図 40 エミュレータと FPGA の出力比較
(上：エミュレータ、下：FPGA)

14 今後の展望

本研究で開発したシステムは、回路リソースの消費を抑えつつ、GANによる画像生成の基本動作をFPGA上で実現することに主眼を置いた。しかし、実用的なアバター生成を想定したリアルタイム処理を実現するためには、ソフトウェア、ハードウェア共にさらなる最適化が不可欠である。本章では、設計さ



図 41 演算時間のボトルネック特定

れたシステムについて現在実装中である要素について説明する。

14.1 学習法改善による高解像度化

今後の展望として、ソフトウェア上の学習手法を工夫することにより、より高画質な画像生成が実現できると考えられる。現状のFPGA実装では、リソース制約の観点から、出力可能な画像サイズには制限がある。しかしながら、本研究で用いた固定ベクトルによる新たな画像空間生成を、RGB成分ではなく分割画像に対して適用することで、この制約を回避できる可能性がある。具体的には、 64×64 画像を4分割して学習を行い、FPGA上では 32×32 画像を4枚出力する構成とすることで、実質的に高解像度画像の生成が可能になると考えられる。本手法に関する具体的な検証実験については、補足資料1に示す。

14.2 演算処理の並列化

図41は本システムのうち、3層目の計算の126番目の計算におけるシミュレーション波形を抜き出したものである。その中でも白枠で示した部分が重みのLOADにかかる区間、赤枠で示した部分が入力と重みを用いて1出力チャネルを得る計算部分となっている。図41からわかるように実際に計算を行う部分が実行時間の中でも多くの割合を占めていることから、現在のアーキテクチャは演算ボトルネックであるといえる。そこで、演算処理の並列化を行うことを目指す。

図42に示すようにメモリ帯域を拡張し、か

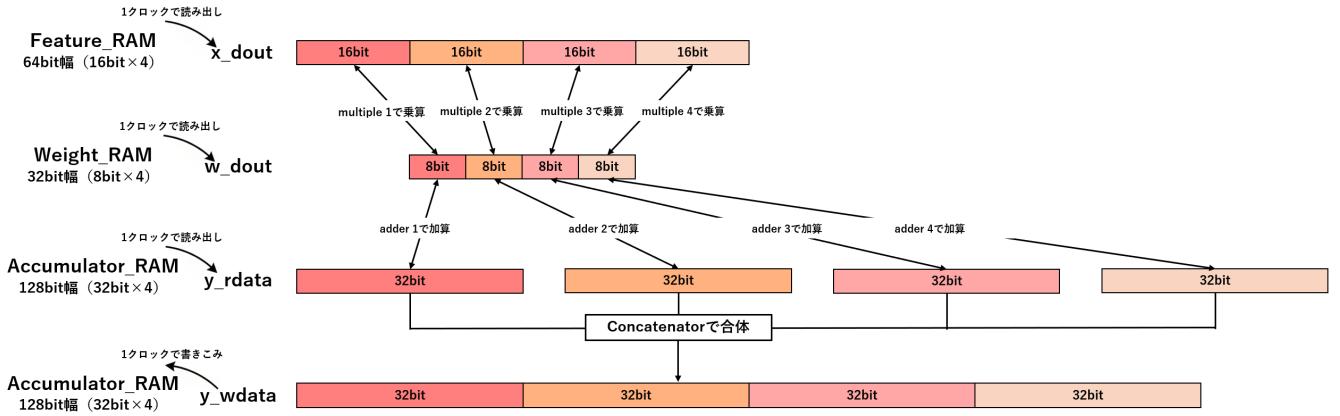


図 42 ビット帯域の分割による並列化の概念図

つ演算に用いていたkernel_miultiple1を各層で4つ準備することで並列演算を行うアーキテクチャを導入する。

現在の設計では、すでにWeight_RAMは 2×2 のカーネル要素(4要素)を保持するため、32bit幅(8bit×4要素)を採用している。しかし、FPGAに実装されたブロックRAM構造上の制約により、同一クロック内で複数アドレスへの同時アクセスは不可能である。このため、従来の手法では1つのカーネル演算に対して4クロックを要していた。

この課題を解決するため、Feature_RAMを64bit(16bit×4要素)、Accumulator_RAMを128bit(32bit×4要素)へとそれぞれ拡張し、演算コアであるkernel_multipleを4系統並列に配置する設計変更を行う。これにより、1クロックの読み出しで、4要素分の入力値および蓄積値を一括して演算コアへ供給できる。

また、演算後の4つの積和結果を单一データに統合するため、新たに結合モジュールConcatenatorを導入する。これにより、複数のBRAMを消費することなく、4つの独立した演算結果を128bitの单一データとして1クロックでメモリへ書き戻すことが可能となる。この帯域分割と物理的な並列化により、演算速度の約4倍へと向上させ、画像生成プロセス全体の劇的な高速化を目指す。

14.2.1 4並列化した演算コアの設計

図43に示す4並列化した演算コアでは、以下のステップで同期演算を実行する。

- 一括データ供給:** Feature_RAMから読み出された64bitの入力値と、Accumulator_RAMからの128bitの蓄積値を、4基の演算ユニット(multiple nおよびadder n)へ同時に供給する。
- 並列積和演算:** 各ユニットは、供給されたデータの中から自身が担当するビット範囲を直接抽出して演算を行う。
 - multiple n:** 64bitの入力値から特定の16bitを選択し、同時に供給される32bit重み値のうち対応する8bit要素を抽出する。抽出された8bit重み値は、乗算前に16bitへと精度拡張され、選択された16bit入力値との乗算が実行される。
 - adder n:** 128bitの蓄積値から対応する32bitを抽出し、同じkernelの乗算結果と合算する。
- 出力の同期結合:** 各adderから出力された32bitずつの演算結果は、Concatenatorによって128bitデータへ合体され、Accumulator_RAMの同一アドレスへ一

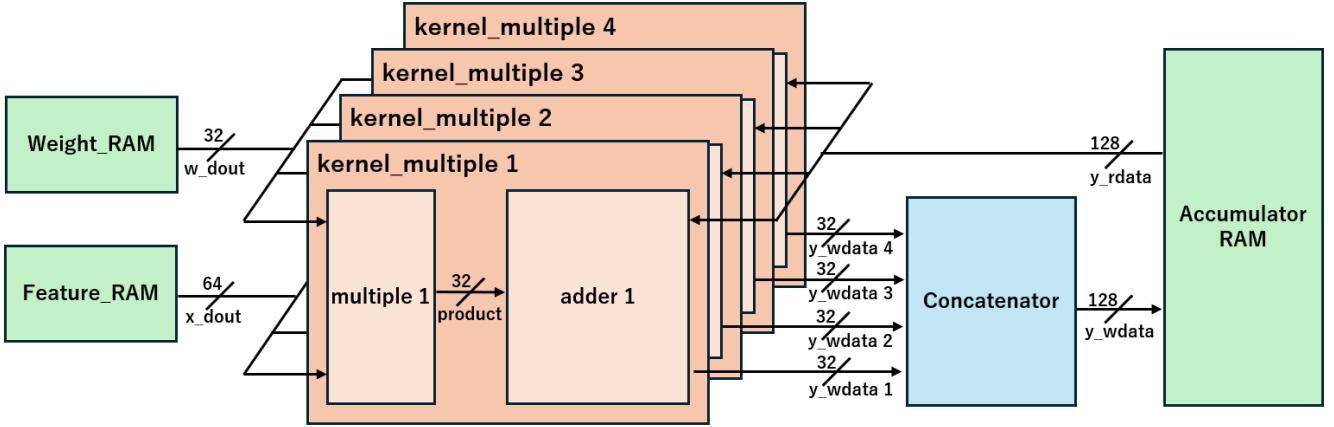


図 43 4 並列化した演算コア構成

括して保存される。

14.3 フルカラー画像生成への対応 (RGB 並列化)

本研究で開発したシステムは、1チャネル（モノクロ）の演算を順次繰り返す逐次実行方式を採用している。そのため、フルカラー画像を生成する際にはR・G・Bの各チャネルに対して個別に演算サイクルを回す必要があった。そこで、前述したカーネル演算の4並列化に加え、RGBの3色を同時に処理する並列を加えることで、合計12並列計算を計画している。

この設計の最大の特徴は、学習時に導入した固定ベクトル v_1, v_2 を活用することで、単一の重みパラメータをRGBの全チャネルで共有し、外部からの入力値（ランダムノイズ z ）や重みの転送コストを一切増加させることなくフルカラー化を実現できることである。

14.3.1 12 並列演算コア (kernel_multiple) の設計

図44に示すように、演算の核となるkernel_multipleユニットは、合計12個（4要素並列×3チャネル並列）の構成をとる。

单一のWeight_RAMから読み出された32bitの重み信号は、RGBすべての演算コアへ同時に供給される。RGB各チャネルに配置された

4つのkernel_multipleは、この共通の重みを用いながら、それぞれの色成分に対応する入力値に対して独立した積和演算を実行する。また、RGB各チャネルは同一座標を同時に処理するため、ステートマシンやアドレスは共通化できる。つまり、单一のControllerやloop_counter、addr_generatorで一括制御が可能である。この制御ロジックの共用化により、リソース消費を抑制しながら高密度な並列演算パスを構築している。

14.3.2 ハードウェア構成の拡張

12並列化に伴い、top_moduleの構成を図45のように拡張する。

- メモリ構成（計7基のRAM）**：重みパラメータを保持するWeight_RAMは1基のみとし、RGB全チャネルで共有する。一方で、各色独立した積和演算を行うため、入力値を保持するFeature_RAMおよび蓄積値を保持するAccumulator_RAMは、RGB各チャネル専用に3基ずつ配置する。
- Input_RGB_Gen の追加**：Input_Deserializerによって受け取られた100次元の入力ノイズ z に対し、固定ベクトル v_1, v_2 を加算して $z, z + v_1, z + v_2$ の3系統を生成するモジュールInput_RGB_Gen

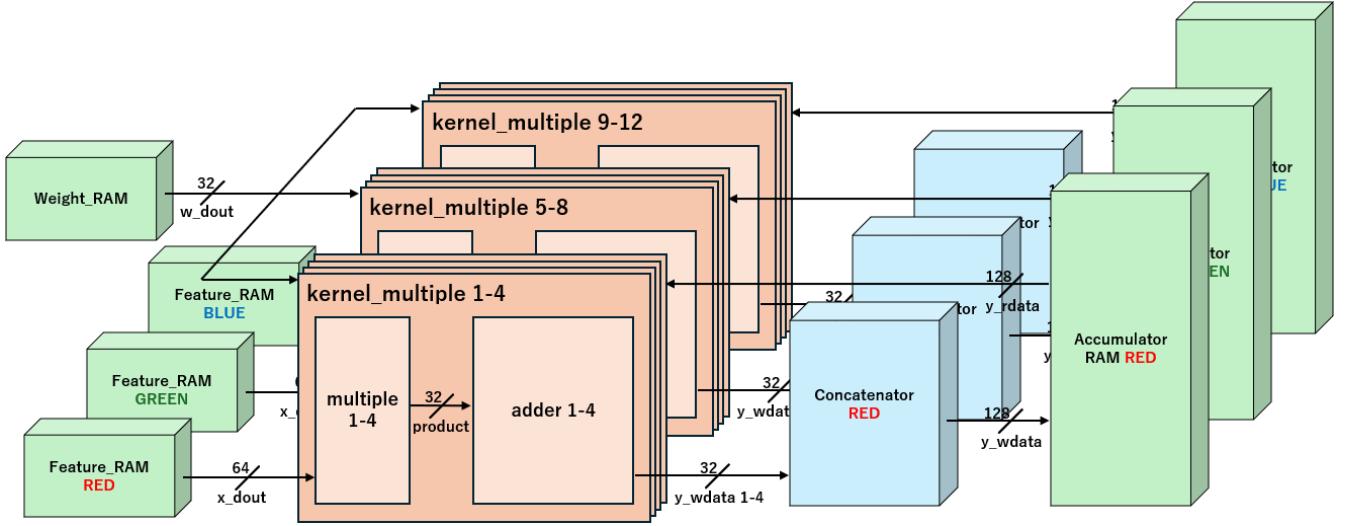


図 44 12 並列化した演算コア構成

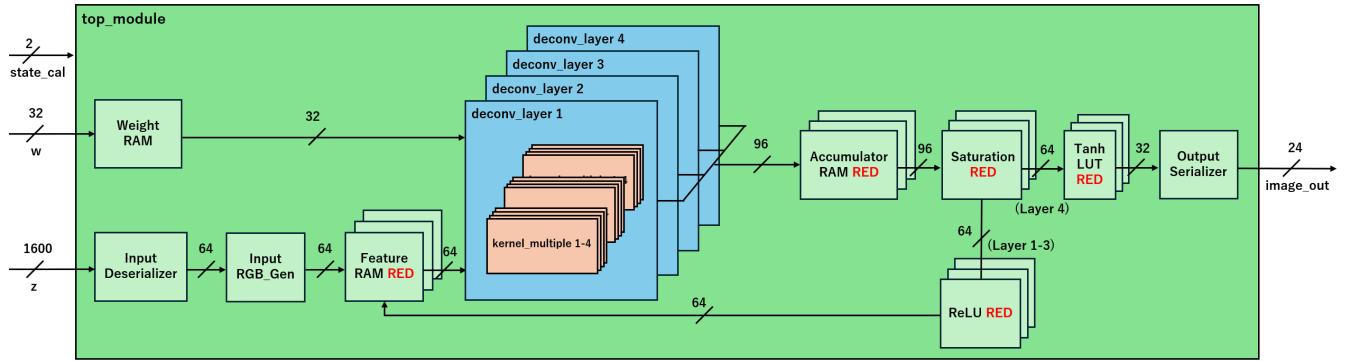


図 45 拡張された計算回路の構造

を追加する。本モジュールの導入により、外部からの転送部を変更することなく、内部でRGBの入力特徴マップを同時生成することが可能となった。

3. 演算部・活性化関数の三重化: 各チャネルの独立性を保つため、Saturation、ReLU、Tanh_LUTの各サブモジュールをそれぞれ3系統配置する。これにより、RGBの全チャネルが最後まで同時に処理される。
4. Output_Serializerの仕様変更: 最終層の演算完了後、各チャネルのTanh_LUTから出力された3系統の8bit画素データは、Output_Serializer内で24bitの单一パケットimage_outへと集約され、一括して外部

へ出力する。これにより、出力にかかる処理時間を大幅に短縮する。

本システムの実現可能性もハードウェアソースの観点から確認する。kernel_multiple1回路の並列化数が12倍となることから、利用するDSP sliceも同様に12倍となる。このことから、表11より、48個のDSP sliceを消費する。また、Feature_RAMおよびWeight_RAMはカラー並列化により3倍のメモリ容量を占有する。このことから、予想されるメモリ使用量は表8と同様に考えると表14のようになる。ZCU104に搭載されたblock RAM総数は364であることからUtilizationは約47%となり、さらにFIFO等の通信で利用されるメモリをさらに増やしたとしても十分実現可能で

あるといえる。

表 14 演算、カラー並列化時の Block RAM 使用量

	サイズ [Kb]	Block RAM
Feature_RAM	1,572	48
Weight_RAM	196	96
Accumulator_RAM	3,145	6
総数	4,913	148

15 まとめ

今回、アバター画像からGANを用いて新たなアバターを生成するシステムの実装を行った。学習モデル作成に当たってはFPGA実装を踏まえ、1つのネットワークでRGB空間を表現できる学習モデルを構築した。回路設計を行うに当たっては、大規模なCNNモデルをFPGAの限られたリソース量で再現するために、PS,PL協同動作するシステムを設計するとともに、今後の拡張性を高めた設計に力を入れた。これらの設計が功を奏した結果、FPGAボードにシステムを実装することができ、CPUと比べての高速化や同様の出力結果を得ることができた。また、11章に示したような今後の展望についても深く考察し、実現可能性を考えることができた。一方で、今後の展望で示した部分は未だに設計段階であり実装途中のものとなる。これからは特に今後の展望に示した実装に力を入れ、より実運用に近いシステムへと昇華させることに取り組みたい。

16 最後に

今年の設計課題はGenerative Adversarial Networks (GAN) であり、新規画像を作成できるという強みに着目しました。この着眼点において、社会的インパクトやオリジナリティ、実現可能性の観点を考慮した結果今回のドット絵のアバター作成をテーマとして選択いたしました。私たちのチームは全員ともハードウェア設計初心者であったことから、

テーマ策定の段階において実現可能性を考えることがとても難しかったです。このような状況の中でも、歴代の先輩方の実装された回路をもとにパラメータや回路リソースの目安を図りながらも、私たち独自のシステムを構築することを心掛けました。特に私たちの実装した「複数層で成り立つCNNモデル」は今まででも初の試みであり、どのようなアーキテクチャで実現するかはメンバー全員での度重なる議論と情報収集により成しえることができました。また、メンバーそれぞれが技術的強みを発揮し、ディープラーニングの観点からも、回路構成の観点からも面白いものを作成するよう努力いたしました。

この大会を通じて、深層学習やハードウェア設計に関する知見を深めることができたのはもちろんのこと、チームで1つのものを「モノ」を作り上げる難しさと楽しさを実感しました。メンバーそれぞれが「ユニークなシステムを作成する」という目標に向かい開発を行いましたが、やはり途中工程では認識のずれが生じるような場面もありました。このようなずれや様々な課題を乗り越え、本文で示しました実機実装まで行うことができた際は大きな達成感を得ることができました。このように、チームメンバーそれぞれの「ユニークさ」の主体性がアバター画像です。チームで取り組み1つの成果物を作り上げたこの経験を今後にも大いに生かしていきたいと思います。また、今後の展望にも示しましたように構想したすべての回路構成を実装することはまだ完遂できておりませんので、引き続き実装を続けつつ、チームで納得のいく成果物を作り上げたいと思います。このような貴重な機会を与えてくださったLSI Design Contest 2026実行委員会の皆様、関係者の方々にチーム一同より心から感謝申し上げます。

補足資料 1

固定ベクトルに対する追加考察・検証

1 固定ベクトル v_1, v_2 の頑健性について

本システムでは、固定ベクトル v_1, v_2 を用いて、 z の分布から少しずれた領域を生成することで、R、G、Bに対する空間を学習させた。しかし、それぞれの領域の重なりが大きい場合、学習時に領域が混ざってしまい学習が上手くいかない場合が想定される。(図46)

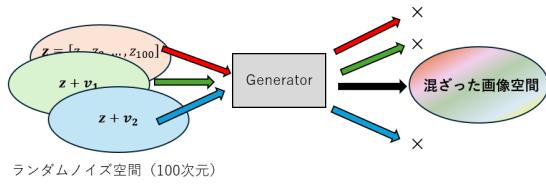


図 46 v_1, v_2 による画像空間が混ざる例

本章では、高次元空間においてランダムサンプリング点が球殻上に分布する性質に着目し、固定ベクトルを用いることで各分布の重なりが生じにくい状況であることを示す。

準備：分散の定義

確率変数 X 、平均 μ 、とすると分散 Var は「平均値からの偏差の2乗の平均」で定義できるから、式(17)となる。 $\mu = E[X]$ を用いると、最終的に式(18)が得られる。

$$Var(X) = E[(X - \mu)^2] \quad (17)$$

$$\begin{aligned} &= E[(X^2 - 2X\mu + \mu^2)] \\ &= E[X^2] - 2\mu E[X] + \mu^2 \\ &= E[X^2] - (E[X])^2 \end{aligned} \quad (18)$$

1.1 高次元空間におけるガウス分布

本節は、参考文献 [9] の第 3.1 章の内容を基にまとめた。独立で平均0、分散1をもつガウス分布からサンプリングした100次元のベクトルを式(19)に示す。

$$z = (z_1, z_2, \dots, z_{100}), z_i \sim \mathcal{N}(0, 1) \quad (19)$$

z のノルム二乗の期待値は、式(20)となる。

$$E\|z\|_2^2 = E\sum_{i=1}^n z_i^2 = \sum_{i=1}^n E[z_i^2] \quad (20)$$

式(20)に式(18)を適用すると、式(21)となる。

$$\begin{aligned} \sum_{i=1}^n E[z_i^2] &= \sum_{i=1}^n [Var(z_i) + (E[z_i^2])] \\ &= \sum_{i=1}^n (1 + 0) = n \end{aligned} \quad (21)$$

よって、 z のノルム二乗の期待値 $E\|z\|_2^2 = n$ となる。ここで、 $S_n = \|z\|_2^2$ とおくと、その分散は式(22)となる。

$$Var(S_n) = \sum_{i=1}^n Var(X_i^2) = \sum_{i=1}^n O(1) = O(n) \quad (22)$$

したがって、 $\sqrt{Var(S_n)} = O(\sqrt{n})$ が言えるから、式(23)となる。

$$S_n = n \pm O(\sqrt{n}) \quad (23)$$

$$\|z\|_2 = \sqrt{n \pm O(\sqrt{n})} \quad (24)$$

ここで、誤差項 h を含む形で $f(a+h)$ を二次まで泰ラー展開すると式(25)となる。

$$f(a+h) = f(a) + f'(a)h + O(h^2) \quad (25)$$

$h = \pm O(\sqrt{n})$ とおき、 $f(x) = \sqrt{x}$ に対して式(25)の泰ラー展開を用いると、式(26)となる。

$$f(x+h) = f(x) + \frac{1}{2\sqrt{x}}h + O(h^2) \quad (26)$$

ここで、 $x = n$ とすると、第二項は分子が $h = \pm O(\sqrt{n})$ で、分母が $2\sqrt{n}$ より、 $\pm O(1)$ である。また、第三項 $O(h^2) \rightarrow 0$ と無視できる。以上のことから、式(24)は式(27)となる。

$$\|z\|_2 = \sqrt{n+h} = \sqrt{n} \pm O(1) \quad (27)$$

したがって、高次元正規ベクトル z の長さ $\|z\|_2$ は \sqrt{n} 付近に集中することが分かる。

1.2 計算機実験

1.2.1 高次元空間における球殻集中現象の検証

本節では、100次元のランダムベクトルが半径 $\sqrt{100} = 10$ 付近の球殻上に集中する性質を確認する。

Python を用いて、20000個の100次元ベクトル $z \sim \mathcal{N}(0, I)$ を生成し、それらのノルムに関する統計量を表15に示す。ノルムの平均値は9.97となり、理論値 $\sqrt{100} = 10$ に近い値をとることが確認できた。また、 $8 < \|z\|_2 < 12$ を満たす割合は0.995であり、全体の99%以上のベクトルがこの範囲に含まれていることが分かる。

表15 潜在変数 $z \sim \mathcal{N}(0, I)$ のノルム分布

指標	値
ノルム平均 $\mathbb{E}[\ z\ _2]$	9.97
標準偏差 $\text{Std}[\ z\ _2]$	0.71
$8 < \ z\ _2 < 12$ に含まれる割合	0.995
$9 < \ z\ _2 < 11$ に含まれる割合	0.844

$z + v_1$ は3.5%、 $z + v_2$ は4.0%しか含まれないことが分かり、 z 空間と $z + v_1$ 、 $z + v_2$ 空間が十分に分離できていることが分かる。そのイメージ図を図47に示す。

表17 固定ベクトル v_1 を加えた潜在変数 $z + v_1$ のノルム分布

指標	値
ノルム平均 $\mathbb{E}[\ z + v_1\ _2]$	13.52
標準偏差 $\text{Std}[\ z + v_1\ _2]$	0.85
$8 < \ z + v_1\ _2 < 12$ に含まれる割合	0.035
$9 < \ z + v_1\ _2 < 11$ に含まれる割合	0.001

表18 固定ベクトル v_2 を加えた潜在変数 $z + v_2$ のノルム分布

指標	値
ノルム平均 $\mathbb{E}[\ z + v_2\ _2]$	13.47
標準偏差 $\text{Std}[\ z + v_2\ _2]$	0.85
$8 < \ z + v_2\ _2 < 12$ に含まれる割合	0.040
$9 < \ z + v_2\ _2 < 11$ に含まれる割合	0.002

z の空間と固定ベクトル空間 $z + v_1, z + v_2$ の分離可能性の検証

次に、潜在変数 z に加算する固定ベクトル v_1, v_2 を生成し、それらの統計量を表16に示す。 v_1, v_2 はサンプリング数が1であるため、ノルム平均は10から1程度離れていることが分かる。

表16 固定ベクトル v_1, v_2 の統計量

ベクトル	ノルム平均	標準偏差	$\ v\ _2$
v_1	0.048	0.918	9.15
v_2	-0.003	0.911	9.07

次に、 $z + v_1$ および $z + v_2$ としたときの統計量を表17、表18に示す。表17、表18から、 z の99%が含まれる $8 < r < 12$ の領域において、

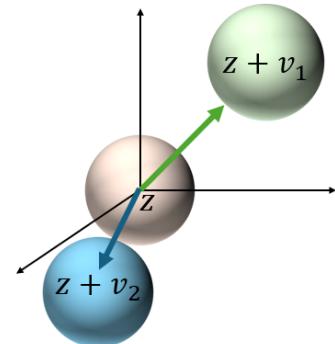


図47 $z, z+v_1, z+v_2$ の分離イメージ図

1.2.2 固定ベクトル空間 $z + v_1$ と $z + v_2$ の分離可能性の検証

次に、固定ベクトル空間 $z_1 = z + v_1$ と $z_2 = z + v_2$ が分離できているかの確認を起こさう。まず、 $z_1 = z + v_1$ の作る球殻は中心が

v_1 、半径が $r = \|z\|$ である。よって、 z_1, z_2 に対して v_1 だけ平行移動すると、 z'_1 は中心 0 に移動する（図48）。したがって、この状況では、 $z'_2 = z_2 - v_1$ が $r \pm 2$ および、 $r \pm 1$ の範囲にどのくらい含まれるかを評価すればよい。その結果を表19に示す。表19から、 z'_2 は $r \pm 2$ および、 $r \pm 1$ の範囲に一つも含まれないことが分かり、固定ベクトル空間 $z + v_1$ と $z + v_2$ は分離できていることが分かる。

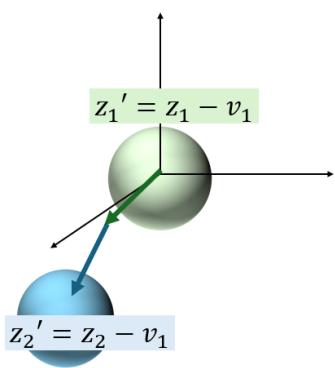


図 48 z_1, z_2 を $-v_1$ 平行移動したイメージ図

表 19 $z + v_1$ が作る球殻（中心 v_1 ）に対する $z + v_2$ の含有率

指標	値
中心 c	v_1
半径 $\ z\ $	9.97
$\ z + v_2 - c\ _2 \in [r \pm 2]$ の割合	0.000
$\ z + v_2 - c\ _2 \in [r \pm 1]$ の割合	0.000

2 固定ベクトルを用いた、高解像度出力の FPGA 応用

前章では、高次元のランダム空間 z に対して、固定ベクトル v_1, v_2 を用いて新しい空間 $z + v_1, z + v_2$ を定義できることを確かめた。そこで、この固定ベクトルを R,G,B 空間ではなく、分割画像に対して適応することで FPGAにおいても高解像度の画像出力が実現できることを示す。

2.1 現状の FPGA の課題とその解決策

現状の FPGA の課題は、NN または CNN 実装時にそのパラメータの多さからリソースに制限がかかり、 32×32 出力程度の画像しか出力できないことである。しかし、 64×64 の正解画像に対して、象限ごとの分割を行い、第1象限の画像空間を z 、第2象限の画像空間を $z + v_1$ 、第3象限の画像空間を $z + v_2$ 、第4象限を $z + v_3$ に対応付けて学習を行うことで、FPGA の出力としては 32×32 画像であるが、 $z \sim z + v_3$ までの出力画像を並べることで 64×64 の画像を生成することが可能となる。4分割して学習を行うことから、この GAN を Quattro GAN と名付け、そのシステムのイメージ図を図49に示す。

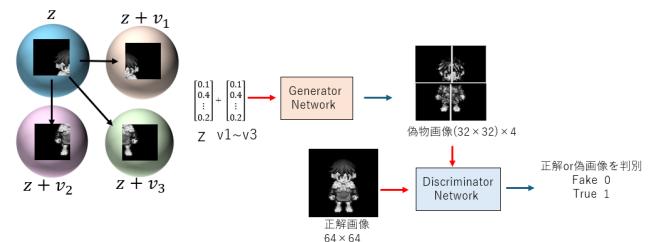


図 49 Quattro GAN のイメージ図

2.2 Quattro GAN の学習環境

Quattro GAN の実現可能性を検証するため、簡易的な実験を行った。Generator のニューラルネットワークには MLP を用い、その構成を表20に示す。正解画像としては、図50

に示す 64×64 ピクセルのアバター画像 911 枚を使用した。

表 20 MLP の層構成 ($100 \rightarrow 256 \rightarrow 32 \rightarrow 1024$)

層	入力 → 出力	活性化
FC1	$100 \rightarrow 256$	ReLU
FC2	$256 \rightarrow 32$	ReLU
FC3	$32 \rightarrow 1024$	Sigmoid

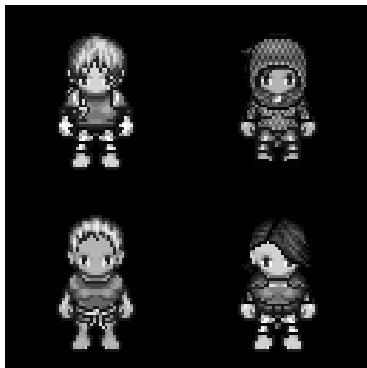


図 50 正解画像のデータセット (4 枚抜粋)

2.3 実験結果

学習結果を図51に示す。図51において、左がランダムな4入力($z^1 \sim z^4$)に対する、 $z, z + v_1, z + v_2, z + v_3$ の出力結果、右が任意のランダム入力に対する $z, z + v_1, z + v_2, z + v_3$ の出力を並べて表示したものである。この結果から、固定ベクトル $v_1 \sim v_3$ を用いることで、各象限ごとの画像を学習できていることが分かる。

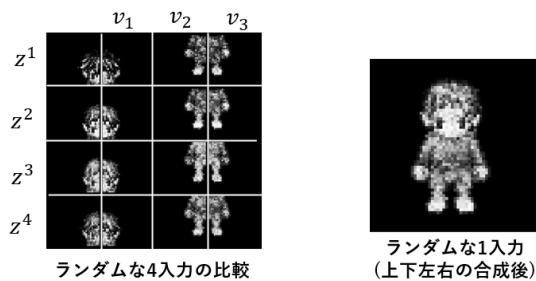


図 51 実験結果

さらに、GANとして連続空間に対する学習ができているかの評価を行う。ランダムな二

つのベクトル z_1, z_2 に対する出力結果を図52に示す。

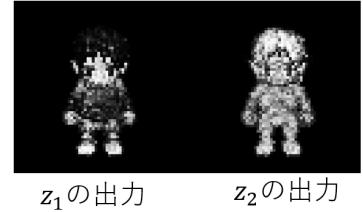


図 52 ランダムな二つのベクトル z_1, z_2 に対する出力結果

この二つのベクトルの間の3点を補完して生成したベクトルに対する出力結果を図53に示す。図53から、連続空間に対しても学習できており、GANとしての学習も機能していることが分かる。

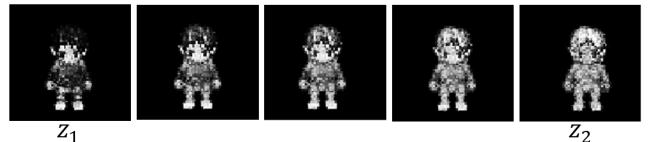


図 53 補間点に対する出力結果

2.4 固定ベクトルと FPGA 応用のまとめ

以上の結果より、固定ベクトルを用いることで、単一のネットワークにおいて複数の画像空間を学習可能であることが確認された。本研究では、R・G・B 空間に応する 3 種類の画像空間および 4 分割画像としての 4 種類の画像空間を、一つのネットワークで表現した。固定ベクトル数を増やすことで、より高解像度な画像出力や、多様な画像表現を可能とするネットワーク構造の実現も期待される。

单一のネットワークから複数の独立した意味を持つ情報を出力できる点は、FPGA 実装において大きな利点であり、回路規模や資源使用量の削減につながる。その結果として、より高機能な FPGA 搭載システムの構築に貢献できると考えられる。

補足資料 2

1 ニューラルネットワーク

ニューラルネットワークは複数の層から構成され、それぞれの層にはニューロンと呼ばれる計算ユニットが存在する。各ニューロンは他のニューロンと結合されており、その結合には重みが付けられている。この重みがニューラルネットワークの学習の鍵となり、入力データに対する適切な出力を生成するために最適化される。層の構造は入力層、中間層（隠れ層）、および出力層に分かれており、隠れ層の数が大きいほどより複雑なデータのパターンを学習する。ニューラルネットワークの構造を図54に示す。

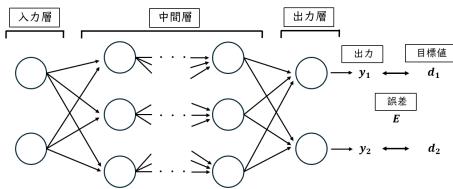


図 54 ニューラルネットワークの構造

(1) ニューロン

ニューロンの概略図を図55に示す。ここで、 z_j^m は第 m 層における j 番目のニューロンの出力値、 w_{ij}^m は第 $m-1$ 層 i 番目のニューロンから第 m 層 j 番目のニューロンへの重み、 u_j^m は第 m 層の j 番目のニューロンの入力値、 b_j^m はバイアス、 f は活性化関数である。

各ニューロンは、前層の出力値 z^{m-1} に重み w^m をかけた重み付き線形和を計算し、その総和にバイアス b_j^m を加えた値を入力値 u_j^m とし式(28)で定義する。ニューロンはこの入力値 u_j^m を活性化関数 f に通すことでの出力値 $z_j^m = f(u_j^m)$ を得る。

$$u_j^m = \sum_i w_{ij}^m z_i^{m-1} + b_j^m \quad (28)$$

ここで、活性化関数 $f(u_j^m)$ は、入力された値に非線形変換を行い、出力を生成する。代表的な活性化関数に、式(29)で表されるReLU関数がある。ReLU関数の微分は、入力値が正のとき1、負のとき0となるため、勾配消失問題を緩和する効果がある。

$$f(u_j^m) = \begin{cases} u_j^m & (u_j^m > 0) \\ 0 & (u_j^m \leq 0) \end{cases} \quad (29)$$

(2) 損失関数

損失関数は、ニューラルネットワークの出力値と目標値の差分を表す関数である。ニューラルネットワークは、目標値に対する損失関数 E の値を最小にするためにユニット間の重みとバイアスを調整する。損失関数の代表的なものに二乗誤差があり、ニューラルネットワークの出力 y_i と、目標値 d_i を用いて式(30)で定義する。

$$E = \frac{1}{2} \sum_i (y_i - d_i)^2 \quad (30)$$

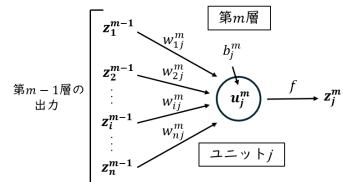


図 55 ニューロンの概略図

(3) 最急降下法

最急降下法は、損失関数を最小化するために、損失関数の勾配を用いて重み w を更新する手法である。最急降下法は、パラメータ w を式(31)で更新する。

$$w \leftarrow w - \eta \frac{\partial E}{\partial w} \quad (31)$$

(4) 誤差逆伝播法

誤差逆伝播法は、出力層から入力層に向かって、各層の重みを更新する手法である。誤差逆伝播法は、出力層の誤差を計算し、その誤差を入力層に向かって逆伝播させることで、各層の重みを更新する。誤差逆伝播法の手順を以下に示す。

出力層

L 層のニューラルネットワークにおける出力層の重みの更新量を考える。損失関数 E の w_{ij}^L に関する偏微分を連鎖律を用いて式(32)に示す。

$$\frac{\partial E}{\partial w_{ij}^L} = \frac{\partial E}{\partial u_j^L} \cdot \frac{\partial u_j^L}{\partial w_{ij}^L} = \delta_j^L z_i^{L-1} \quad (32)$$

ここで、 u_j^L は式(33)で表されるから、 u_j^L を w_{ij}^L で偏微分すると z_i^{L-1} となる。

$$\begin{aligned} u_j^L &= \sum_i w_{ij}^L z_i^{L-1} + b_j^L \\ &= w_{1j}^L z_1^{L-1} + w_{2j}^L z_2^{L-1} + \dots + w_{ij}^L z_i^{L-1} \\ &\quad + \dots + w_{nj}^L z_n^{L-1} + b_j^L \end{aligned} \quad (33)$$

次に、 δ_j^L を連鎖律を用いて式(34)に示す。

$$\delta_j^L = \frac{\partial E}{\partial u_j^L} = \frac{\partial E}{\partial z_j^L} \cdot \frac{\partial z_j^L}{\partial u_j^L} \quad (34)$$

ここで、損失関数 E が式(30)で表せることと、 z_j^L が出力値 y_j であることから、損失関数 E の z_j^L に関する偏微分は式(35)で表される。

$$\frac{\partial E}{\partial z_j^L} = \frac{\partial}{\partial y_j} \left(\frac{1}{2} \sum_i (y_i - d_i)^2 \right) = y_j - d_j \quad (35)$$

また、活性化関数 f をReLU関数とすると、 $z_j^L = f(u_j^L) = u_j^L$ であるから、 z_j^L に関する u_j^L の偏微分は1となる。したがって、式(32)は式(36)に変形できる。

$$\frac{\partial E}{\partial w_{ij}^L} = (y_j - d_j) z_i^{L-1} \quad (36)$$

式(36)において、 y_j は出力値、 d_j は目標値、 z_i^{L-1} は前層の出力値であるから、第 L 層において計算可能な値である。

中間層

第 l 層の重みの更新量を考える。損失関数 E の w_{ij}^l に関する偏微分を連鎖律を用いて式(37)に示す。

$$\frac{\partial E}{\partial w_{ij}^l} = \frac{\partial E}{\partial u_j^l} \cdot \frac{\partial u_j^l}{\partial w_{ij}^l} = \delta_j^l z_i^{l-1} \quad (37)$$

中間層における誤差分 δ_j^l を考えるためにあたって、第 l 層と第 $l+1$ 層におけるニューロン間の入出力関係を図56に示す。

損失関数 E の定義は式(30)であるから、第 l 層の重み更新には第 $l+1$ 層の出力値が必要となる。第 $l+1$ 層の入力値 u_i^{l+1} は第 l 層の出力値 u_j^l の関数でもあるから、損失関数 E は $E(u_1^{l+1}(u_j^l), u_2^{l+1}(u_j^l), \dots, u_k^{l+1}(u_j^l))$ の関数としてみることができる。したがって、損失関数 E の u_j^l に関する偏微分は多変数関数の連鎖律を用いて式(38)となる。

$$\begin{aligned} \delta_j^l &= \frac{\partial E}{\partial u_j^l} = \sum_k \frac{\partial E}{\partial u_k^{l+1}} \cdot \frac{\partial u_k^{l+1}}{\partial u_j^l} \\ &= \sum_k \delta_k^{l+1} \cdot \left[\frac{\partial u_k^{l+1}}{\partial z_j^l} \cdot \frac{\partial z_j^l}{\partial u_j^l} \right] = \sum_k \delta_k^{l+1} \cdot w_{kj}^{l+1} \end{aligned} \quad (38)$$

ここで、最後の式変形において $\frac{\partial u_k^{l+1}}{\partial z_j^l} = w_{kj}^{l+1}$ 、 $\frac{\partial z_j^l}{\partial u_j^l} = 1$ を用いた。 δ_k^{l+1} および w_{kj}^{l+1} の値は第 $l+1$ 層の値であるため、出力層側から更新を行うことで既知の値となる。つまり、第 l 層目の δ_j^l の値は第 $l+1$ 層目の δ_k^{l+1} ($k = 1, 2, \dots$) から求まる。このように、誤差逆伝播法では出力層から入力層に向かって誤差を逆伝播させることで、各層の重みの更新を行う。

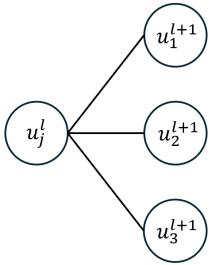


図 56 第 l 層と第 $l + 1$ 層におけるニューロン間の入出力関係

参考文献

- [1] 総務省 情報通信白書編集委員会. 令和 6 年版 情報通信白書. Technical report.
- [2] Nick Yee and Jeremy Bailenson. The Proteus Effect: The Effect of Transformed Self-Representation on Behavior. *Human Communication Research*, Vol. 33, No. 3, pp. 271–290, 2007.
- [3] Diederik P. Kingma and Max Welling. Auto-Encoding Variational Bayes, December 2022. arXiv:1312.6114 [stat].
- [4] Ian J. Goodfellow, Jean Pouget-Abadie, Mehdi Mirza, Bing Xu, David Warde-Farley, Sherjil Ozair, Aaron Courville, and Yoshua Bengio. Generative Adversarial Networks, June 2014. arXiv:1406.2661 [stat].
- [5] Alec Radford, Luke Metz, and Soumith Chintala. Unsupervised Representation Learning with Deep Convolutional Generative Adversarial Networks, January 2016. arXiv:1511.06434 [cs].
- [6] Bingqi Liu, Jiwei Lv, Xinyue Fan, Jie Luo, and Tianyi Zou. Application of an Improved DCGAN for Image Generation. *Mobile Information Systems*, Vol. 2022, No. 1, p. 9005552, 2022. eprint: <https://onlinelibrary.wiley.com/doi/pdf/10.1155/2022/9005552>.
- [7] Advanced Micro Devices, Inc. *ZCU104 Evaluation Board User Guide*. AMD.
- [8] Advanced Micro Devices, Inc. *What is PYNQ*. AMD.
- [9] Roman Vershynin. An Introduction with Applications in Data Science.