

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. ハードの工夫点 並列化など？？
4. ハードの工夫点

2章では、本システムを社会実装するにあたって、データセット拡張における有用性を示す。3章では、準備として生成モデルのなかでのGANの位置づけ、およびGANの学習アルゴリズムについて説明する。4章では、ソフトウェアにおける本システムの特徴を示す。5章では、ハード。

2 研究背景

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]。

本研究では、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、本研究では画像生成処理を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。

3 画像生成手法と GAN の位置づけ

アバター画像を生成するにあたり、どのような画像生成手法を用いるかは重要である。以下では、従来の画像生成手法の特徴を簡潔に整理し、本研究で用いる手法の位置づけを示す。

3.1 教師あり学習による画像生成

教師あり学習に基づく画像生成では、入力画像と対応する正解画像の組を用いて学習を



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)

行い、生成画像と正解画像との差を損失関数として最小化する。このような手法では、画素ごとの差分を評価する損失関数が用いられることが多い、代表的なものとして二乗誤差や絶対誤差が挙げられる。

これらの手法は、入力と出力の対応関係が明確なタスクにおいて有効であり、既存画像の再構成や変換といった用途で広く用いられてきた。

3.2 教師あり学習の限界

画像生成タスクにおいて、出力が多様な分布を持つ場合、画素単位の誤差を最小化する教師あり学習では、複数の正解を同時に満たそうとする結果、生成画像が平均的な外観となる傾向がある。この性質は、生成結果において輪郭のぼやけや細部表現の欠落として現れることがある。

また、教師あり学習では学習データに含まれない特徴を持つ画像を生成することが難しく、生成結果が既存データの補間にとどまる場合が多い。そのため、外観の多様性を持つ画像を生成する用途においては、別のアプローチが検討されている。

3.3 GAN の特徴と優位性

GANは、生成モデルと識別モデルを用いた敵対的学習によって画像生成を行う手法である。生成モデルはランダムな入力から画像を

生成し、識別モデルは生成画像と実画像とを判別する。両者を同時に学習させることで、生成モデルは実画像の分布に近い画像を生成するように更新される。

GANの特徴は、特定の正解画像との画素単位での一致を目的とせず、画像全体の分布を学習する点にある。このため、学習データの特徴を反映した新たな画像を生成でき、教師あり学習に基づく手法と比べて、より自然な外観を持つ画像の生成が可能である。

4 準備

本章では、3.1において生成モデルにおけるGANの位置づけを示し、3.2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、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。

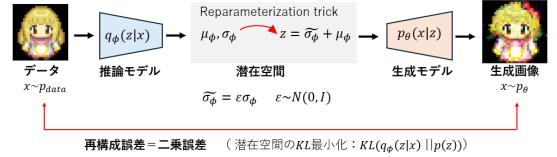


図2 VAE の概要図

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。

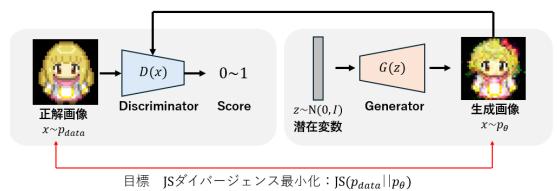


図3 GAN の概要図 (VAE 比較)

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ット $D(x; \theta_d)$ 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。

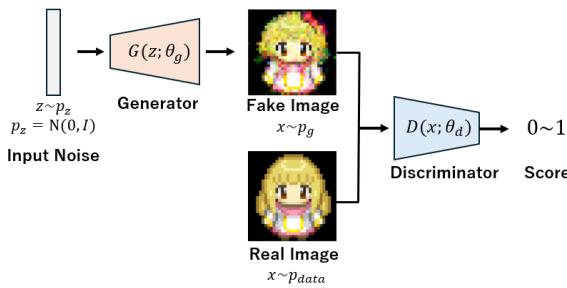


図4 GAN の概要図

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。

3.2.1、3.2.2にてDiscriminatorおよびGeneratorの最適解について、3.2.3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、対称性を持つさせるための指標として 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。

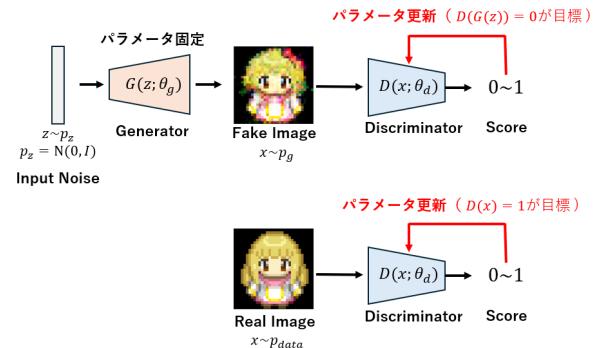


図 5 Discriminator の学習時の概要図

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。

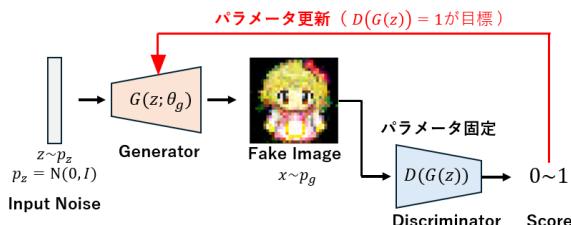


図 6 Generator の学習時の概要図

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、値が固定の 100 次元のランダムベクトル v_1, v_2 を用いることで、 z は R 画像、 $z + v_1$ は G 画像、 $z + v_2$ は B 画像の学習を行う。

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って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 画像、G 画像、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 実装時のイメージ

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。カラー画像を生成するには 3ch (RGB) の出力が必要であるが、メモリ制約のため FPGA 上で $32 \times 32 \times 3ch$ の画像を同時に出力することは困難である。

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 種の入力を生成する。具体的には、 z を R 画像の学習、 $z + v_1$ を G 画像の学習、 $z + v_2$ を B 画像の学習に対応付ける (図 7)。

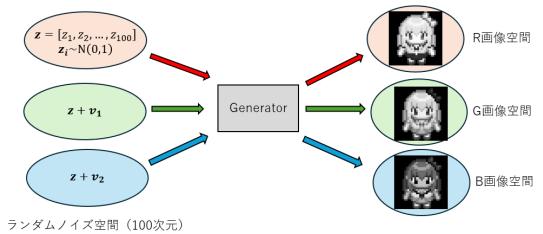


図 7 固定ベクトルを用いた RGB 学習

学習したパラメータおよび固定ベクトル v_1, v_2 を FPGA 実装することで、任意の 100 次元入力 z 入力すると、FPGA 内部で $z, z + v_1, z + v_2$ の 3 種のベクトルに対する演算を行うことができ、R · G · B に

対応した画像を生成できる(図8)。

R用・G用・B用に別々のネットワークを学習する場合、FPGAへパラメータを都度転送する必要がある。一方、本方式では入力を z 、 $z+v_1$ 、 $z+v_2$ のように工夫することで一つのネットワークでカラー画像を出力できることが強みである。

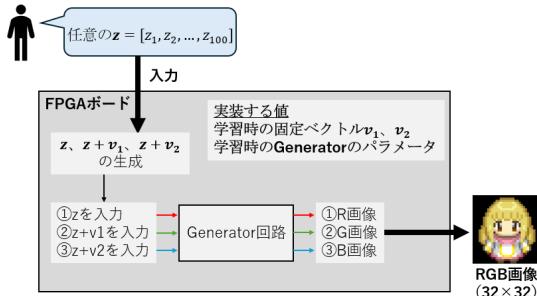


図8 FPGA実装時のカラー化演算イメージ

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、4.2.1~4.2.3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。

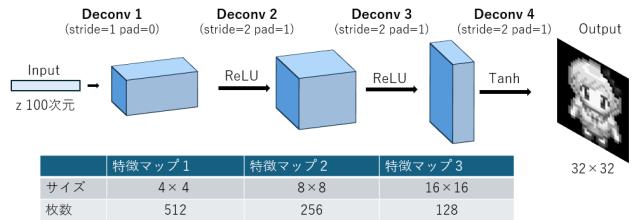


図10 Generatorのネットワーク構造

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。

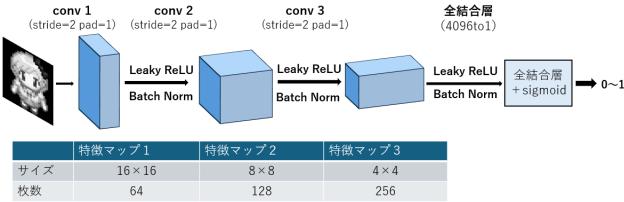


図 11 Discriminator のネットワーク構造

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。

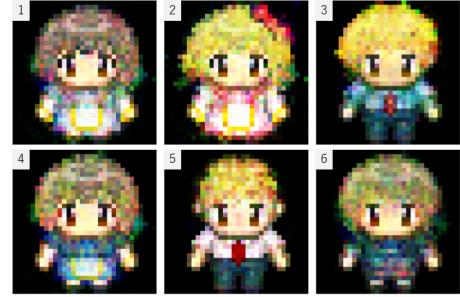


図 12 6種のランダム入力に対する出力結果

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 という連続空間に對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

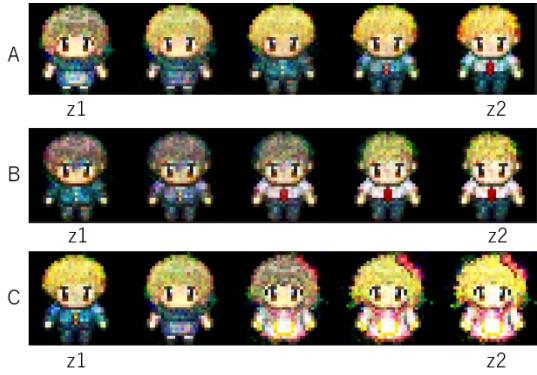


図 13 3 種のランダム入力セットに対する補間結果

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○章の補足資料に記述した。

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。

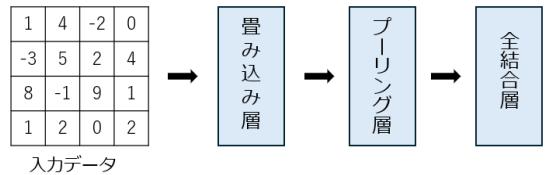


図 14 CNN の概要図

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 マスクスライドしながら、各位置において要素積の計算を行う。

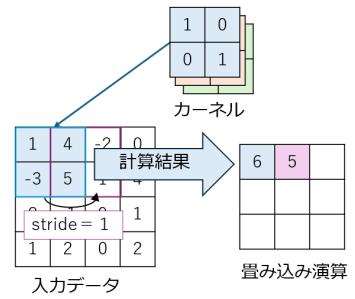


図 15 畠み込み演算

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採用する。(図 16)

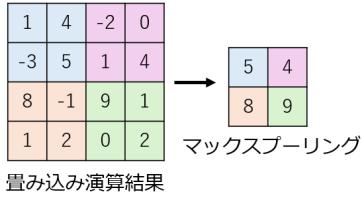


図 16 マックスプーリング

6.2 逆畳み込み演算

CNN では、情報を抽出するため畳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、逆畳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、逆畳み込みを行うことで 32×32 の画像を生成する。逆畳み込みは以下の 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. 畳み込み演算

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。

1	2	3
2	1	2
3	2	1

stride=1

1	0	2	0	3
0	0	0	0	0
2	0	1	0	2
0	0	0	0	0
3	0	2	0	1

stride=2

1	0	0	2	0	0	3
0	0	0	0	0	0	0
0	0	0	0	0	0	0

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。

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。

1	2	3
2	1	2
3	2	1

Kernel size
1 × 1

0	0	0	0	0
0	1	2	3	0
0	2	1	2	0
0	3	2	1	0
0	0	0	0	0

Kernel size
2 × 2

0	0	0	0	0	0
0	0	1	2	3	0
0	0	2	1	2	0
0	0	3	2	1	0
0	0	0	0	0	0
0	0	0	0	0	0

Kernel size
3 × 3

図 18 ゼロパディング

0	0	0	0	0	0	0
0	0	0	0	0	0	0
0	0	1	2	3	0	0
0	0	2	1	2	0	0
0	0	3	2	1	0	0
0	0	0	0	0	0	0
0	0	0	0	0	0	0

Padding = 0

0	0	0	0	0	0	0
0	1	2	3	0	0	0
0	2	1	2	0	0	0
0	3	2	1	0	0	0
0	0	0	0	0	0	0

Padding = 1

1	2	3
2	1	2
3	2	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	
Deconv2		4 × 4	stride=2 padding=1 kernel数 512×256
		8 × 8	
Deconv3		8 × 8	stride=2 padding=1 kernel数 256×128
		16 × 16	
Deconv4		16 × 16	stride=2 padding=1 kernel数 128×1
		32 × 32	

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「カーネルの高さ - 1」である。

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。

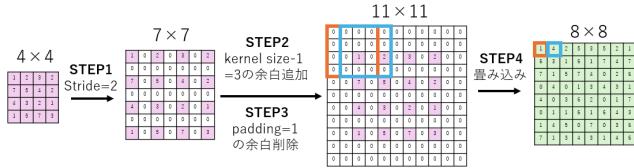


図 21 Deconv2 における $4 \times 4 \rightarrow 8 \times 8$ の計算例

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。

層	入力	カーネル	出力
Deconv1	input1 1 [□ □ ... □] \cdot^T	$\begin{bmatrix} 512 \\ \vdots \\ 512 \end{bmatrix}$	output1 (4×4) $= 1 \begin{bmatrix} \text{□} & \text{□} & \dots & \text{□} \end{bmatrix}$
Deconv2	input2 (=output1) 1 [□ □ ... □] \cdot^T	$\begin{bmatrix} 256 \\ \vdots \\ 256 \end{bmatrix}$	output2 (8×8) $= 1 \begin{bmatrix} \text{□} & \text{□} & \dots & \text{□} \end{bmatrix}$
Deconv3	input3 (=output2) 1 [□ □ ... □] \cdot^T	$\begin{bmatrix} 128 \\ \vdots \\ 128 \end{bmatrix}$	output3 (16×16) $= 1 \begin{bmatrix} \text{□} & \text{□} & \dots & \text{□} \end{bmatrix}$
Deconv4	input4 (=output3) 1 [□ □ ... □] \cdot^T	$\begin{bmatrix} 1 \\ \vdots \\ 128 \end{bmatrix}$	tanh =

図 22 各層ごとの入出力およびカーネル行列の関係

図 22 の計算関係から分かるように、各層では入力 $\text{input}[1][i]$ とカーネル $\text{kernel}[i][k]$ の行列積を計算することで、出力 $\text{output}[k]$ を得ている。 j 番目の出力を得るには、 j 列のカーネル $\text{kernel}[i][j]$ と入力 $\text{input}[1][i]$ 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 $\text{input}[1][i](i = 1, 2, \dots, 100)$ に対して、512 次元の $\text{output}[k](k = 1, 2, \dots, 512)$ 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。

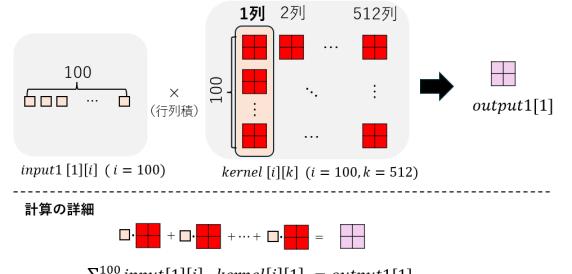


図 23 Deconv1 の 1 列目の計算例

図 23 のように、一列目の計算は式 (14) となる。

$$\sum_{i=1}^{100} \text{input1}[1][i] \cdot \text{kernel1}[i][1] = \text{output1}[1] \quad (14)$$

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。

$$\begin{aligned} & \sum_{k=1}^{512} \left[\sum_{i=1}^{100} \text{input1}[1][i] \cdot \text{kernel1}[i][k] \right] \\ &= \text{output1}[k] \end{aligned} \quad (15)$$

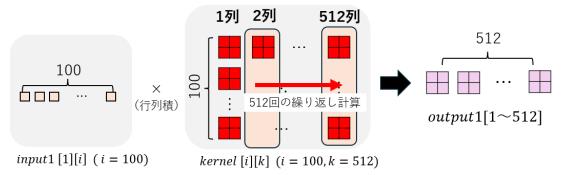


図 24 Deconv1 の全列の計算例

7 エミュレータにおける、固定小数点の検討

本章では、固定小数点の工夫により、エミュレータにおいて float の精度と同様の計算結果が得られることを示す。FPGA 実装にあたり、必要最低限のビット幅で高精度なエミュレータを設計することを目標とした。

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 $-128 \sim 127$ にマッピングされる。具体的に、実数値で 0.6046 は $77.38 \simeq 78$ (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、 $-64.000 \sim 63.998$ 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 $-128.000 \sim 127.996$ であり、分解能は 1/256 である。これを int16 の $-32768 \sim 32767$ 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 $77.38 \simeq 78$ 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

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

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、 $Q7.8 \times Q7.8 = Q14.16$ 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、「 $24025 \div 256 = 93.84 \simeq 94$ 」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、 $-128 \sim 127$ 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 を足したものを作成したものを tanh テーブルとして実装した。

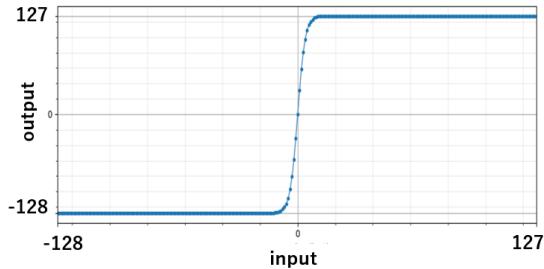


図 25 tanh テーブル (int8 対応)

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。

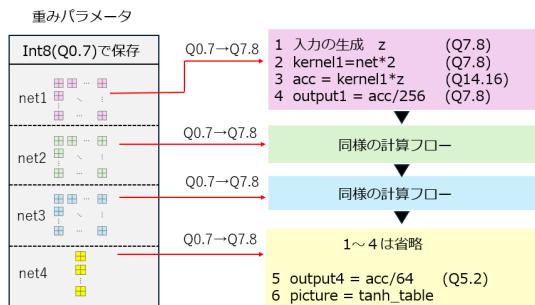


図 26 固定小数点の検討のまとめ

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。また、AMD の提供する Python productivity for Zynq(PYNQ) 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。本回路では 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

表 6 PS 部の仕様

APU	Dual-core Arm Cortex-A53
Architecture	Armv8-A
OS	PYNQ Linux
Framework	PYNQ
Language	Python

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

本システムでは、ram や配線数等のリソースの影響を考慮した結果、前述した学習モデルの中でも Generator のみを FPGA ボードに実装し、エミュレータで学習したパラメータを使用した。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。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。使用機器についてでも記述した通り、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 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 回路が 1 出力の演算終了信号を送信し、AXI-LITE 形式で PS 側がこれを受け取る
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要素である図 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、○○章のハードウェア上の工夫点にて詳しく解説する。

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 間の通信を可能である。特に AXI-LITE Module は図

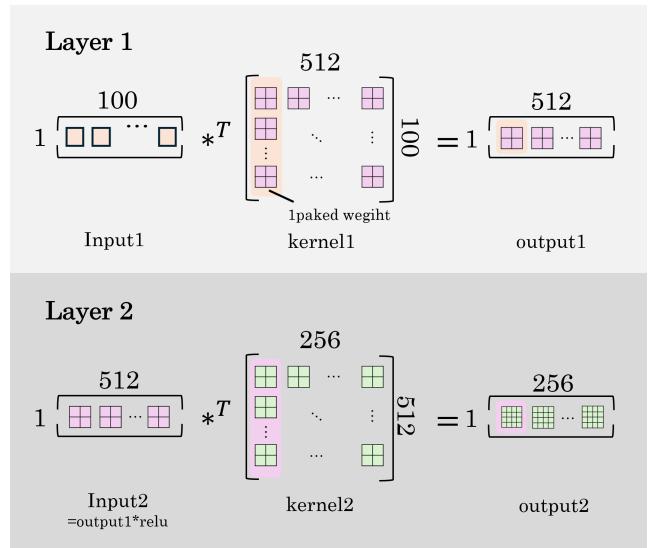


図 27 計算手順の概要

29 のようにあらわされる。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 00011）
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。

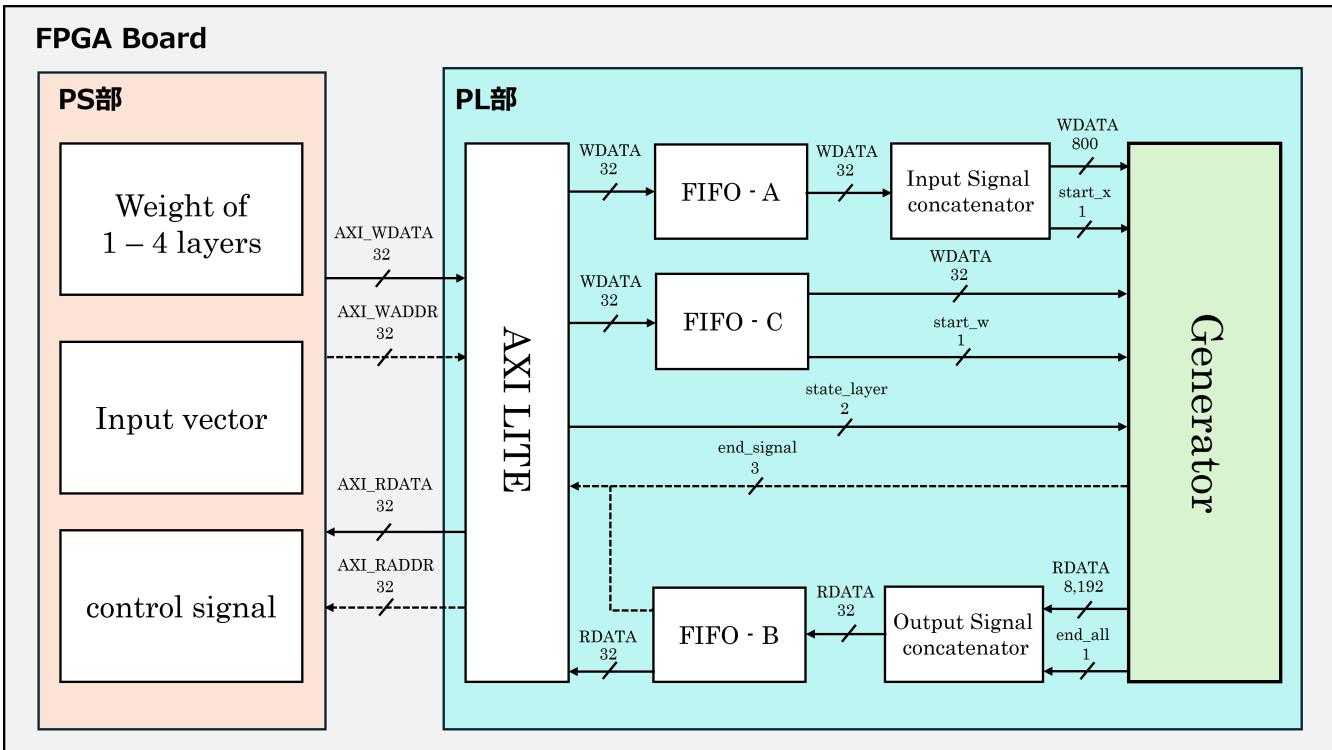


図 28 システムのアーキテクチャ

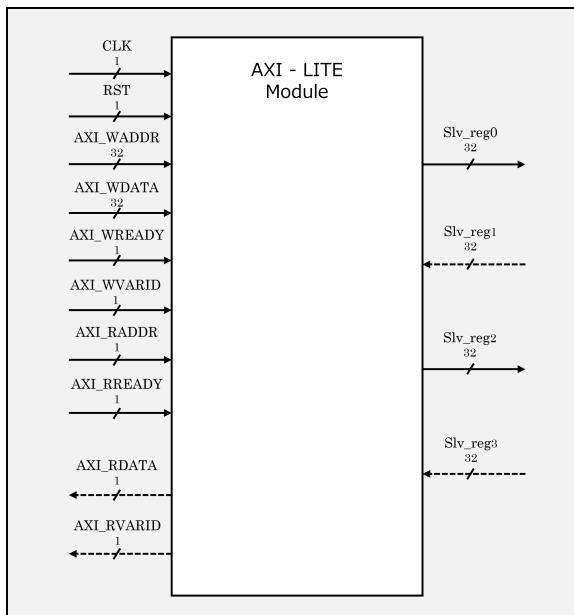


図 29 AXI-LITE module の構成

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		

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 $32 \times 32 \times 8\text{bit}$ 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。

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、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。次節にて詳しい説明を述べる。

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 cocatenator では、Generator に対して必要な入力データを 1CLK で渡

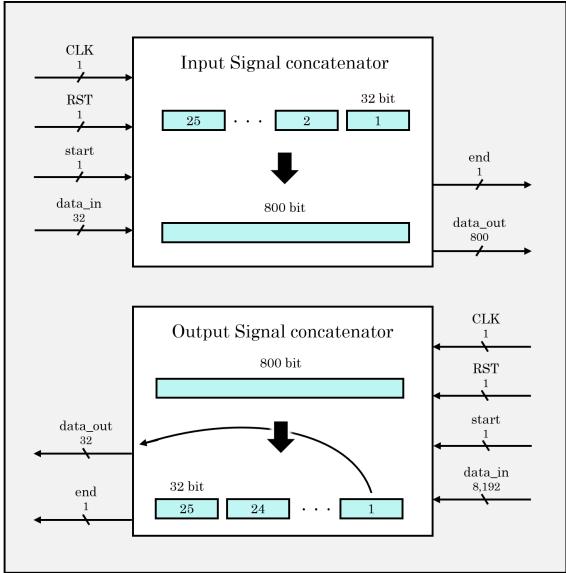


図 30 Input, output Signal concatenator の構成

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。

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 次元のノイズデータ (8bit × 100 要素) を受け取る入力インターフェースである。本モジュールは、800bit の並列データを 8bit 単位に分割し、それぞれ演算用の 16bit 幅へビット拡張して順次転送する。
- 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、 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。詳しくは、○○で説明する。
- Saturation:** Accumulator_RAM から読み出された 32bit の演算結果に対し、飽和演算を行なながら 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ノイズ入力 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。並行して、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。

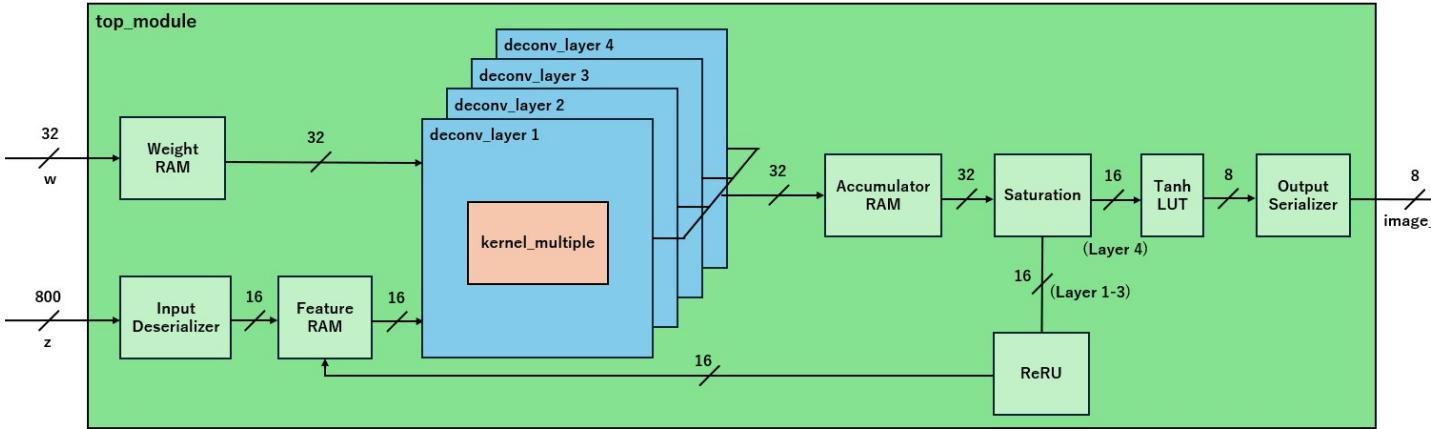


図 31 計算モジュールの階層構造と詳細データフロー図

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 ReRU 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。

- 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。
- 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、全データの転送が完了するまで待機する。

- 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。
- 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。
- 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。
- 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。
- 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。
- レイヤーループ判定 (Check_Loop 2)** : 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。
- 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 ReRU 活性化関数を

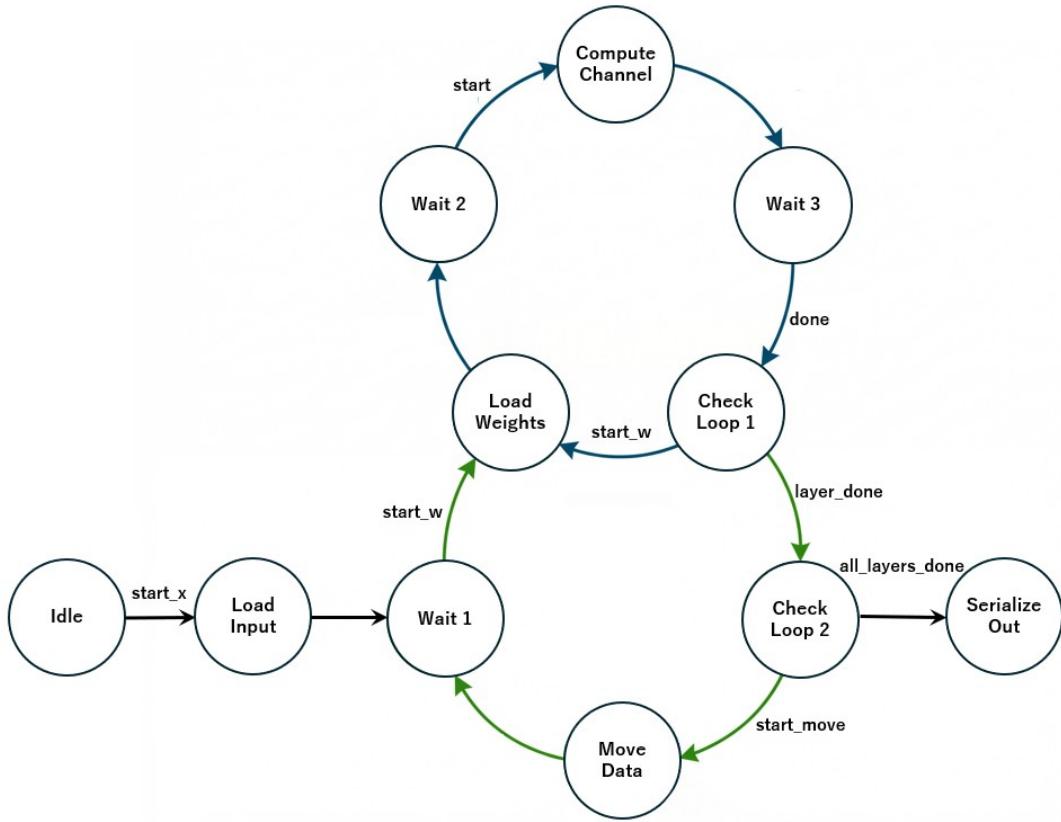


図 32 全体制御ステートマシンの簡略化状態遷移図

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。

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。

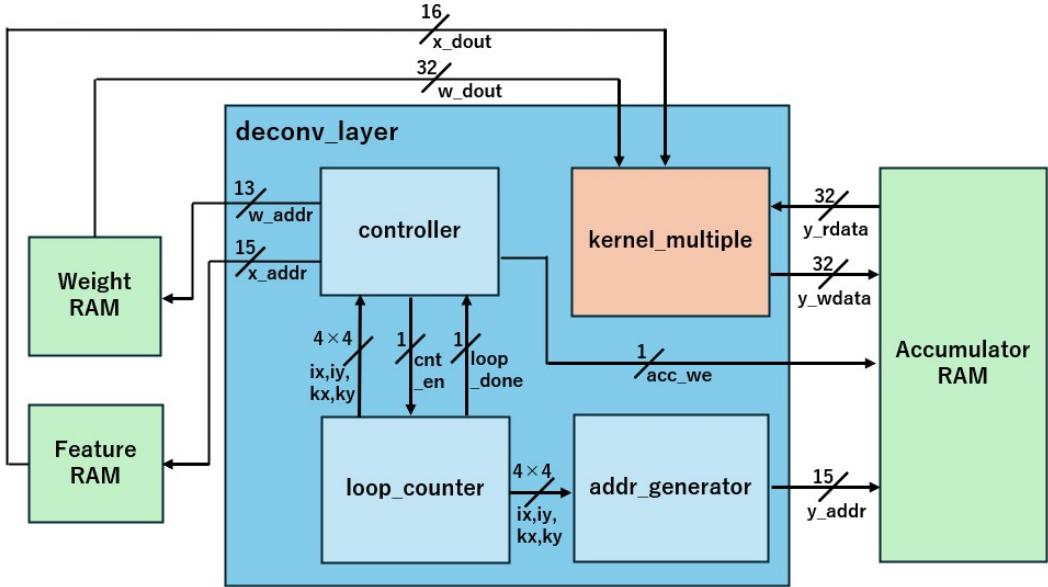


図 33 deconv_layer の内部構成

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。

まず、top_module から信号 start を受信すると、Controller が主導して Accumulator_RAM を初期

化し、演算の準備を整える。その後、Controller が loop_counter に対して cnt_en を発行することで、座標生成が開始される。

生成された座標信号は二手に分かれ、一方は Controller で読み出しアドレスに変換され、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。すると、Feature_RAM と Weight_RAM からは入力値と重み値、Accumulator_RAM から蓄積値が kernel_multiple に読み出される。

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。その加算結果 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。この間に loop_counter でカウントが進むと、Accumulator_RAM の読み出しと書き込みのアドレスが異なってしまう。そのため、蓄積値の読み出しと演算結果の書き戻しが完了するまで、Controller でカウントは停止される。

この一連のプロセスが、全座標に対して繰り返し実行されることで、1 チャネル分の畳み込み結果が Accumulator_RAM 上に形成されていく。最終的に全ての計算が完了したタイミングで、Controller は上位の top_module へ終了信号 done を出力する。

10.2.3 deconv_layer の制御ステートマシン

deconv_layer の動作は、Controller によって厳密に管理されている。そのステートマシンを図 34 に示す。

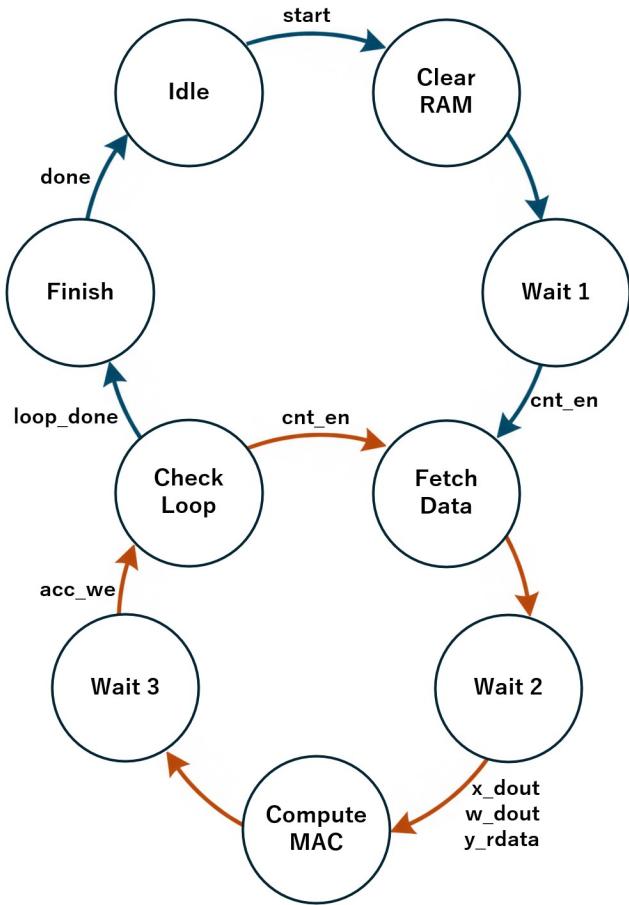


図 34 層内制御ステートマシンの状態遷移図

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) : 使用する Accumulator_RAM の領域を 0 でリセットする。
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。それに基づき各 RAM へ読み出しアドレスを提示する。
5. 読み出し待機 (Wait 2) : RAM からデータ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) : 確定した入力値・重み値・蓄積値を kernel_multiple へ供給し、1 座標分の積和演算を実行する。
7. 演算完了待機 (Wait 3) : kernel_multiple が MAC 処理を行い、Accumulator_RAM へ書きこむタイミングで、Controller が acc_we を High にする。
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。

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、1 カーネル分に相当する 32bit 幅 ($8\text{bit} \times 4$) 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、供給された座標信号は Fix_Pre によって出力

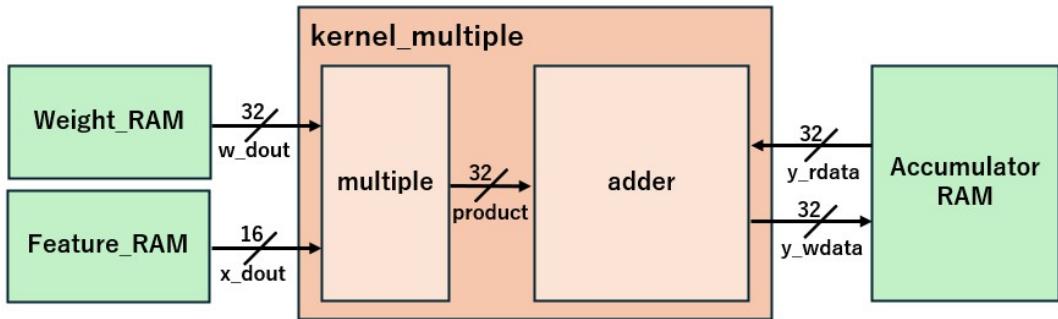


図 35 kernel_multiple の内部構成

座標へと変換され、この値が Accumulator RAM の読み出し・書き込みアドレスの特定に使用される。同時に、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、multiple へと送られる。

演算フェーズでは、

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、○○節で示した計算手順をとった理由について説明を行う。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テク

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

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。

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。本回路では、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 のどちらともの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 を利用して区間を確認すると 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 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 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 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 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 から入力 x のうち 1 変数、RAM-W 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 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 への書き込みを行った後、次の計算結果と RAM-Y 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狩猟信号である 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 のデータを relu 関数にとした後、RAM-X に格

納される

3. RAM-X への格納が終了した際に、end_move が立つ。RAM-X 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 の出力および入力要素の RAM-X の出力を用いた演算結果を、RAM-Y 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 の値に ReLu 関数を適用し、RAM-X 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 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 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 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 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 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。

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る。

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 10 に示す。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。

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 11 に示す環境で動作させた際の性能比較を行う。

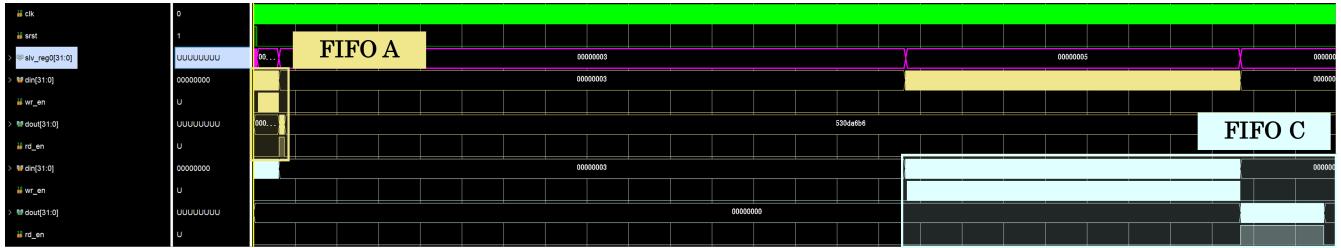


図 36 FIFO の分岐動作に関するシミュレーション波形

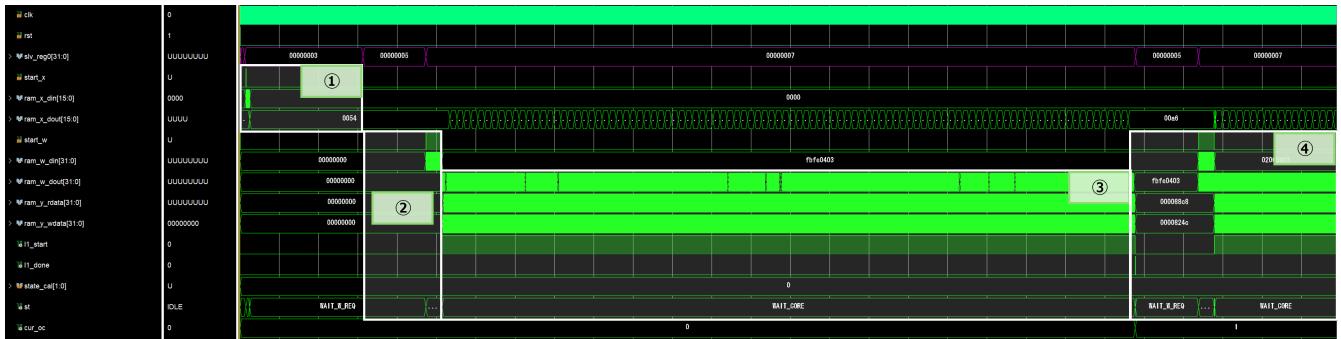


図 37 1出力要素分のシミュレーション波形



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

表 11 エミュレータ PC の仕様

表 10 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	

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 12 のように計測された。表 12 より、FPGA の計算時間は CPU のものに比べて 2.28 倍の



図 39 Generator 計算終了時のシミュレーション波形

表 12 CPU と FPGA の計算時間比較

CPU	FPGA
304.26[s]	133.32[s]

高速化を達成していることが確認できる。

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。

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 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

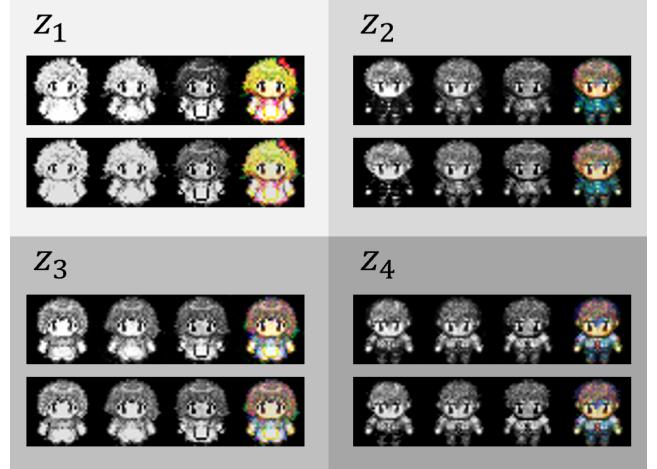


図 40 エミュレータと FPGA の出力比較
(上：エミュレータ、下：FPGA)

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



図 41 演算時間のボトルネック特定

(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。

14.1.1 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

- 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括して保存される。

14.2 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。

14.2.1 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。

単一の 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、単一の 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

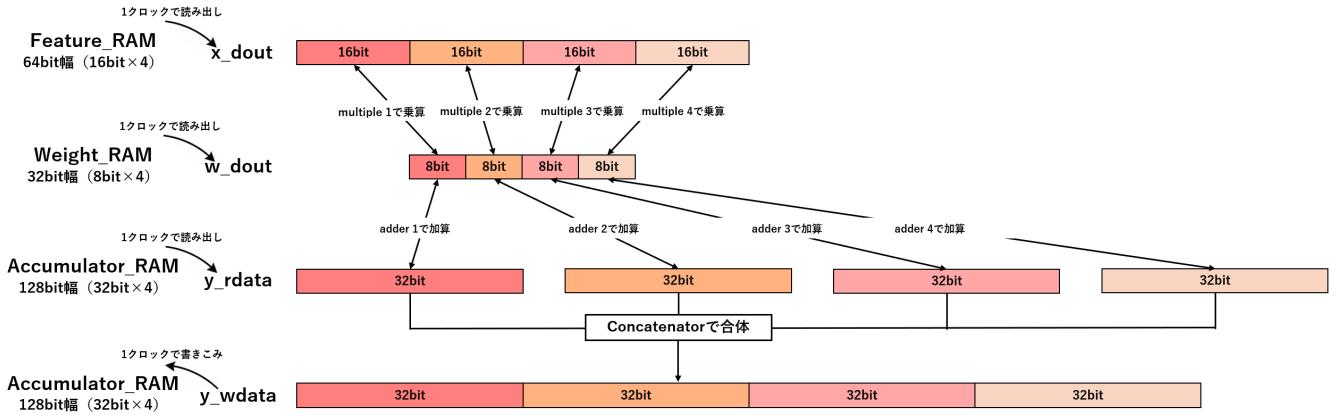


図 42 ビット帯域の分割による並列化の概念図

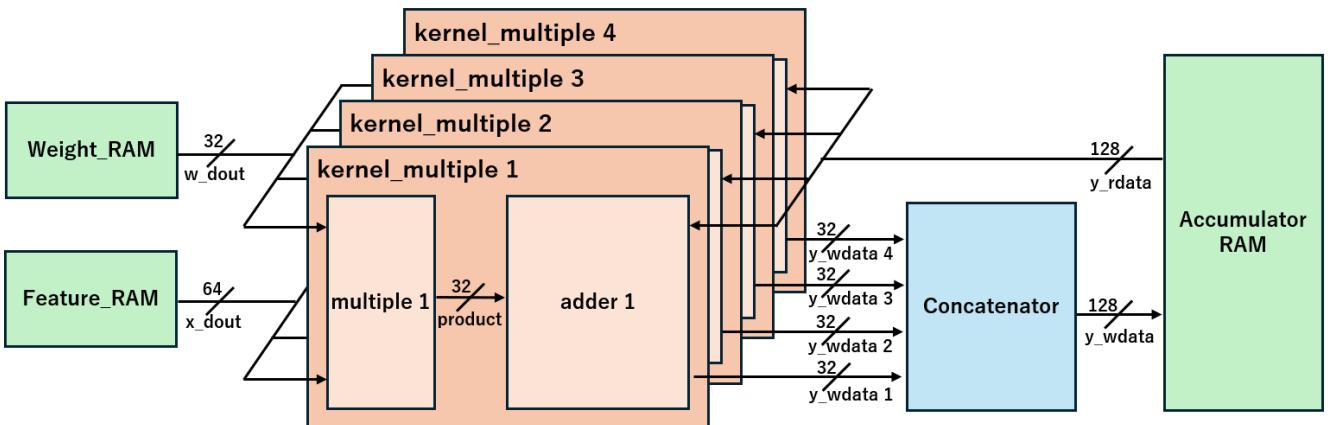


図 43 4 並列化した演算コア構成

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。

14.2.2 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 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。

- 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。
- 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 の単一パケット 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リ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

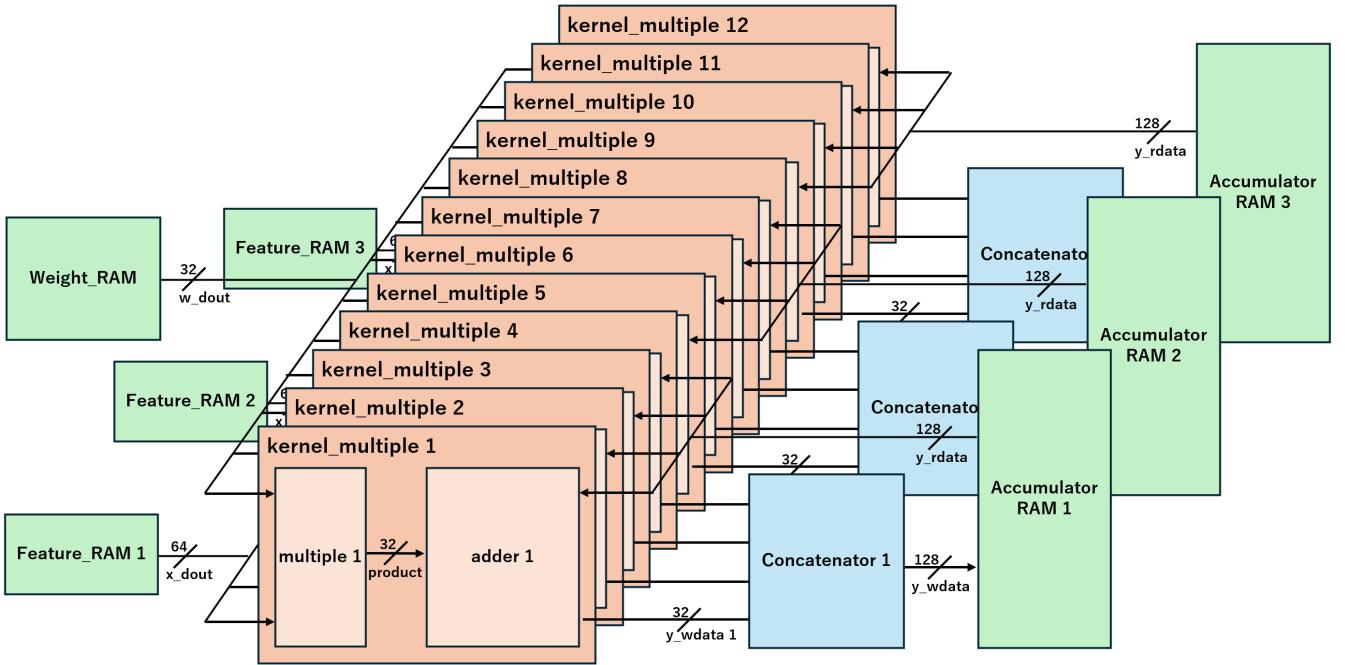


図 44 12 並列化した演算コア構成

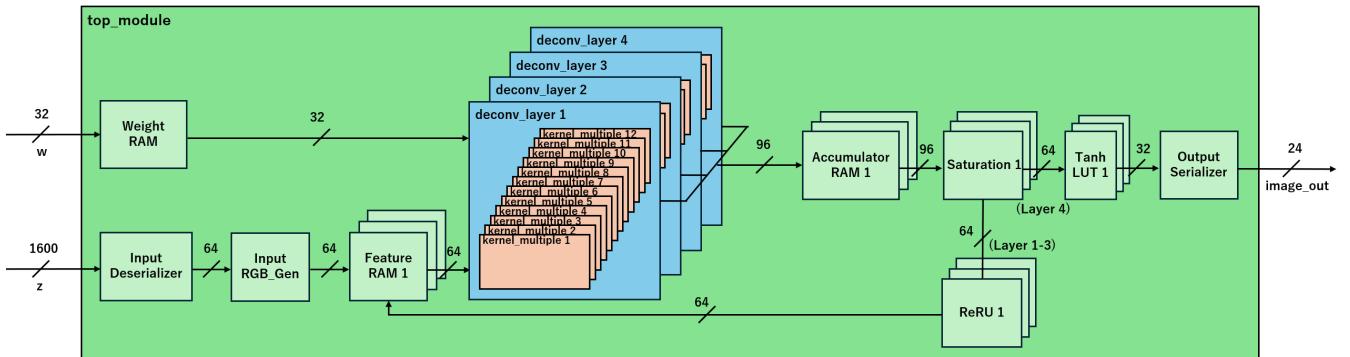


図 45 拡張された計算回路の構造

同様に 12 倍となる。このことから、表 10 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 13 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ても十分実現可能であるといえる。

表 13 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

14.3 システムの応用用途

15 まとめ

16 最後に

今年の設計課題は Generative Adversarial Networks (GAN) であり、昨年度と比べて○○でした。

そのテーマにおいて、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發揮し、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い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。

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

○章において、固定ベクトル v_1, v_2 を用いて、 z の分布から少しづ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)

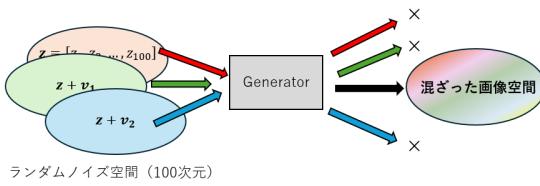


図 46 v_1, v_2 による画像空間が混ざる例

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。

$$\begin{aligned} Var(X) &= E[(X - \mu)^2] \\ &= E[(X^2 - 2X\mu + \mu^2)] \\ &= E[X^2] - 2\mu E[X] + \mu^2 \\ &= E[X^2] - (E[X])^2 \end{aligned} \quad (17) \quad (18)$$

1.1 高次元空間におけるガウス分布

本節は、参考文献 [7] 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。

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

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 $f(a + h)$ を二次までテイラー展開すると式 (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) のテイ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 14 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 14 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\]$	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

表 17 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	0.00150

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 15 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 度離れていることが分かる。

表 15 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 16、表 17 に示す。表 16、表 17 から、 z の 99% が含まれる $8 < r < 12$ の領域において、 $z + v_1$ 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 16 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	0.00105

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なう。まず、 $z_1 =$

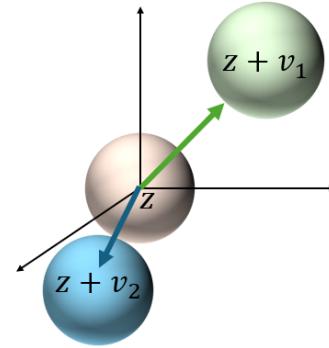


図 47 $z, z+v_1, z+v_2$ の分離イメージ図

$z + v_1$ 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 18 に示す。表 18 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。

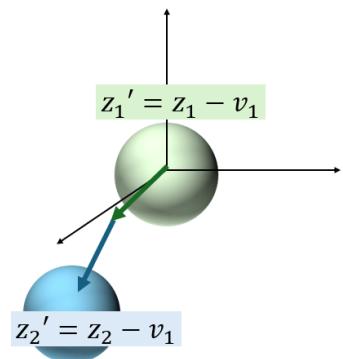


図 48 z_1, z_2 を $-v_1$ 平行移動したイメージ図

表 18 $z + v_1$ 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

ワークには MLP を用い、その構成を 表 19 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 19 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

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がある。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。

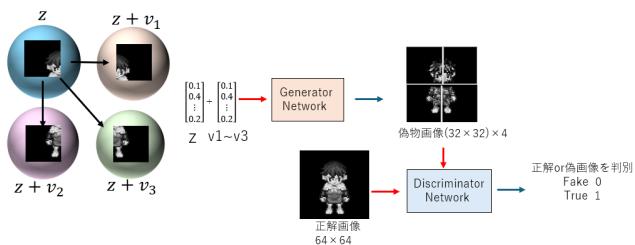


図 49 Quattro GAN のイメージ図

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

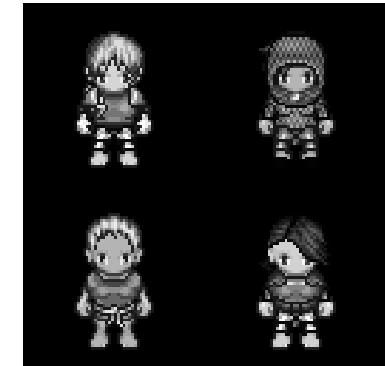


図 50 正解画像のデータセット (4 枚抜粋)

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。

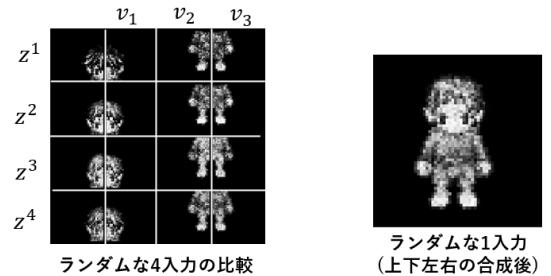


図 51 実験結果

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。

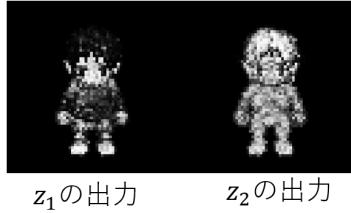


図 52 ランダムな二つのベクトル z_1, z_2 に対する出力結果

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。

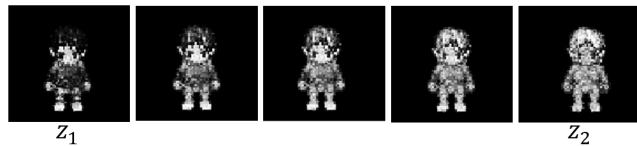


図 53 補間点に対する出力結果

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、单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 空間に応じて 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

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。

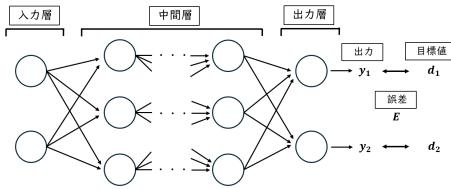


図 54 ニューラルネットワークの構造

(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 に通すことによって出力値 $z_j^m = f(u_j^m)$ 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。

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

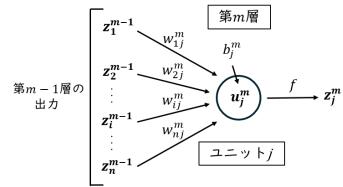


図 55 ニューロンの概略図

(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。

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

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。

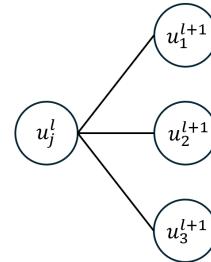


図 56 第 l 層と第 $l+1$ 層におけるニューロン間の入出力関係

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. *Leprint*: <https://onlinelibrary.wiley.com/doi/pdf/10.1155/2022/9005552>.
- [7] Roman Vershynin. An Introduction with Applications in Data Science.