The 28th LSI DESIGN CONTEST in OKINAWA 2025

（高信頼知的集積システム研究センターシンポジウム）

2025年コンテストテーマ:
Variational Autoencoder

開催日: 令和7年3月7日(金)
沖縄県石垣市、結い心センター

主催: LSIデザインコンテスト実行委員会, 高信頼知的集積システム研究センター

共催: 琉球大学工学部, 九州工業大学情報工学部

協賛: 電子デバイス産業新聞(旧半導体産業新聞), ギガファーム株式会社,
電子情報通信学会スマートインフォメディアシステム研究会,

後援: 九州職業能力開発大学校, 沖縄職業能力開発大学校, CQ出版社

http://www.lsi-contest.com

目次

コンテストの概要・目的

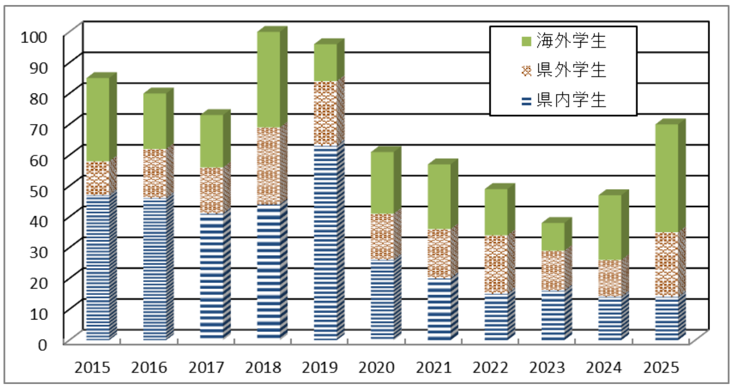
九州・沖縄さらに東南アジア地域の半導体産業および組み込み機器等のエレクトロニクス産業の振興を目的に、標題の学生向けの設計コンテストを毎年実施しております。主催は、琉球大学および九州工業大学の教員で構成するLSIデザインコンテスト実行委員会で実施しております。2025年は、国内外を含め70名（25チーム）を超えた応募がありました。

東シナ海および南シナ海沿岸のLSI産業拠点地域（九州、韓国、台湾、中国、シンガポール、フィリピン、マレーシア)は、世界の半導体市場の実に40％を越える高いシェアを有しており（九州以東の日本本土は含みません！)、その中心に位置する沖縄はそのビジネスチャンスが大変高いものと期待されます。そうした地理的好条件の沖縄にて、学生向けLSI設計コンテストを実施し、学生すなわち未来のエンジニアの設計スキルを上げることにより、将来的に沖縄や東南アジアでの企業誘致やベンチャー起業につなげたいと考えております。

今回は、AI の一つである「Autoencoder」がテーマになります。処理の高速化，回路規模の削減を目指したハードウェア設計など、いろいろなアイデアが生まれることを期待しております。

本コンテストの主旨をご理解頂き、多くの学生の皆様（高専、大学、大学院生）の参加を期待しております。

学生参加数の推移



国内

琉球大学、千葉大学、九州工業大学、会津大学、大分県立工科短期大学、沖縄職業能力開発大学校、大阪工業大学、大阪大学、京都大学、近畿大学、高知工科大学、神戸大学、芝浦工業大学、職業能力開発大学校、東海大学、東京工業大学、東京都立科学技術大学、東京理科大学、東北大学、徳島大学、豊橋技術科学大学、長崎大学、新潟大学、広島大学、法政大学、山形大学、横浜国立大学、立命館大学、早稲田大学、沖縄高専、有明高専、木更津高専、久留米高専、豊田高専、北陸先端科学技術大学院大学

海外

Bandung Institute of Technology 、Institut Teknologi Telkom 、Telecommunication College Bandung 、Telkom University （インドネシア）、Univ. of Science、Ho Chi Minh city、Hanoi Univ. of Science （ベトナム）、Chosun 大学（韓国）、Univ. of Valladolod（スペイン）、Univ. of Kaiserslautern（ドイツ）、NTI（エジプト）、McMaster 大学（カナダ）、南西大学（中国）

これまでの設計課題

■ 2001 年：　「デジタル CDMA レシーバ」
■ 2002 年：　「差集合巡回符号エラー訂正回路」
■ 2003 年：　「静的ハフマン符号用の可変長デコーダ」
■ 2004 年：　「共通鍵暗号 AES 用 SubByte 変換回路」
■ 2005 年：　「デジタル FM レシーバ」
■ 2006 年：　「2 次元積符号繰り返しデコーダ」
■ 2007 年：　「64 点高速フーリエ変換」
■ 2008 年：　「RSA 暗号デコーダ」
■ 2009 年：　「Small RISC Processor」
■ 2010 年：　「エラー訂正：BCH 符号」
■ 2011 年：　「DCT」
■ 2012 年：　「16/64/128-point Flexible FFT」
■ 2013 年：　「SW・HW 協調設計を用いたノイズ除去システム」
■ 2014 年：　「SW・HW 協調設計を用いたノイズ除去システム」
■ 2015 年：　「正弦波発生回路」
■ 2016 年：　「人物検出用パターンマッチング回路」
■ 2017 年：　「人物検出動画像処理システム」
■ 2018 年：　「ニューラルネットワーク
　　（バックプロパゲーション）」
■ 2019 年：　「深層学習（バックプロパゲーション）」
■ 2020 年：　「畳み込みニューラルネットワーク（CNN）」
■ 2021 年：　「強化学習」
■ 2022 年：　「Deep Q-Network」
■ 2023 年：　「局所的最大値／最小値」
■ 2024 年：　「Autoencoder」

■ **2025 年：　「Variational Autoencoder」**

開 催 概 要

名　　称	:	「第28回 LSI デザインコンテスト in 沖縄 2025」
実行委員長	:	九州工業大学大学院　情報工学研究院　情報・通信工学研究系　名誉教授　尾知博
主　　催	:	LSI デザインコンテスト実行委員会、高信頼知的集積システム研究センター http://www.lsi-contest.com/
共　　催	:	琉球大学工学部、九州工業大学情報工学部
協　　賛	:	電子デバイス産業新聞（旧半導体産業新聞）、 ギガファーム株式会社、電子情報通信学会スマートインフォメディアシステム研究会、
後　　援	:	CQ 出版社、 九州職業能力開発大学校、沖縄職業能力開発大学校
実行事務局	:	九州工業大学情報工学部情報・通信工学科黒崎研究室 LSI デザインコンテスト実行委員会事務局
日　　時	:	2025 年 3 月 7 日（金）　13:00～18:00
会　　場	:	結い心センター 〒907-0022　沖縄県石垣市字大川 100 番地 3
目　　的	:	実践的な課題を用いた学生対象のデジタル集積回路設計のコンテストであり、半導体集積回路設計能力の向上とともに国際的に交流することで学生の工学に関する視野を広めることを目的としている。
対　　象	:	国内大学・大学院生、高専学生、アジア地域大学生
来場予定者	:	100 名（入場無料）
募集方法	:	各大学・高専へのパンフレット送付、LSI 関連雑誌等へのリリース
ホームページ	:	http://www.LSI-contest.com/

【審査員】
実行委員長：　九州工業大学　尾知 博
審査委員長：　琉球大学　和田 知久
東京大学　藤田 昌宏
大阪大学　尾上 孝雄
千歳科学技術大学　宮永 喜一
鳥取大学　伊藤 良生
九州工業大学　黒崎 正行
琉球大学　吉田 たけお
ディポネゴロ大学　Wahyul Syafei Amien
バンドン工科大学　Trio Adiono
九州職業能力開発大学校
沖縄職業能力開発大学校
他協賛企業・法人審査員　　　　　　　　（敬称省略・順不同）

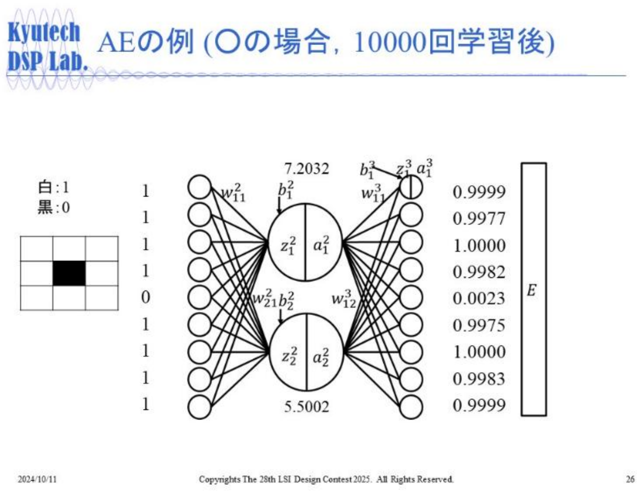
連絡先
住　所：〒820-8502 福岡県飯塚市川津 680-4　九州工業大学大学院情報工学研究院情報・通信工学研究系
電　話：0948-29-7667　　Email: support@LSI-contest.com
LSI デザインコンテスト実行委員会委員長
担当者　尾知　博

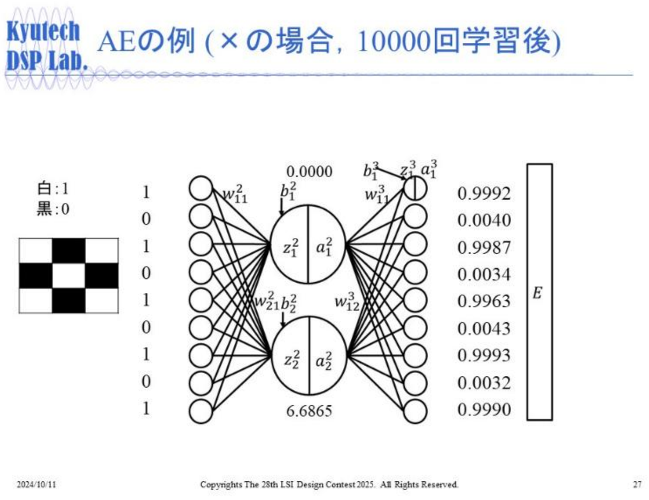
【設計テーマ】　Variational Autoencoder

　第28回のテーマは、AIの中でも異常堅守や雑音除去などに応用されている変分オートエンコーダーです。今回は「Variational Autoencoder」をテーマとして、処理の高速化、回路規模の削減を目指したハードウェア設計を行うことを目的とします。Level1の課題では、3×3の〇×を判定する回路を設計します。Level2の課題では、画素数を増やした任意の画像に対するVariational Autoencoderの回路を設計します。Level3の課題では問題や構造はすべて自由とし、よりユニークなAutoencoderを使用したシステムを設計します。
　要求されている設計は、HDL（VHDLもしくはVerilogHDL）による設計と論理合成および検証結果です。

実装アルゴリズムと環境

■　Variational Autoencoder の概要
　Variational Autoencoder は次元を圧縮するようなエンコーダーと復元を行うデコーダーとで構成されています。
　例として，3×3 の画像における Variational　Autoencoder の構成図を示します。





■　実装環境
Synopsys® Synplify Pro® /Premier
Synopsys® Design Compiler®
Mathworks® MATLAB® /Simulink®
Xilinx Vivado® Design Suite
または設計環境に応じて、Synplify Pro/Premier or その他論理合成ツール、RTL hand coding(VHDL or Verilog-HDL)などの設計環境が挙げられます。

課題

- 1. Level1:初心者向け

３×３の〇×を判定せよ

・ハードウェア設計のアウトライン

・シミュレーション用の verilog-HDL　ファイル

・ハードウェア設計のアウトラインの続き(ブロックの詳細)

- 2. Level2:中級者向け

素数を増やして任意の画像（黒白、グレースケール）

・ハードウェア設計のアウトライン

・シミュレーション用の verilog-HDL　ファイル

・ハードウェア設計のアウトラインの続き(ブロックの詳細)

- 3. Level3:上級者向け

unlimited (VAE なども歓迎いたします)

・ハードウェア設計のアウトライン

・シミュレーション用の verilog-HDL　ファイル

・ハードウェア設計のアウトラインの続き(ブロックの詳細)

審査：JUDGE

■　審査メンバーによる以下の 4 項目各 10 点で審査を実施（10 point each）

1）アカデミック的、新奇アイデア的な観点（Academic, New Idea）

2）実用設計、産業面応用的な観点（Used in real life, Good for industry）

3) FPGA 等の実装レベルの観点（Good prototype by FPGA etc.）

4）プレゼンテーションの観点（Good presentation）

表彰：AWARD

■　優勝（電子情報通信学会 SIS 賞）SIS AWARD

【アカデミック的、新奇アイデア的な観点】の BEST

■　その他、2)、3）、4)の観点からも賞を贈呈します。

手書き解答正誤判定システム

チーム名：ミヤウツー_EX デッキ
宮城洸利　西村泰星　新里侑之介　細野拓海

Handwritten answer accuracy judgment system

Miyagi Hiroto, Nishimura Taisei, Shinzato Yunosuke, Hosono Takumi
Department of Production Electronics and Information Systems Technology, Okinawa Polytechnic College
2994-2 Ikehara Okinawacity Okinawa, 904-2141, Japan
Email address：j2421319@okinawa-pc.ac.jp　j2421316@okinawa-pc.ac.jp
j2421308@okinawa-pc.ac.jp　j2421318@okinawa-pc.ac.jp

1. はじめに

昨今，小学校における教員不足が深刻な問題となっている．そのため，特に反復学習が求められる計算ドリルなどの課題において，教員が児童一人ひとりの解答を確認し，正誤判定を行うことが難しくなっている．これにより，児童が自分で学習を進めることができず，学力の定着に支障をきたす恐れがある．

また，ワシントン大学の研究[1]によれば，手書きでエッセイを書く小学生のほうが，より完成度の高い文章を作成し，読み方を習得する速度も速いことが判明した．手で文字を書く行為は，文字を書く能力，すなわち文字を認識する力を向上させ，その習得速度を加速させるためである．多くの小学校では，マークシート方式ではなく手書きでの解答を行っている．ただし，手書き解答では採点の自動化できず，採点時間がかかるという欠点がある．

これらの課題を解決するため，本システムでは，Variational Autoencoder（以下，VAE）を用いて，児童が手書きで解いた計算問題を自動的に正誤判定する仕組みを開発する．このシステムにより，児童は自分のペースで学習を進めることができ，教員は個別指導に割く時間を削減することが可能となる．

2. Variational Autoencorder について

本システムは手書き文字の認識に VAE を用いた．

VAE は，生成モデルの一種であり，入力データを低次元の潜在空間に圧縮し，そこから元のデータを再構成する自己符号化器（Autoencoder）の一形式である．VAE は，エンコーダとデコーダからなり，エンコーダで元の画像を潜在変数に落とし込む．その後，確率的な手法を用いて確率分布としてモデル化する．そして，デコーダで潜在変数から画像を生成し出力する（図 1 参照）．これにより，新しいデータの生成や異常検出，データの補完など多岐にわたる応用が可能になる．

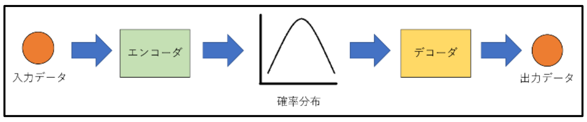


図 1. VAE のネットワーク構造

3. システム概要

今回作成する手書き解答正誤判定システム（以下システム）の概要を示す．

3.1 システムと操作について

あらかじめ Zybo Z7-20（以下 FPGA ボード）上に VAE 回路を作り，手書き文字の特徴を学習させる．

1．児童がボード上のボタン 0 を押下すると， PC 上に立ち上がっているソフトウェア（Tera Term）にランダムな計算式が出題される．
2．児童はホワイトボードとマーカを使用し，手書きで回答を行う．
3．Web カメラで解答を取得及び画像処理を行う．入力されたデータを SD カードに保存する．
4．SD カードを FPGA ボードに挿入する．
5．FPGA ボード上のボタン 1 を押下すると，VAE 回路で教師データから得られたデータとテストデータから得られたデータを比較し，正誤判定を行う．
6．結果を Tera Term に表示することで，利用者は，解答が正答又は誤答なのかを判断することができる．

本システムの開発環境を表 1 に示す．

表 1. 開発環境

OS	・Windows10
開発ソフト	・MATLAB 2020b ・Vivado 2020.2 ・Xilinx Vitis 2020.2 ・Tera Term
評価ボード	・DIGILENT 社製 ZYBO_Z7-20
言語	・MATLAB ・C 言語
その他	・ホワイトボード ・マーカ ・Web カメラ ・SD カード

3.2 シミュレーション

MATLAB 上で検証したシミュレーションについて以下に示す．教師データとテストデータの画像サイズは，FPGA ボードにデータを落とし込む際にできるだけ画像サイズを小さくするために，シミュレーションで使用する画像のサイズを MNIST の画像サイズの 28x28 を基準として，徐々に小さくしていったところ，6x6 のサイズが扱いやすいことが分かったため，6x6 をデータの画像サイズとした．

3.2.1 教師データの作成

あらかじめ 0 から 9 の手書き数字を 6 パターン作成する（図 2）．

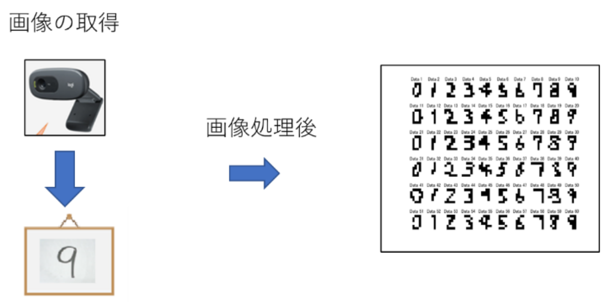


図 2. 教師データ作成の流れ

3.2.2 テストデータの作成

例えば「9」の手書き数字を入力し，画像処理を行う．6×6 ドットのサイズに圧縮する（図 3）．

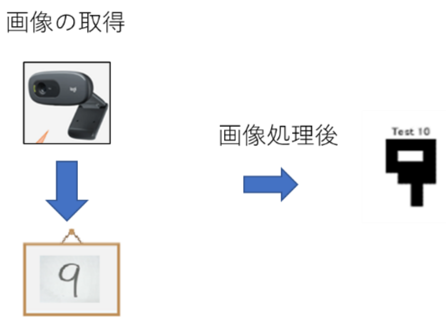


図 3. 「9」のテストデータ

3.2.3 教師データとテストデータを比較して判別する

入力されたテストデータ（図 4 右端）と近似値となる教師データを 3 パターン抽出する（図 4 中の左 3 つ）．判定方法には k 近傍法を使用した．

まず教師データとのユークリッド距離を計算し，距離が最も近い 3 つの教師データを抽出する．次に抽出された教師データのクラスラベルを取得し，最も頻出するクラスラベルを判定結果とする．ラベルをその画像が示す数字と一致させ，ラベルを用いて数字判定をしている．

判別結果を図 5 のように MATLAB に出力する．

*クラスラベルとは，教師データの画像 0〜9 につけたラベル（0〜9）である．

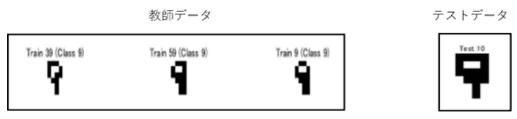


図 4. 左：教師データ，右：テストデータ

図 5. 判別結果

4. 実験及び検証

0 から 9 の数字をそれぞれ手書きし，正確に判別できるか検証した．

検証方法としてまず，手書きで 0〜9 の数字を各 1 桁ずつ入力する．その数字をテストデータとする．

次に，3.2 手順 3 で示した方法で教師データとテストデータを比較し判別する．

最後に，判別結果でテストデータと同じ数の場合は”○”異なる場合は”×”とした．それを 5 回行った結果が表 2 である．また，表 2 を基に導いた判別精度の結果を表 3 に示す．

表 2. それぞれの手書き数字の判別結果

	1回目	2回目	3回目	4回目	5回目
0	○	○	○	○	○
1	○	○	○	○	○
2	○	×	○	○	×
3	○	×	○	×	○
4	○	○	○	○	○
5	×	×	×	×	×
6	○	○	○	×	○
7	×	×	×	×	○
8	○	×	○	×	×
9	×	×	○	○	○

表 3. 数字ごとの判別精度

判別精度	数字
100%	0,1,4
80%	6
60%	2,3,9
40%	8
20%	7
0%	5

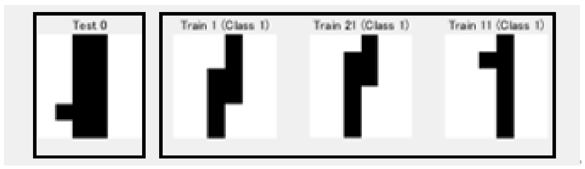


図 6. 1 の判別結果
（左：テストデータ　右：教師データ）

表 3 にある判別精度が 0%の「5」は手書き数字を読み込ませると, 図 8 のように教師データからの候補は「3」と「5」の 2 種になった. その中で , 最も頻出するクラスラベルが「3」なので, その結果, 「3」が出力された.

判別結果が低い理由として, 扱う情報量を少なくしたことで, 「3」と「5」のパターンが似た情報になったこと. さらに教師データの不足や手書き数字の書き方の違いなどがあげられる.

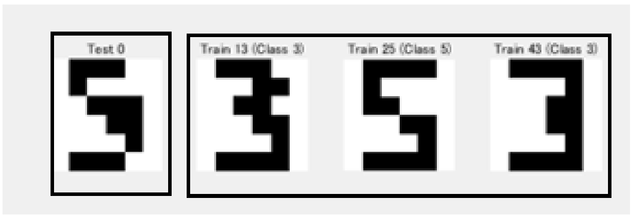


図 7.　4 回目の 5 の判別結果
（左：テストデータ　右：教師データ）

5. ハードウェアについて

5x5 の回路設計だと素子の占有率が 90%を超えており, 6x6 の規模は実装できないと考えたため 5x5 の回路作成を行っていたが, プログラムの実行結果が" nan(not a number)" となり計算値が出力できなかった. レジスタのアドレス設定やパラメータ調整などを行ったが改善されなかった. そのため, FPGA ボード上での VAE 実装と出題機能や解答機能などの作成ができなかった.

6. 追加で行ったシミュレーションについて

6.1 判定方法のパラメータ選定（k など）

最も適した k の値を決定するため, 数字を正しく判別できた割合を確認し, k の値を 3 とした. その理由として、k の値を 4 以上などに設定すると, 無駄なパターンが候補として選ばれる可能性が高く, 2 以下だと多数決で決定できないことがあげられる.

6.2 数字の切り取り（2 桁）と桁（十の位など）の判定

手書き数字が 2 桁の場合, それぞれの桁を正しく識別する必要がある. そこで, ラベリング処理を行い, オブジェクトごとに数字判別を行った.

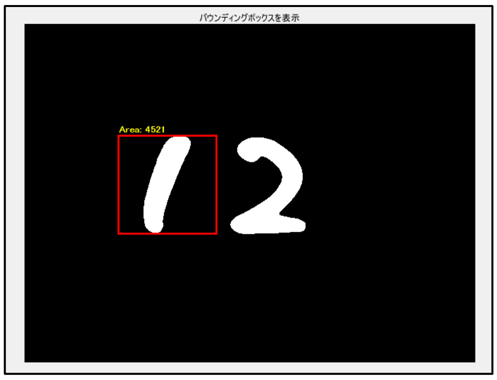


図 8.　十の位の判別

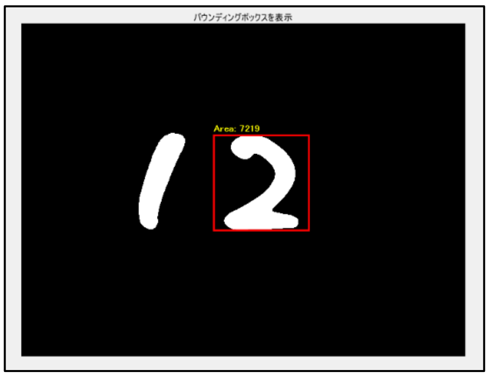


図 9.　一の位の判別

6.3 画像処理について

以下の手順で画像処理を行い, テストデータとした.

①手書き数字を撮影し, 画像を取得.
②グレースケール変換を行う.
③背景と数字を明確に分離するために, 二値化処理を行う.
④ノイズ除去を行う.
⑤画像から数字のみをトリミングするために, 背景を黒にする必要があるため, 白黒反転を行う.
⑥オブジェクトを囲う四角形を算出し, 正方形に調整した上で, 数字部分のトリミングを行う.
⑦白黒反転を行い, 背景を白に戻す.
⑧画像の型を logical 型から uint8 型に変換する.
⑨画像サイズを 6x6 にリサイズする.
⑩二値化処理を行う.

6.4 MNIST の中からデータを目視で抽出

手書き数字の判別精度を高めるため, 教師データのパターンを追加することにした.

元々の教師データは, ホワイトボードに数字を書き, それを撮影して画像処理を行い出力した画像を使用していたが, 実際に様々な数字のパターンを書いて撮影するのは時間がかかる.

このため, 追加する教師データは MNIST の手書き数字に画像処理を行い出力した画像を使用することにした. しかし, 画像処理後の MNIST の手書き数字（図

10）を見てみると,数字には見えないデータが多く見られることが分かる.

そのため,画像処理後のデータを人の目で見て数字と認識できるようなデータのみを選定し,教師データとして追加した.

教師データ追加後の判別精度は表 4 のような結果となった.表 3 と比べると,判別精度が 100%の数字は「1」のみと,少なくなっているが,最も低い判別精度でも 57.1%となっているため,全体的に精度は高くなったと考えられる.

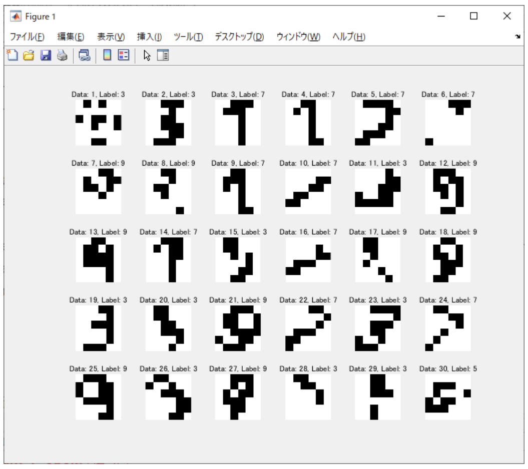


図 10. MNIST データのリサイズ(6x6)画像

表 4. 教師データ追加後の判別精度

数字	判定結果							判定成功率	数字
	1回目	2回目	3回目	4回目	5回目	6回目	7回目		
0	0	0	5	0	0	0	0	85.7	0
1	1	1	1	1	1	1	1	100.0	1
2	4	2	1	2	7	2	2	57.1	2
3	3	3	3	1	3	3	3	85.7	3
4	1	4	4	5	1	4	4	57.1	4
5	1	0	5	5	6	6	5	71.4	5
6	1	6	6	6	0	1	6	57.1	6
7	7	7	9	7	7	8	7	71.4	7
8	1	8	8	7	8	8	8	71.4	8
9	9	9	1	4	9	9	9	71.4	9

7. まとめ

現在,FPGA の VAE は 3x3 ができており,シミュレーションで行った 6x6 の回路サイズにするよう作業中である. 今回のレポートでは時間の都合により,FPGA ボード上の機械学習の実験は間に合わなかったが,各機能を MATLAB 上でシミュレーションし,動作確認することはできた.

今後は, すべての機能を統合した上で, 利用者がよりシステムを使いやすくなるような工夫を施していく考えである.

LSI コンテストに向けたシステム開発を通じて,Variational Autoencoder(VAE)についてより深く理解することができた.

今後の方針として MATLAB では数字の判別精度の学習向上のため判定方法や教師データの選定などを行う. FPGA では,VAE 回路の増設を行う予定である.

8. 参考資料

[1] 「記憶,情報整理,精神の治癒…… 手書きが結局,いちばん効果的な理由」
https://diamond.jp/articles/-/201900

HW/SW Co-Design of a Variational Autoencoder for Generating Hand Gesture Images

Team Machine Gun K

1st Keigo FUKUDA
Grad. School of Computer Science and Systems Eng.
Kyushu Institute of Technology
Iizuka, Japan
fukuda.keigo193@mail.kyutech.jp

2nd Mikito KAWANO
Kyushu Institute of Technology
Iizuka, Japan
kawano.mi@dsp.cse.kyutech.ac.jp

3rd Yuta HASHIMOTO
Kyushu Institute of Technology
Iizuka, Japan
hashimoto.yu@dsp.cse.kyutech.ac.jp

***Abstract*—In recent years, generative models utilizing AI have been actively applied in fields such as video generation, natural language processing, and image transformation, with various potential applications being explored. The Variational Autoencoder (VAE) model designed in this study is known as a powerful method for image generation and image transformation. The probabilistic generative process of VAE differs from simple data compression, providing robustness that allows it to adapt to diverse datasets. Additionally, compared to conventional generative methods, it enables more flexible representation.**

The theme of this study, HW/SW co-design, is a method used when implementing software functions in hardware. Hardware implementation is particularly effective when embedding systems into home appliances and automobiles, as it enables computational optimization, leading to reductions in processing time and circuit delay.

In this study, we designed a system using a VAE model that takes binary image data of rock-paper-scissors hand gestures as input and generates the corresponding hand images. The activation functions ReLU and Sigmoid were employed to prevent gradient vanishing, reduce computational cost, and stabilize binary image generation. The Adam optimization algorithm was used to ensure stable parameter optimization. Furthermore, the system was implemented based on HW/SW co-design, and it was confirmed that both software and hardware operations performed as expected.

I. Introduction

In recent years, generative AI technology has rapidly advanced, leading to innovative applications across various fields. In particular, generative models have been actively utilized in video generation and natural language processing. In the domain of video generation, deepfake technology, which has been recognized as a social issue, employs generative AI techniques [1]. In natural language processing, chatbots and OpenAI's ChatGPT leverage generative AI technology. Additionally, in manufacturing, image transformation techniques using generative AI have been applied to defect detection [2].

Generative AI is an algorithm that generates new data from input data. Among these methods, the Variational Autoencoder (VAE) is widely used as a powerful approach for learning probabilistic generative processes. The probabilistic nature of VAE ensures output diversity and enables adaptation to various data patterns. These characteristics allow VAE to achieve a higher level of expressiveness compared to conventional generative methods, making it suitable for a wide range of tasks.

Furthermore, many software-based AI systems, including generative AI, often face challenges in optimizing response speed and processing efficiency when compared to hardware implementations. For example, in automotive object recognition AI, real-time processing is crucial for accident prevention. In such cases, hardware implementation enables parallel processing for faster computation and circuit design optimization for reduced processing time and latency, thereby achieving high real-time performance.

In this study, we conduct an HW/SW co-design of a VAE model that takes binary image data of rock-paper-scissors hand gestures of size 32×32 as input. The system architecture is expanded to 784-680-180-40-180-680-784, incorporating the ReLU activation function to prevent gradient vanishing and reduce computational costs, as well as the Sigmoid activation function for binary image generation. Additionally, the Adam optimization algorithm is employed along with He initialization and mini-batch learning, ensuring stable training even for complex human hand data.

II. Neural Networks

A neural network is a type of machine learning model that processes input data through signal transmission and computes an output signal. The model evaluates predictions using the training data or its labels and updates parameters such as weights based on these evaluations. This iterative process, known as learning, allows the model to approximate predictions that match the training data.

In this study, deep learning is defined as unsupervised learning using deep neural networks. Hereafter, all neural networks described in this paper refer to those applying unsupervised learning.

A. Learning Algorithm of Neural Networks

The learning algorithm based on backpropagation is described using Fig.**??** as an example.

The input layer receives the training data $\boldsymbol{x}(=\boldsymbol{a}_0)$. The weighted sum $\boldsymbol{h}_1$ at the i-th unit of the hidden layer is expressed by Eq.(1).

$$\boldsymbol{h}_1(i) = \sum_{j=1}^{3} \boldsymbol{W}_1(i,j)\boldsymbol{a}_0(j) + \boldsymbol{b}_1(i) \tag{1}$$

The unit output $\boldsymbol{a}_1$ of the hidden layer is given by Eq.(2).

$$\boldsymbol{a}_1(i) = f_1(\boldsymbol{h}_1(i)) \tag{2}$$

The output layer processes the hidden layer's outputs similarly and produces the prediction $\boldsymbol{y}$.

$$\boldsymbol{h}_2(i) = \sum_{j=1}^{2} \boldsymbol{W}_2(i,j)\boldsymbol{a}_1(j) + \boldsymbol{b}_2(i) \tag{3}$$

$$\boldsymbol{y}(i) = f_2(\boldsymbol{h}_2(i)) \tag{4}$$

The error between the predicted value $\boldsymbol{y}$ and the corresponding training data $\boldsymbol{t}$ is used to update the weights $\boldsymbol{W}$ and biases $\boldsymbol{b}$ through backpropagation. Since the neural network in this study assumes unsupervised learning, the training data $\boldsymbol{t}$ is identical to the input data $\boldsymbol{x}$. The loss function $\mathcal{L}$ is defined by Eq.(5).

$$\mathcal{L} = \sum_{i=1}^{2} E\left(\boldsymbol{y}(i), \boldsymbol{t}(i)\right) \tag{5}$$

Next, to minimize the loss $\mathcal{L}$, the weights $\boldsymbol{W}$ and biases $\boldsymbol{b}$ of each unit are updated. The gradients of the loss function with respect to the parameters are computed by differentiation. The unit error δ_2 at the output layer is given by Eq.(6).

$$\delta_2(i) = \frac{\partial \mathcal{L}}{\partial \boldsymbol{h}_2(i)} = \frac{\partial \mathcal{L}}{\partial \boldsymbol{y}(i)} f_2'(\boldsymbol{h}_2(i)) \tag{6}$$

Based on the unit error δ_2, the weight gradient $\Delta\boldsymbol{W}_2$ and bias gradient $\Delta\boldsymbol{b}_2$ for the output layer are computed as follows:

$$\begin{aligned}\Delta\boldsymbol{W}_2(i,j) = \frac{\partial \mathcal{L}}{\partial \boldsymbol{W}_2(i,j)} &= \frac{\partial \mathcal{L}}{\partial \boldsymbol{h}_2(i)} \frac{\partial \boldsymbol{h}_2(i)}{\partial \boldsymbol{W}_2(i,j)} \\ &= \delta_2(i) \frac{\partial \boldsymbol{h}_2(i)}{\partial \boldsymbol{W}_2(i,j)}\end{aligned} \tag{7}$$

$$\begin{aligned}\Delta\boldsymbol{b}_2(i) = \frac{\partial \mathcal{L}}{\partial \boldsymbol{b}_2(i)} &= \frac{\partial \mathcal{L}}{\partial \boldsymbol{h}_2(i)} \frac{\partial \boldsymbol{h}_2(i)}{\partial \boldsymbol{b}_2(i)} \\ &= \delta_2(i) \frac{\partial \boldsymbol{h}_2(i)}{\partial \boldsymbol{b}_2(i)}\end{aligned} \tag{8}$$

Based on the unit error δ_2, the weight gradient $\Delta\boldsymbol{W}_2$ and bias gradient $\Delta\boldsymbol{b}_2$ for the output layer are computed as follows:

$$\Delta\boldsymbol{W}_2(i,j) = \delta_2(i) \frac{\partial \boldsymbol{h}_2(i)}{\partial \boldsymbol{W}_2(i,j)} \tag{9}$$

$$\Delta\boldsymbol{b}_2(i) = \delta_2(i) \frac{\partial \boldsymbol{h}_2(i)}{\partial \boldsymbol{b}_2(i)} \tag{10}$$

Similarly, the unit error δ_1 of the hidden layer, the weight gradient $\Delta\boldsymbol{W}_1$, and the bias gradient $\Delta\boldsymbol{b}_1$ are computed as follows:

$$\delta_1(i) = \sum_{j=1}^{2} \delta_2(j)\boldsymbol{W}_2(j,i) f_2'(\boldsymbol{h}_1(i)) \tag{11}$$

$$\Delta\boldsymbol{W}_1(i,j) = \delta_1(i) \frac{\partial \boldsymbol{h}_1(i)}{\partial \boldsymbol{W}_1(i,j)} \tag{12}$$

$$\Delta\boldsymbol{b}_1(i) = \delta_1(i) \frac{\partial \boldsymbol{h}_1(i)}{\partial \boldsymbol{b}_1(i)} \tag{13}$$

The parameters are updated based on their gradients to minimize the loss function $\mathcal{L}$. Since minimizing the loss requires moving in the opposite direction of the gradient, the update equations for each parameter are expressed as follows:

$$\boldsymbol{W}_2(i,j) \leftarrow \boldsymbol{W}_2(i,j) - \eta\Delta\boldsymbol{W}_2(i,j) \tag{14}$$

$$\boldsymbol{b}_2(i) \leftarrow \boldsymbol{b}_2(i) - \eta\Delta\boldsymbol{b}_2(i) \tag{15}$$

$$\boldsymbol{W}_1(i,j) \leftarrow \boldsymbol{W}_1(i,j) - \eta\Delta\boldsymbol{W}_1(i,j) \tag{16}$$

$$\boldsymbol{b}_1(i) \leftarrow \boldsymbol{b}_1(i) - \eta\Delta\boldsymbol{b}_1(i) \tag{17}$$

This process is repeated iteratively to train the network.

B. ReLU Function

The ReLU (Rectified Linear Unit) function is widely used as an activation function in the hidden layers of neural networks. It outputs the input value itself if it is greater than or equal to zero; otherwise, it outputs zero. The ReLU function is defined in Eq.(18), and its derivative is given in Eq.(19).

$$f(x) = \max(0,x) = \begin{cases} x & \text{if } x \geq 0, \\ 0 & \text{if } x < 0. \end{cases} \tag{18}$$

$$f'(x) = \begin{cases} 1 & \text{if } x \geq 0, \\ 0 & \text{if } x < 0. \end{cases} \tag{19}$$

C. Sigmoid Function

The Sigmoid function is commonly used as an activation function in the output layer of neural networks. The Sigmoid function maps the input into the range of $(0, 1)$. It is defined in Eq.(20), and its derivative is given in Eq.(21).

$$f(x) = \frac{1}{1 + \exp(-x)} \tag{20}$$

$$f'(x) = \frac{\exp(-x)}{(1 + \exp(-x))^2} = f(x)(1 - f(x)) \tag{21}$$

D. Weight Initialization

In this study, He Initialization [5] is used for weight initialization. He initialization is a technique suitable for stabilizing the convergence of loss in neural networks utilizing the ReLU activation function. The weight initialization for the connection between layer $l-1$ and layer l is given by Eq.(22).

$$\boldsymbol{W}_l(i,j) \sim \mathcal{N}\left(0, \frac{2}{U_{l-1}}\right) \tag{22}$$

E. Mini-Batch Learning

Mini-batch learning is a training method where N training data samples are randomly shuffled and divided into m groups of B data points (mini-batches). Each training iteration processes one mini-batch, and the parameters are updated based on the mini-batch error $\mathcal{L}_m$. After all mini-batches have been processed once, the training data is reshuffled, and learning continues in the same manner.

The advantage of mini-batch learning is that it enables stable learning while reducing computational costs, as the entire dataset is not used in a single iteration. Additionally, since each mini-batch contains data with different characteristics, the risk of getting stuck in local optima is reduced.

F. Adam Optimization

Adam (Adaptive Moment Estimation) [6] is an optimization algorithm that combines the properties of Momentum and RMSProp. Adam updates parameters using the first moment m and the second moment v. Given the parameter to be updated θ and its gradient with respect to the loss $\mathcal{L}$, denoted as $\Delta\theta$, the update equations for the first and second moments at step t are given by:

$$m_t = \beta_1 m_{t-1} + (1 - \beta_1)\Delta\theta_t \tag{23}$$

$$v_t = \beta_2 v_{t-1} + (1 - \beta_2)\Delta^2\theta_t \tag{24}$$

where $\beta_1, \beta_2 \in [0, 1)$ are decay coefficients used for smoothing the moments. The second moment's second term is defined as $\Delta^2\theta = \Delta\theta \cdot \Delta\theta$.

The first and second moments are bias-corrected using:

$$\hat{m}_t = \frac{m_t}{1 - \beta_1^t} \tag{25}$$

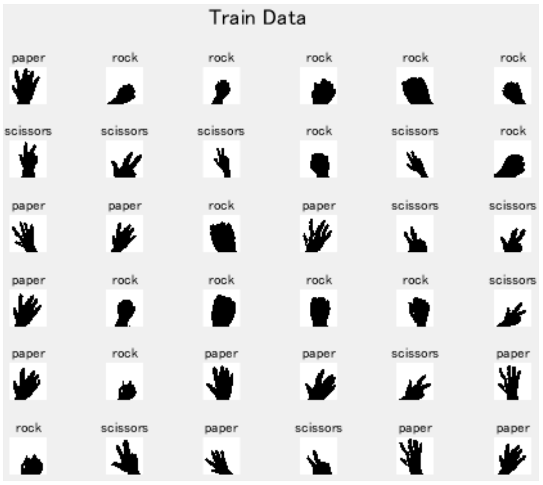


Fig. 1. Training Data

$$\hat{v}_t = \frac{v_t}{1 - \beta_2^t} \tag{26}$$

Finally, the parameter update rule for θ is given by:

$$\theta_t = \theta_{t-1} - \eta \frac{\hat{m}_t}{\sqrt{\hat{v}_t} + \epsilon} \tag{27}$$

where ϵ is a small constant (typically 10^{-8}) to prevent division by zero.

III. Proposed Model Overview

This section describes the dataset used in this study, as well as the learning and testing models. The learning model in this study is implemented using MATLAB 2024b.

A. Dataset

The dataset used in this model consists of images of hands from nine participants, where each participant captured five instances of rock, paper, and scissors gestures. Each image is sized at 32×32 [px].

1) Data Preprocessing: To enhance the dataset while maintaining the image size, transformations such as scaling and rotation are applied. This process includes padding after downscaling and cropping after upscaling, ensuring that the final image size remains 32×32 [px].

A total of 8192 images are used for training after preprocessing. Fig.1 presents a sample of the training data.

B. Learning Model

The structure of the VAE model designed in this study for generating rock-paper-scissors hand images is shown in Fig.2. The layers are numbered starting from the input layer as layer 0, and the number of units in each layer is set as follows:

- Layer 0 (Input Layer): 784 units
- Layer 1: 680 units
- Layer 2: 180 units
- Layer 3: 40 units
- Layer 4: 40 units

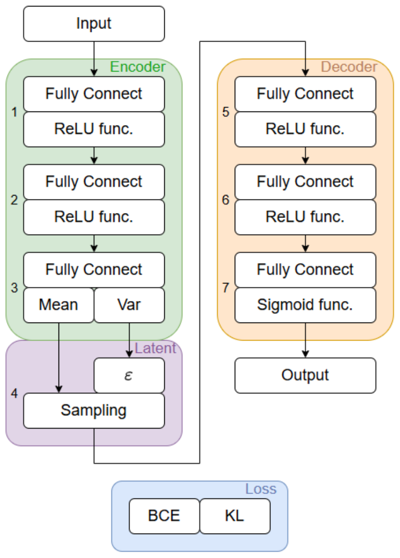


Fig. 2. Model Overview

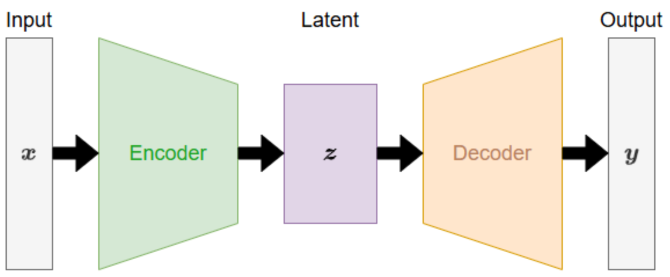


Fig. 3. AEś Overview

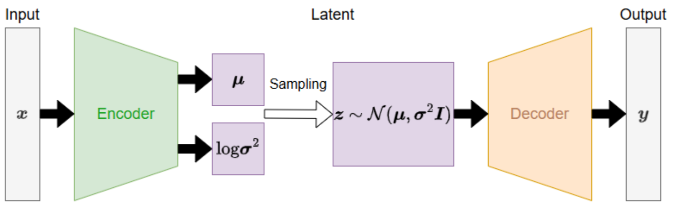


Fig. 4. Overview of VAE with Sampling

- Layer 5: 180 units
- Layer 6: 680 units
- Layer 7 (Output Layer): 784 units

He initialization is used for weight initialization, and Adam is employed for parameter updates. Additionally, the input data is transformed from the range $[0, 1]$ to $[-1, 1]$ for centering around zero.

The model is trained with 8192 images (2^{13}) over 3000 iterations. Since mini-batch learning is used, each training step processes a batch of 64 images (2^5).

1) Overview of Variational Autoencoder (VAE): First, the fundamental concept of an Autoencoder (AE) is discussed.

An Autoencoder (AE) is a generative algorithm that performs dimensionality reduction to reconstruct input data. The structure of AE is shown in Fig.3. AE consists of a bottleneck structure where an encoder transforms input data $\boldsymbol{x}$ into lower-dimensional latent variables $\boldsymbol{z}$, and a decoder reconstructs $\boldsymbol{z}$ back into an output prediction $\boldsymbol{y}$ of the same dimension as $\boldsymbol{x}$.

A Variational Autoencoder (VAE) is a generative algorithm similar to AE, incorporating an encoder and decoder. Fig.4 presents the VAE structure. The key difference from AE is that VAE assumes the following conditions for latent variables:

- The latent variables $\boldsymbol{z}$ follow a probability distribution.
- The input data $\boldsymbol{x}$ follows a conditional probability distribution.

To ensure the latent variables follow a probability distribution, VAE includes a mean layer and a variance layer. Since variance $\boldsymbol{\sigma}^2$ is constrained to non-negative real numbers $\mathbb{R}_{\geq 0}$, this study applies log variance $\log \boldsymbol{\sigma}^2$ to extend its domain to all real numbers $\mathbb{R}$.

2) Loss Function: The loss function $\mathcal{L}$ of VAE is obtained as the sum of the following two terms:

- **Reconstruction Loss** E_{rec} : Measures the deviation between the predicted output $\boldsymbol{y}$ and the training data $\boldsymbol{t}$.
- **Regularization Term** E_{reg} : Measures the divergence of the latent variable $\boldsymbol{z}$.

$$\mathcal{L} = E_{rec}(\boldsymbol{y}, \boldsymbol{t}) + E_{reg}(\boldsymbol{\mu}, \boldsymbol{\sigma}) \tag{28}$$

The first term in Eq.(28), the reconstruction loss E_{rec}, is expressed as Eq.(30). Here, U_O represents the number of units in the predicted output $\boldsymbol{y}$.

$$E_{rec}(\boldsymbol{y}, \boldsymbol{t}) = -\sum_{i=1}^{U_O} (\boldsymbol{t}(i) \log \boldsymbol{y}(i) + (1 - \boldsymbol{t}(i)) \log(1 - \boldsymbol{y}(i))) \tag{29}$$

The reconstruction loss E_{rec} is defined using Binary Cross Entropy (BCE), as shown in Eq.(30). This is appropriate since the training data $\boldsymbol{t}$ consists of binary values, and the predicted output $\boldsymbol{y}$ is in the range $\mathbb{R}_{(0,1)}$.

$$E_{rec}(\boldsymbol{y}, \boldsymbol{t}) = -\sum_{i=1}^{U_O} (\boldsymbol{t}(i) \log \boldsymbol{y}(i) + (1 - \boldsymbol{t}(i)) \log(1 - \boldsymbol{y}(i))) \tag{30}$$

The regularization term E_{reg} is defined using Kullback-Leibler divergence (KL divergence), as shown in Eq.(31). Here, U_L represents the number of latent variable units.

$$E_{reg}(\boldsymbol{\mu}, \boldsymbol{\sigma}) = -\frac{1}{2} \sum_{i=1}^{U_L} \left(1 + \log \boldsymbol{\sigma}^2(i) - \boldsymbol{\mu}^2(i) - \boldsymbol{\sigma}^2(i)\right) \tag{31}$$

The regularization term ensures that the latent variable distribution approximates a standard normal distribution (mean $\boldsymbol{\mu}(i) = 0$, standard deviation $\boldsymbol{\sigma}(i) = 1$), achieving a minimum value of zero under these conditions.

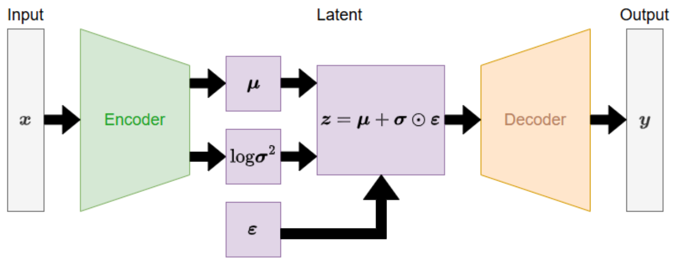


Fig. 5. Overview of VAE using the Reparameterization Trick

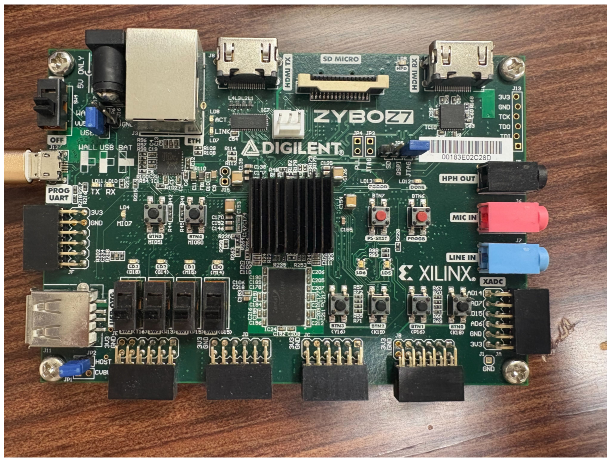


Fig. 6. ZYBO ZYNQ-7020

3) Reparameterization Trick: Fig.4 includes a sampling operation, which is non-differentiable and thus prevents learning via backpropagation. To enable differentiability, we apply a technique known as the Reparameterization Trick. The structure of a VAE incorporating this trick is shown in Fig.5.

By generating random noise ε from a standard normal distribution $\mathcal{N}(0, 1)$, we can approximate the latent variable z as shown in Eq.(32). This enables differentiable sampling.

$$z(i) = \mu(i) + \sigma(i)\varepsilon(i) \tag{32}$$

IV. HW/SW Co-Design Experimental Validation

This section describes the validation of the designed VAE learning model and the obtained results.

A. Software Processing

To implement part of the system, MATLAB R2024b was used to compute up to the third layer, and the results were used as input data for the hardware implementation.

B. Hardware Implementation

The hardware implementation was conducted using MATLAB R2022b Simulink, Vivado 2020.2, and Vitis. The ZYBO ZYNQ-7020 (XC7Z020-1CLG400C) evaluation board was used for hardware implementation. The board used is shown in Fig.6.

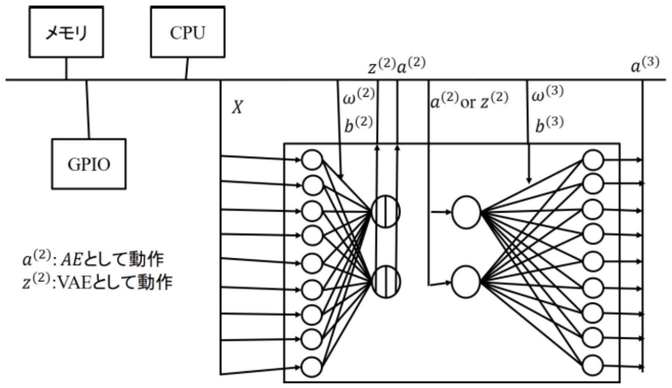


Fig. 7. Hardware Design [3]

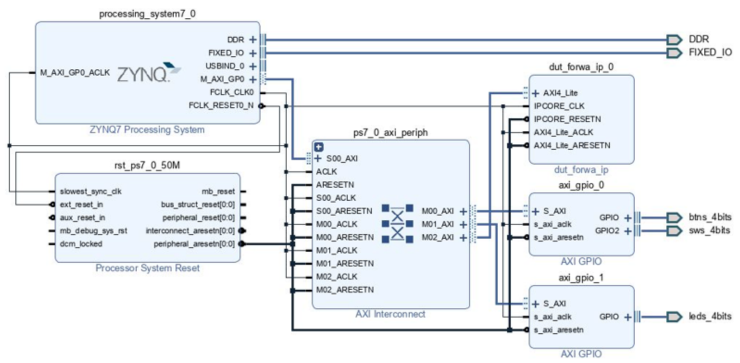


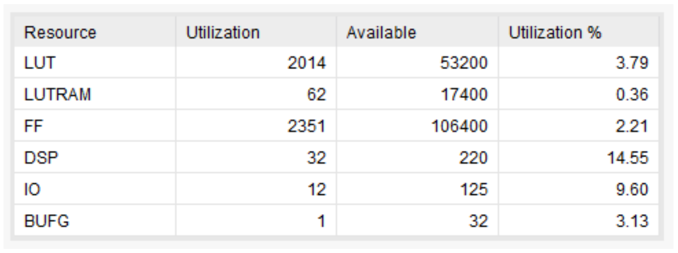
Fig. 8. Block Diagram

1) Hardware Design: The blocks used for the hardware implementation are shown in Fig.7. The block diagram is presented in Fig.8, and the hardware resource utilization is shown in Fig.9. These designs were created using MATLAB Simulink and Vivado 2020.2. In this study, the 3rd, 4th, and 5th layers of the model in Fig.2 were implemented in hardware.

2) Partitioned Implementation: The partitioned implementation method is illustrated in Algorithm1 using the number of units on the encoder side as an example. The decoder side can also be implemented using the same method.

3) Evaluation of the proposed method: The test dataset is input into the model to verify its proper operation on the board. The test dataset consists of 1000 images obtained by capturing five images of hand gestures (rock, paper, scissors) from individuals not included in the training dataset, followed by data augmentation. The process from inputting the test data to generating the output images is examined, and a comparison between input and output images is performed.

The test dataset is shown in Fig.10, and the test results are shown in Fig.11. From Fig.11, it is confirmed that the model successfully learns the shape of hand gestures and generates their approximate contours appropriately. Additionally, the classification accuracy of rock-paper-scissors recognition in this test was 0.980.



Resource	Utilization	Available	Utilization %
LUT	2014	53200	3.79
LUTRAM	62	17400	0.36
FF	2351	106400	2.21
DSP	32	220	14.55
IO	12	125	9.60
BUFG	1	32	3.13

Fig. 9. Hardware Resource Utilization

Algorithm 1 Partitioned Implementation: Encoder Section
Model unit count: $M_I = 180, M_O = 40$, partitioned unit count: $N_I = 9, N_O = 2$
(I: Input Layer, O: Output Layer)
Encoder Input: $X \in \mathbb{R}^{N_I}$, Weights $\omega \in \mathbb{R}^{N_O \times N_I}$, Bias $b \in \mathbb{R}^{N_O}$
Encoder Output: Pre-activation $Z \in \mathbb{R}^{N_O}$, Post-activation $Y \in \mathbb{R}^{N_O}$

```
Require: $\boldsymbol{x} \in \mathbb{R}^{M_I}$
Require: $\boldsymbol{W} \in \mathbb{R}^{M_O \times M_I}, \boldsymbol{b} \in \mathbb{R}^{M_O}$
Ensure: $\boldsymbol{y} \in \mathbb{R}^{M_O}$
 1: for $i \leftarrow 1$ to $M_H/N_H$ do
 2:     $K^{\text{begin}} \leftarrow N_H(i-1)+1$
 3:     $K^{\text{end}} \leftarrow N_H i$
 4:     $b \leftarrow \{\boldsymbol{b}(k)\}_{k=K^{\text{begin}}}^{K^{\text{end}}}$
 5:     for $j \leftarrow 1$ to $M_I/N_I$ do
 6:         $L^{\text{begin}} \leftarrow N_I(j-1)+1$
 7:         $L^{\text{end}} \leftarrow N_I j$
 8:         $X \leftarrow \{\boldsymbol{x}(l)\}_{l=L^{\text{begin}}}^{L^{\text{end}}}$
 9:         $\omega \leftarrow \{\boldsymbol{W}(k,l)\}_{k=K^{\text{begin}},l=L^{\text{begin}}}^{K^{\text{end}},L^{\text{end}}}$
10:         $Z, Y \leftarrow$ Encoder$(X, \omega, b)$ (Perform Encoder computation)
11:         $b \leftarrow Z$
12:     end for
13:     $\{\boldsymbol{y}(k)\}_{k=K^{\text{begin}}}^{K^{\text{end}}} \leftarrow Y$
14: end for
```

V. Conclusion

In this study, we conducted HW/SW co-design for a system that generates images of rock-paper-scissors hand gestures. Specifically, we introduced the Adam optimization algorithm, He initialization, and mini-batch learning into the VAE model to ensure stable training. Additionally, ReLU was applied to the hidden layer activation function to prevent gradient vanishing and reduce computational cost, while Sigmoid was employed in the output layer activation function to optimize loss calculation for binary data output.

The designed system was validated to evaluate whether it achieves sufficient performance as a generative AI and whether it functions correctly on hardware. A comparison of input and output images confirmed that the system successfully generated corresponding hand shapes, achieving a remarkably high classification accuracy of 0.980. This result indicates that the system operates as intended on hardware and that HW/SW co-design was effectively implemented.

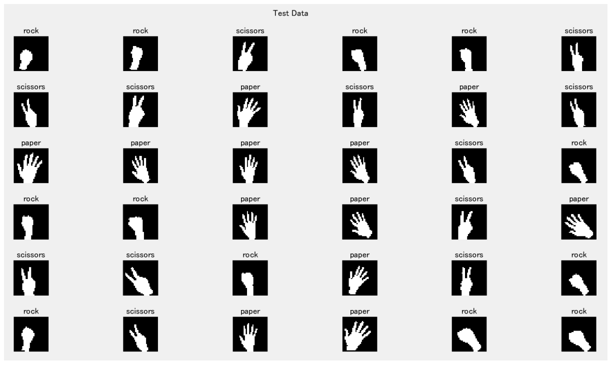


Fig. 10. Test Dataset

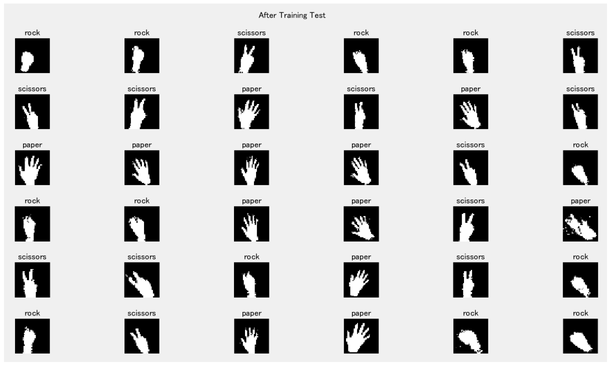


Fig. 11. Generated Images (After Training)

Future work includes extending HW/SW co-design to the entire system. In this study, HW/SW co-design was implemented only for the 3rd, 4th, and 5th layers. To integrate the entire system, it will be necessary to employ partitioned implementation and variable file storage methods. Additionally, further improvements in generation accuracy may require employing more powerful feature extraction methods such as convolutional neural networks (CNNs).

References

[1] Y. Kashiwamura, "Deep Fake Threat - Shocking images created by the latest AI," Dai-ichi Life Research Institute, Inc. Available: https://www.dlri.co.jp/pdf/ld/2018/wt1810b.pdf, Accessed: Feb. 13, 2025.
[2] Miyagi Prefecture, Japan, "Image processing technology using AI to reduce manpower for visual inspections at manufacturing sites," Available: https://www.pref.miyagi.jp/documents/22979/788626.pdf, Accessed: Feb. 13, 2025.
[3] R. Moriyama, "A Study on Variational Autoencoders with HW/SW Co-Design," (in Japanese), Kyushu Institute of Technology, Graduation Thesis, Feb. 2023.
[4] R. Okuyama, "Reduction of computational complexity of tumor detection using FCN and its implementation," (in Japanese), Kyushu Institute of Technology, Graduation Thesis, Feb. 2023.
[5] K. He, X. Zhang, S. Ren, and J. Sun, "Delving Deep into Rectifiers: Surpassing Human-Level Performance on ImageNet Classification," arXiv:1502.01852v11, Feb. 6, 2015.
[6] D. P. Kingma and J. Lei Ba, "Adam: A Method for Stochastic Optimization," arXiv:1412.6980v9, Jan. 30, 2017.

摩耗劣化パターンの自動判断システム
―VAE による摩耗パターン解析の応用―

1st Shintani Yuhi
Kyusyu Politechnic College
Department of Electronics and Information Tecnology
Fukuoka, Japan
2317117@kyushu-pc.ac.jp

2nd Miyaoka Yoshihiro
Kyusyu Politechnic College
Department of Electronics and Information Tecnology
Fukuoka, Japan
2317136@kyushu-pc.ac.jp

I. はじめに

履物（靴やインソール）は，使用頻度や環境条件に応じて摩耗が進行し，その進行度合いは個人の歩行パターンや使用環境に依存する特性を有する。この摩耗の進行は，下肢のバイオメカニクスや姿勢制御に影響を及ぼす可能性があり，その定量的評価と早期異常検知が不可欠である。

本研究では，インソールの摩耗状態を LED によって定量的に可視化し，使用者が客観的に評価可能なシステムを提案する。本システムは，歩行動態の最適化や適切な履物選択の指標を提供し，運動器障害や疲労蓄積のリスク低減，さらには運動パフォーマンスの向上に寄与することを目的としている。また，本技術は職業性負荷の生体力学的評価にも応用可能である。

本研究の中核は,Variational Autoencoder (以下，VAE)を用いたインソール摩耗状態の自動判断システムの開発である。本システムでは，インソールの画像を入力として使用し，摩耗状態を解析して異常部位を特定する。VAE は画像特徴量を潜在空間に圧縮し再構成画像を生成する。この再構成過程において，非定形的な摩耗パターンや異常部位は再構成画像と入力画像の差分として顕在化するため，この差分解析により異常検知を実現する。

本システムにより，歩行動態の詳細な分析が可能となり，使用者の運動器障害や疲労リスクの有用であるのみにならず，スポーツ科学分野における運動パフォーマンス向上や職業性負荷の定量的評価にも応用可能である。

II. 概要

A. システム概要

図 1 に示したものが，インソールの摩耗具合を可視化する装置である。本装置は，2 色 LED が実装され，摩耗度に応じて赤色(重度摩耗)，橙色(中度摩耗)，緑色(軽度摩耗)の 3 段階で状態を表示する。

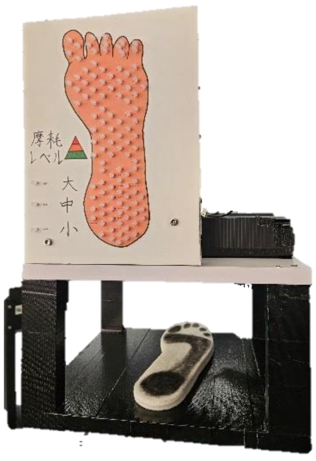


図 1. 足形表示機器外観

本システムの特徴は，個別 LED 制御ではなく，画像解析に基づく領域別摩耗の可視化手法を採用している点にある。具体的には，図 2 に示す 10 箇所の解剖学的領域に対して，個別に信号出力が可能な ZYBO 接続回路を実装している。

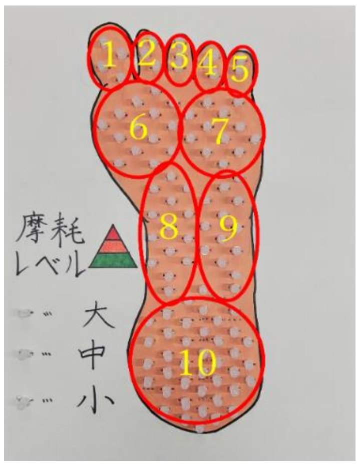


図 2. 部位分類

本装置の摩耗解析アルゴリズムでは，USB カメラから取得した画像を VAE に入力として供給し，事前に学習させた新品のインソールに近づけることで摩耗具合の差分をとる。その後，異常検知した部位に対応した LED の制御信号を生成する。

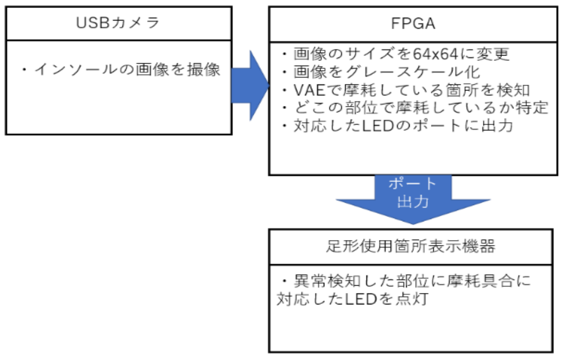
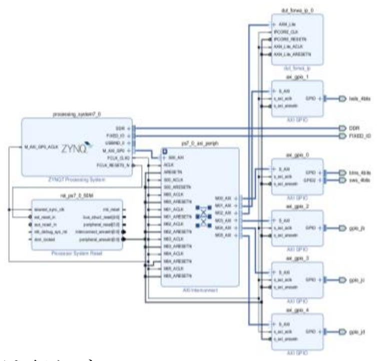

USBカメラ
・インソールの画像を撮像
FPGA
・画像のサイズを64x64に変更
・画像をグレースケール化
・VAEで摩耗している箇所を検知
・どこの部位で摩耗しているか特定
・対応したLEDのポートに出力
ポート出力
足形使用箇所表示機器
・異常検知した部位に摩耗具合に対応したLEDを点灯

図 3. システム構成図

図 3 に本システムの構成図を示す。本システムのハードウェア構成は，FPGA ボード，カスタム設計された基盤，および 2 色 LED アレイから構成される。ZYBO が演算処理を担当し，中央部から側面に配置された LED 出力インターフェースを制御するハードウェアロジックを実装している。我々が作成した, FPGA の構成を図 4 に示す。



① VAE の処理を行うブロック
② 2 色 LED を制御するためのブロック
③ FPGA 内部のデバック用の SW, BTN のブロック
④ FPGA 内部のデバック用の LED のブロック

図 4. FPGA の構成

また，今回のハードウェアサイズも合わせて以下に示す。

表 1. ハードウェアサイズ

Resource	Utilization	Available	Utilization %
LUT	4128	17600	23.45
LUTRAM	62	6000	1.03
FF	4344	35200	12.34
BRAM	2.5	60	4.17
DSP	72	80	90
IO	24	100	24
BUFG	1	32	3.13

B. *画像入力部*

USB カメラから取得した画像は，前処理としてグレースケール化および画像のサイズ変更を施す。これにより，VAE での画像処理の負荷を軽減することができる。よって，学習効率と実行速度を向上させられる。

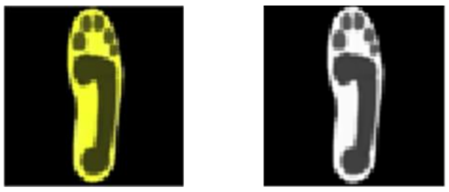
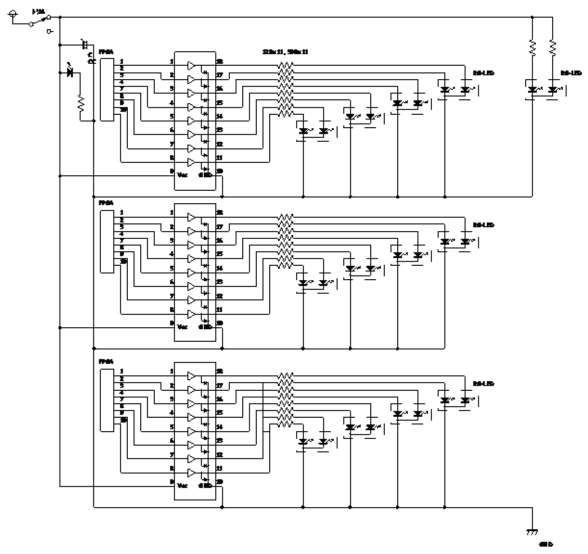


図 5. インソールのグレースケール化

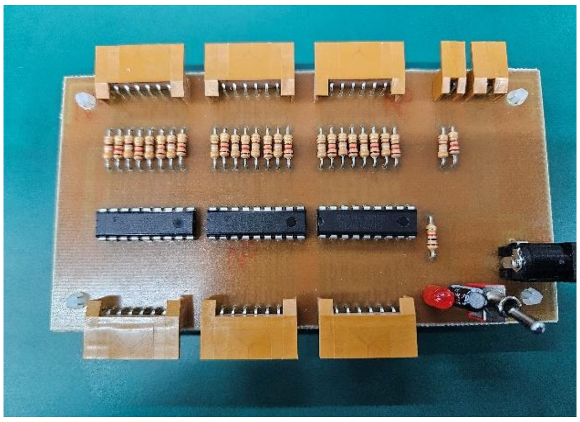
C. *処理制御部*

前処理された画像は FPGA ボード上で処理される FPGA に実装された VAE によってインソール画像を入力データとして読み込み，新品のインソールとの差分から LED 処理を実行する。

図 6 に LED 制御するトランジスタアレイを使った増幅回路を示す。



(a) LED 増幅制御回路

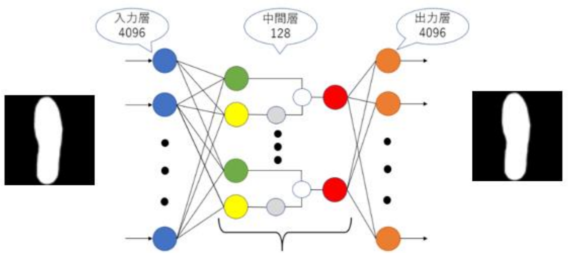


(b) LED 増幅基板

図 6. 増幅回路

D.　ハードウェア化

VAE のハイパーパラメータは，図 7 に示す最終的な値を使用した。また学習回数は[20000 回]，学習率は[0.0001]である。



IP 名: dut_forwa_ip_0

図 7.　ハードウェア化

III.　実験

A.　実験内容

本研究では，USB カメラから取得した画像データを 64x64 ピクセルにリサイズし，グレースケール変換後，FPGA に実装された VAE によってインソールの摩耗状態を定量的に評価した。評価結果は 2 色 LED アレイによって摩耗箇所および程度を可視化する。

VAE アルゴリズムの最適化過程では，MATLAB 環境下で以下の手順による実験を実施した。

① 画像データの前処理(グレースケール変換およびインソール領域分割)
② 入力データの可視化と検証

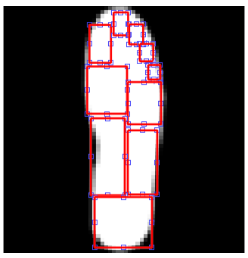


図 8.　部位の範囲指定

③ 解剖学的指標に基づく 10 領域(母指，示指，中指，環指，小指，母指球，中足骨，足弓外，側縁，踵部)のマスク生成。図 8 参照
④ 64×64 ピクセルの画像データの 4096 次元ベクトルへの変換
⑤ ニューラルネットワークアーキテクチャ構築
⑥ 学習データセットによる VAE の最適化
⑦ 再構成誤差の定量的評価
⑧ 閾値に基づく異常検知の実施

以上のステップを行った。

B.　実験結果

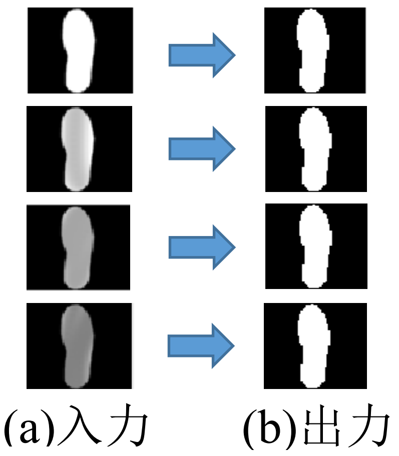


図 9.　学習データを VAE に入力した結果

図 9(a)に示す学習データセット(新品インソールおよび証明条件を考慮したバリエーション，計 4 サンプル)を用いて VAE を訓練した。図 9(b)は学習データの再構成結果を示し，環境要因の影響を排除した理想的な状態の再構成に成功していることが確認された。

学習過程の収束特性を図 10 に示す。誤差関数は学習の進行に伴い単調減少し，初期の急速な収束後，漸近的に安定状態に達している。ただし，限定的な学習データセットによる過学習の可能性は考慮すべき課題である。

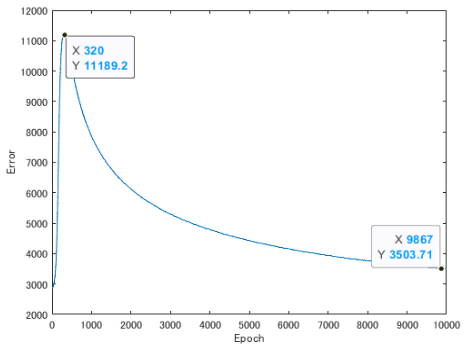


図 10.　学習率のグラフ

図 11 は未知のテストデータに対する異常検知結果を示す。摩耗度は 3 段階(重度:赤，中度:橙，軽度:緑)で可視化され，入力画像と再構成画像間の差分解析により摩耗領域が明確に検出された。

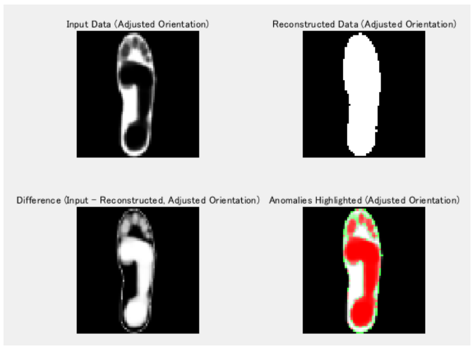


図 11. 差分による異常検知

図 12 は解剖学的領域ごとの摩耗評価結果を示す各領域に対するマスク処理により，部位特異的な摩耗パターンの定量的評価が可能になった。

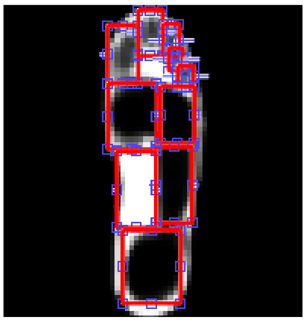


図 12. マスク適用

C. 検証

図 13 に示す実物インソールを用いて実証実験を実施した。USB カメラにより取得した摩耗インソールの画像データを 64×64 ピクセルにリサイズし，グレースケール変換を適用した。

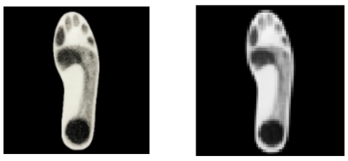


図 13. 実物・前処理を施したインソール

VAE の再構成特性により，環境ノイズおよび照明の影響が効果的に抑制され，摩耗箇所の識別精度が向上した。

図 14 は異常箇所の空間分布及び程度の定量的評価結果を示している。

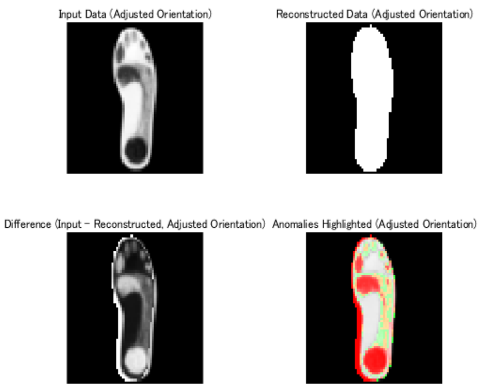


図 14. 実物インソールを用いた異常検知

本手法により，インソールの摩耗状態を高精度で検知可能なシステムが実現された。特に，摩耗を異常状態として捉える本アプローチは，リアルタイム摩耗判断システムとしての実用性を示唆している。

図 15 は解剖学的領域別の摩耗評価結果を足形表示デバイスにマッピングした結果を示す。

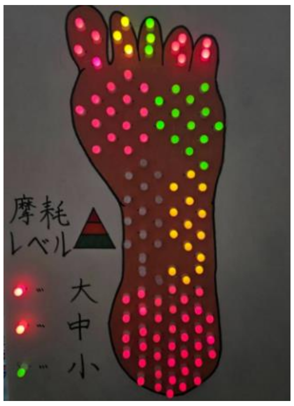


図 15. 足形表示デバイス

IV. おわりに

本研究では，VAEを基盤としたインソール摩耗状態の可視化システムを提案し，その有効性を実証した。本システムは，歩行動態の最適化や運動器の健康管理に寄与するのみならず，スポーツ科学や産業保健分野における応用可能性を示唆している。今後の研究課題として，摩耗パターンと環境アーティファクトの識別精度向上，および暗色系インソールへの適用性拡大が挙げられる。これらの技術的課題の克服により，より堅牢で汎用性の高い摩耗診断システムの実現が期待される。

さらに，本システムの臨床応用に向けて，多様な年齢層や歩行パターンを持つ被験者データの蓄積が必要である。また，機械学習モデルの継続的な改良により，個人差や使用環境の違いに対する適応性を向上させることが重要である。特に，インソールの摩耗状態と歩行時の負荷分布の相関を定量的に解析し，モデルに組み込むことで，より精度の高い診断結果が得られると考えられる。また，応答時間の短縮やリアルタイ

ム解析の実現に向けたハードウェアの最適化も課題として挙げられる。

加えて，IoTプラットフォームとの統合により，長期的な摩耗進行の追跡や予防的なメンテナンス推奨システムへの発展も期待される。これにより，医療機関やスポーツ施設とのデータ共有が可能となり，利用者の健康状態を包括的に管理するエコシステムの構築が視野に入る。また，クラウドベースのデータ処理を導入することで，地域や集団レベルでの動態データの統計分析が可能となり，歩行リスクの予測モデル開発や健康政策の立案に寄与する可能性がある。

さらに，本システムをウェアラブルデバイスと連携させることで，より直感的で使いやすいユーザーインターフェースの提供が可能となる。これにより，日常生活の中での利用が促進され，利用者の行動変容や健康意識の向上も期待される。このような技術的・機能的な拡張を通じて，歩行健康管理における包括的なソリューションとしての発展が見込まれる。

参考文献

[1] "LSI Design Contest" LSI Design Contest Committee. [Online]. Available: http://www.lsi-contest.com/. [Accessed: Feb. 25, 2025]

回路のレイアウトを確認するシステム

チーム名：Taki

照屋寛武　高安柊衣　大城紀裕

－Circuit Layout Checker System－

Teruya Hiromu, Takayasu Toui, Osiro Norihiro

Department of Production Electronics and Information Systems Technology, Okinawa Polytechnic College

2994-2 Ikehara Okinawacity Okinawa, 904-2141, Japan

Email address：j2421313@okinawa-pc.ac.jp　j2421311@okinawa-pc.a c.jp

j2421304@okinawa-pc.ac.jp

Abstract—We developped a system to streamline the verification of component placement on printed circuit boards (PCBs). The system determines whether the component placement is correct and visualizes any errors by using a Variational Autoencoder (VAE). The Zynq-7 Z7-20 FPGA is used to process the images by dividing them into smaller regions. The system successfully detects errors and also reducing the reliance on visual inspection by through MATLAB simulations and FPGA implementation. This system is expected to be a valuable tool in PCB design and manufacturing processes.

1.　背景

電子回路基板の設計・製造工程において，部品の不足が回路の正常な動作を妨げ，製品の不具合を引き起こすことがある．部品が不足している場合，その部分の回路が機能しなくなり，基板全体の性能が低下するだけでなく，基板の過熱等による思わぬ事故が発生する原因となる．特に未経験者が目視で確認する場合，部品の不足に気づかずに作業が進んでしまうことが多い．また，目視確認では微細な部品や配置ミスを見逃すリスクが高く，これが不良品を生み出す要因となる．

このような問題を解決するために，部品の有無を確認するシステムを導入することで，部品の不足を迅速に発見し，未然に不具合を防ぐとともに安全を確保することができる．

そこで，私たちは基板上の部品の位置が正しい配置であるかを確認し，間違い部分を表示するシステムを作成することにした．

2.　Variational Autoencoderについて

VAE(Variational Autoencoder)は，AE(Autoencoder)の拡張版で，ニューラルネットワークを使った生成モデルの一種である．主に教師なし学習で使われ，データ生成能力が強化されている．

VAE の特徴は，AE と同様にデータを圧縮・抽出する仕組みを持ちながら，特徴空間（潜在空間）を確率分布として扱う点である．エンコーダは入力データを「平均」と「分散」のベクトルにマッピングし，そこから潜在変数を確率的にサンプリングする．これにより，潜在空間全体にわたる分布を学習でき，新しいデータの生成が可能となる．

デコーダは，サンプリングされた潜在変数から元のデータを復元する．潜在空間は多変量正規分布のような事前分布に従うよう制約されるため，モデルの汎用性が向上する．また，この仕組みによりノイズ除去や特徴抽出も可能である．

その結果，VAE はデータの復元だけでなく，新しいデータ生成や潜在表現の解析に役立つ．

3.　開発環境

今回使用する Zynq-7　Z7-20 （以下，FPGA)は，VAE を用いて正しい配置の画像と検証画像を比較後，SD カードに結果を保存する．本システムの開発環境を表 1 に示す．

表 1.開発環境

OS	[PC] ・Windows11
開発ソフト	[Zynq-Z7-20] ・MATLAB 2020b ・Vivado 2022.2 ・Vitis HLS 2020.2
評価ボード	・DIGILENT 社製 Zynq-Z7-20
言語	[PC] ・MATLAB [zynq-7020] ・C 言語

4. システム概要

今回作成する回路のレイアウトを確認するシステム(以下,システム)の概要を示す.

4.1 システムの主な目的と有用性

VAE を用いて部品の正しい配置の画像と検証画像を比較し,間違い部分を白色でエリア表示するものとする．基板の判別できる範囲を拡張することで，基板設計や製造工程における部品配置の制度を向上させるとともに部品配置や配置ミスを早期に発見するためのツールとなり，目視確認に依存するリスクを大幅に減らすことができる.

4.2 オリジナルアイデア

画像サイズを10×10として開発を行う予定であった．しかし，FPGA で開発を進める中で，Zynq-7 Z7-10 を使用予定であったが,3×3 よりも画像サイズが大きくなると DSP が 100%を超えてしまった．また，Zynq-7 Z7-20 でも，画像サイズが 5×5 の時点で DSP の使用率が 90%を超え，乗算や加算などのデジタル信号処理を行う部分で必要な容量が不足していることが判明した.

そのため，10×10 の画像を 5×5 の画像の 4 つの領域に分割し処理を行うことにした.

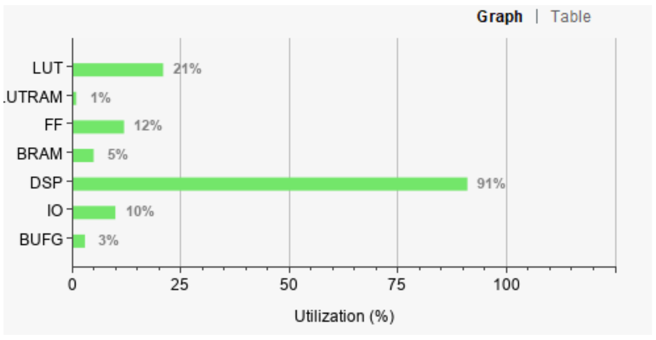


図 1. DSP 使用率

4.3 システム動作の流れ

システム全体の動作の流れについて説明する.

① MATLAB を用いてカメラを起動し，元画像（カラー64x64）と検証画像（10×10）をそれぞれ撮影する．撮影後，検証画像を 4 つの領域に分割して SD カードに保存する.

② VAE を用いて,分割した 5x5 サイズの正しい配置の画像と検証画像を比較する．間違い部分を白色の枠で囲んだ判定画像 4 枚を作成し，SD カードに保存する.

③ 元画像と 4 つの領域に分割した判定画像を再構築して得られる画像を合成する．合成する際，判定画像の白色の部分のみ元画像を白色にする．その後，生成画像を表示する.

システム全体の動作の流れを図 2 に示す.

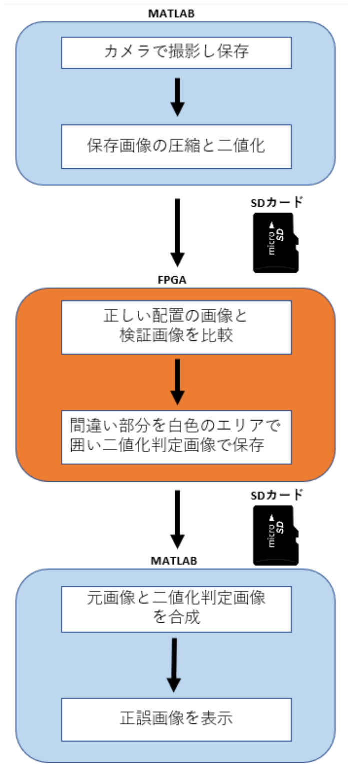

MATLAB
カメラで撮影し保存
保存画像の圧縮と二値化
SDカード
FPGA
正しい配置の画像と検証画像を比較
間違い部分を白色のエリアで囲い二値化判定画像で保存
SDカード
MATLAB
元画像と二値化判定画像を合成
正誤画像を表示

図 2. システムの流れ

5. 実験及び検証

5.1 MATLAB

システムの作成にあたって,PC 上で MATLAB を用いた VAE の画像生成シミュレーションを実施した.

教師データは正しい配置の画像を 4 分割した画像をあらかじめ用意する．教師データを図 3 に示す.

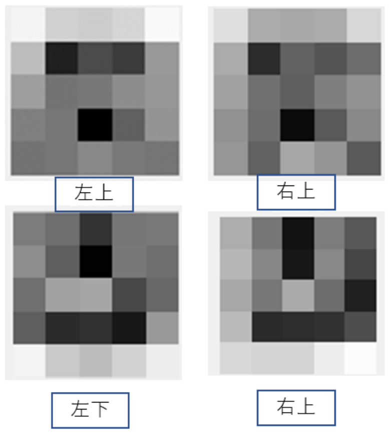


図 3. 教師データ

また, テストデータとして, 正しい配置の画像の他に左側のみ部品の無い検証画像を利用した．テストデータを図 4 に示す．

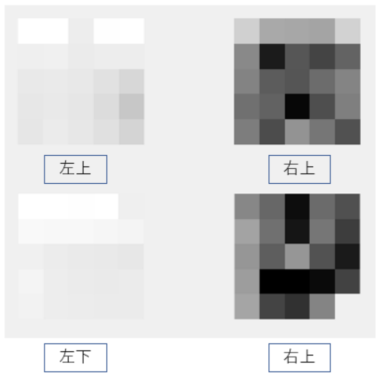


図 4. テストデータ

VAE を行う際，教師データとテストデータの同じ位置の部分を比較した．また，学習率を 0.003，エポック数を 10000 として行った．VAE 後の画像を図 5 に示す．

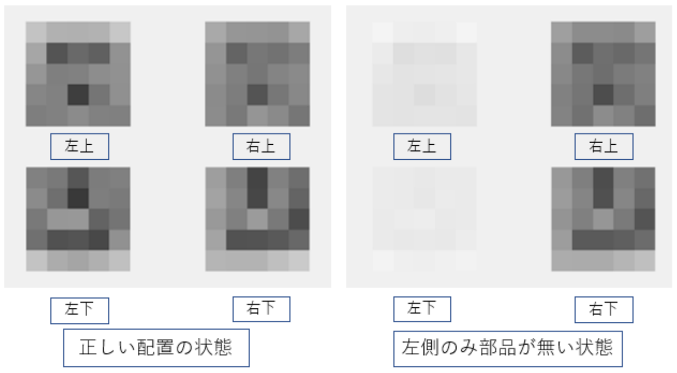


図 5. VAE 後の画像

間違い部分を判別するために VAE 後の画像と教師データの差分から，間違い部分を白色，正しい部分を黒色とし，二値化判定画像を生成した．このとき，教師データの値の±0.2 を閾値としている．これにより，間違い部分と正しい部分が一目でわかるようになる．
二値化判定画像を図 6 に示す．

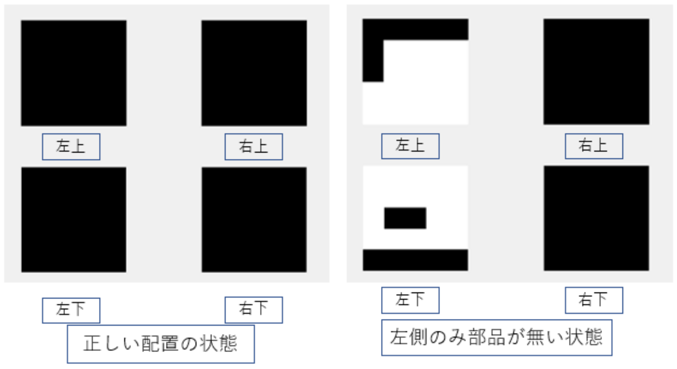


図 6. 二値化判定画像

最後に，元画像と二値化判定画像を合成し間違い部分のみ白色で表示した．これにより，二値化判定画像よりもどの部品が間違っているかをわかりやすくした．正誤判定を図 7 に示す．

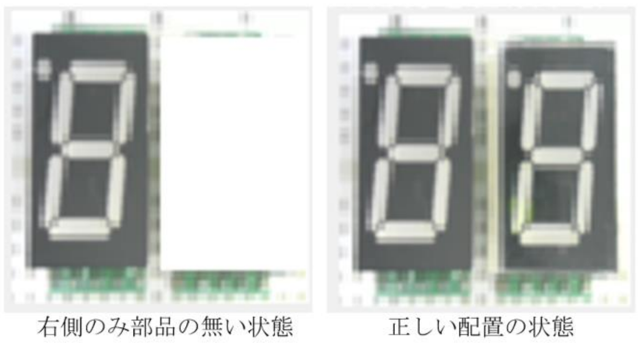


図 7. 正誤画像

5.2 FPGA

MATLAB 上で行った VAE から二値化判定画像出力までの流れを，FPGA 上で実施した．このとき，教師データとテストデータは MATLAB と同じものを使用した．

MATLAB による VAE と同様，教師データとテストデータの同じ位置の部分を比較した．このとき，学習率を 0.003，エポック数を 10000 として行った．VAE 後の画像を図 8 に示す．

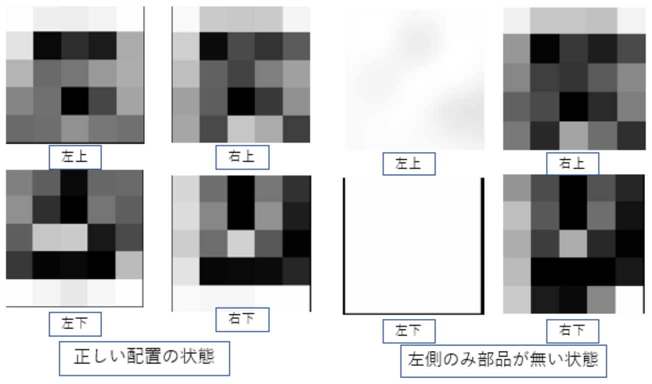


図 8. VAE 後の画像

間違い部分を判別するために VAE 後の画像と教師データの差分から，間違い部分を白色，正しい部分を黒色とし，二値化判定画像を生成した．このとき，教師データの値の±0.2 を閾値としている．
二値化判定画像を図 9 に示す．

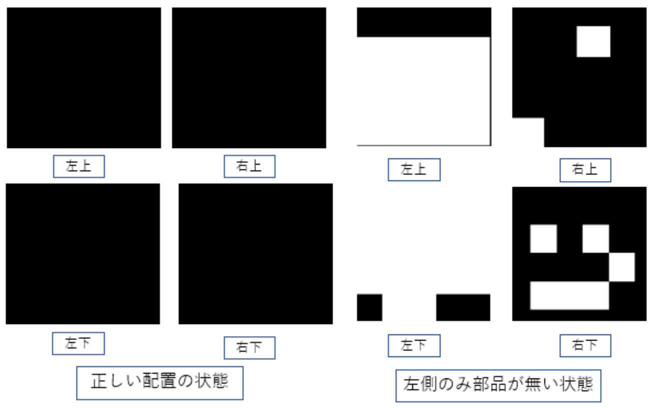


図 9. 二値化判定画像(閾値±0. 2)

しかし，閾値±0. 2 だと間違い部分であるにも関わらず一部の部分が正しいと判別していたため，閾値を±0. 4 に変更し，再度二値化判定画像を生成した．二値化判定画像を図 10 に示す．

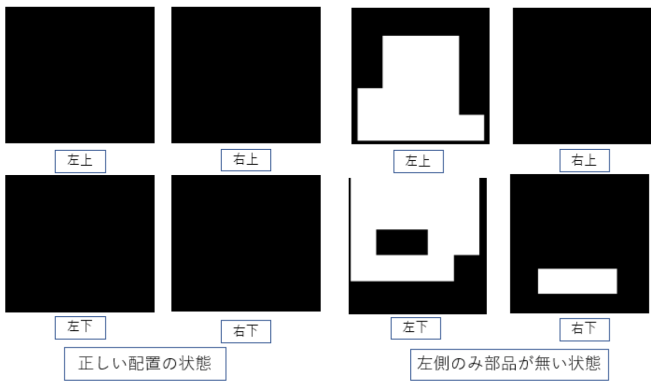


図 10. 二値化判定画像(閾値±0. 4)

図 10 では図 9 よりも正確に判別できていたため，実際に元画像と合成し間違い部分のみ白色で表示した．正誤判定を図 11 に示す．

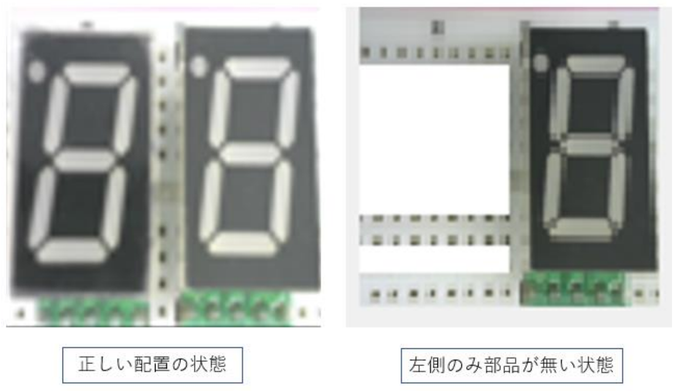


図 11. 正誤画像

5.3 MATLAB と FPGA の結果比較

今回，両方で 5×5 の 4 つの領域に分割し VAE を行ったが，MATLAB ではソフトウェアのみでの実装，FPGA では，ソフトウェアとハードウェアの両方を行っている．MATLAB では大体の位置の間違いを判定しており，FPGA では細かな部分の間違いを判定できていたため，VAE の精度に差が出ていることが分かった．圧縮時の値が変化し，復元する際にその差が大きくなったため，VAE 後の値が大きく変化し，二値化判定において差が出たのではないかと考える．

6. 評価

画像の 4 分割，二値化，VAE での画像生成，二値化判定，画像を合成し間違い部分をマークする 5 個の項目は，MATLAB と FPGA により実装できており，正誤判定を行うことが可能なため〇とする．しかし，検証画像の撮影と保存の項目については，撮影位置が固定であり，部屋の明るさの変動や，別の位置で撮影を行うと正常に判定ができない恐れがあるため△とする．各項目の評価を表 2 に示す．

表 2. 評価

項目	評価
検証画像の撮影と保存	△
画像の 4 分割	○
二値化	○
VAE での画像生成（MATLAB, FPGA）	○
二値化判定	○
画像を合成し間違いの部分をマーク	○

7. スケジュール

1 月中に要求仕様書と基本設計書を作成し，2 月中に開発を行い，3 月中に発表準備を行う予定であったが，VAE の理解に時間を要してしまった．また，実際に画像を数値に変換して VAE を実行した際，データ量が不足しオーバーフローが発生したため，値が想定通りに表示できなかった．このため，シミュレーションと実装に遅れが生じた．

遅れを挽回するために，画像を分割し VAE を複数回実行することで改善を図りつつ，資料作成などの予定を変更し，FPGA の作業を皆で分担して行った．その結果，FPGA に実装しシステムの流れ通りに正誤画像を出力させることができ，開発スケジュールを取り戻すことができた．開発スケジュールは図 12 に示す．

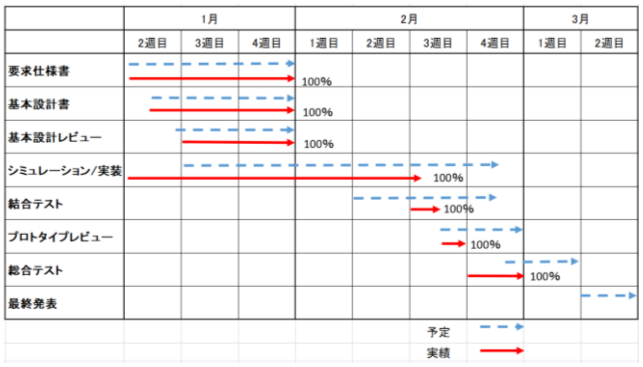


図 12. 開発スケジュール

8. まとめ

MATLAB と FPGA で 5×5 の 4 つの領域に対する VAE の実装に成功した．シミュレーションでは判別がうまくいっていたものの，実際に機器に組み込む際には，値が大きく変化するとともに実装できる容量についても検討しなければならず，その部分において多くの試行錯誤を重ねる必要があり，とても大変であった．

しかし，最終的に FPGA に組み込むことができたため，今後は判別可能な領域を増やし，より大きな基板の判別ができるようにシステムを拡張していきたいと考えている．

9. 参考資料

「VAE は異常検知に役立つ？初心者向けに概要やメリットを分かりやすく解説」
https://www.tryeting.jp/column/2382/

「オートエンコーダー」
https://jp.mathworks.com/discovery/autoencoder.html

Implementation of a VAE-Based Circuit for Image Compression and Anomaly Detection and Its Potential Use in Edge Computing

1st Yuki Imamura
Kyushu Institute of Technology
School of Computer Science and Systems Engineering
Department of Computer Science and Networks
Fukuoka, Japan
imamura.yuuki475@mail.kyutech.jp

2nd Taiga Kawasaki
Kyushu Institute of Technology
School of Computer Science and Systems Engineering
Department of Computer Science and Networks
Fukuoka, Japan
kawasaki.taiga711@mail.kyutech.jp

***Abstract*—With the recent evolution of wireless communication technology and the spread of 5G, there is a need to create a new communication environment that takes advantage of the characteristics of low latency and multiple simultaneous connections. On the other hand, there is a potential problem due to the increased burden of cloud processing. Edge computing is attracting attention as a method to solve this problem. In this study, we developed a system that compresses and analyzes acquired images by implementing VAE using FPGA to realize efficient data processing on an edge server. As a result, the system was able to perform image compression and anomaly detection. However, the system's performance is expected to be further improved by utilizing higher-accuracy FPGAs and by incorporating color image processing.**

***Index Terms*—VAE, FPGA, Edge Computing, Image Compression, Anomaly Detection**

I. Introduction

In recent years, wireless communication technology has improved dramatically, and 5G communications are becoming increasingly popular. Unlike conventional communication standards such as 4G, 5G has the three characteristics of "high speed and large capacity," "low latency," and "multiple simultaneous connections," of which "low latency" and "multiple simultaneous connections" are important axes for building a new communication environment. Conventional 4G communications have focused on smartphones and cell phones used by people. However, 5G assumes that IoT devices such as vehicles, drones, and sensors will be connected to the network in large numbers.

In addition, cloud computing is the mainstream in the IT industry today [4]. As shown in the left side of Fig.1, the cloud performs the processing from the edge device and notifies the edge device of the result. However, there is a limit to the amount of data acquired from many devices that can be processed only by the cloud, and the amount of traffic will increase, making it difficult to enjoy the benefits of 5G.

To solve these problems, edge computing has been gaining attention in recent years. Edge computing is a technology that performs part of the processing that is conventionally performed in the cloud, such as data processing at a base station or server located near the user terminal (smartphone or IoT device) [2] [3]. Using this technique, it is possible to distribute the processing that is required in the cloud to edge computing, reduce the amount of communication traffic, and take advantage of the "low latency" that is one of the features of 5G.

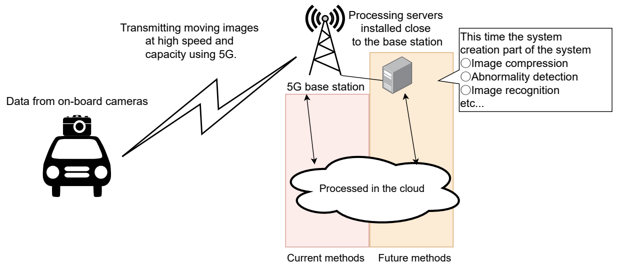

Transmitting moving images at high speed and capacity using 5G.
Processing servers installed close to the base station
This time the system creation part of the system
◯Image compression
◯Abnormality detection
◯Image recognition
etc...
5G base station
Data from on-board cameras
Processed in the cloud
Current methods
Future methods

Fig. 1: Image of edge computing and the part of the system created this time

Therefore, we considered the use of VAE and FPGAs to realize edge computing. A system for detecting foreign objects on the road has already been created using VAE, and certain results have been obtained [5]. Therefore, in this project, we aimed to create a part of the process on the right side of Fig.1 by processing images acquired from an onboard camera using an FPGA equipped with a VAE.

II. Method

The tools and their versions used in the creation of this system are shown in Table I. ZYNQ-7010 with DIGILENT SoC is used as the FPGA evaluation board.

A. System Structure

This VAE-equipped system has the following two functions.

1 image compression
The dimensional compression capability, one of the features of VAE, is applied to images.

TABLE I: Tools used in this development

Usage	tools used
For VAE simulation	MATLAB 2024b
Hardware simulation	MATLAB/Simulink 2022a
HDL Code Generation	HDL Coder
FPGA Design Software	Vivado 2022.1
Hardware acceleration	Vitis 2022.1
Evaluation Board	DIGILENT ZYNQ-7010

2 Anomaly detection
Another feature of VAE, anomaly detection is performed by comparing the original image and the generated image.

This section describes the overall concept. The images used in this project are grayscaled for simplicity. The image is divided into blocks like a JPEG, and VAE is used for each block. The block size is set to 16 × 16. PSNR is used to compare the original image with the compressed image for each block. Image compression and anomaly detection are processed using PSNR. For image compression, if the PSNR is above a set threshold, the compressed latent space is used as the compressed data, since there is no problem in using the compressed latent space. For anomaly detection, if the PSNR value is less than the threshold value, the system is set to judge the image as an anomaly because there is a high possibility that it is not a road.

In order to realize the above functions, the structure of VAE is described inII-A1. The details of the FPGA structure are explained in II-A2, and finally the SoC FPGA structure is explained in II-A3.

1) VAE Structure: A schematic of the VAE structure is shown in Fig.2. Since 16 × 16 images are used, 256 input dimensions and 256 output dimensions were used in the design. The latent space was designed to have 16 dimensions to allow for a certain degree of discrimination even after compression. The encoder part uses a ReLU function for the mean and a soft plus function for the variance. The decoder part uses a sigmoid function.

$$f(x) = x \qquad : ReLU function \tag{1}$$

$$f(x) = \log(1 + e^x) \qquad : Softplus function \tag{2}$$

$$f(x) = \frac{1}{1 + e^{-x}} \qquad : Sigmoid function \tag{3}$$

The VAE learning method is shown in Fig.3. First, a large number of road-only blocks are prepared. The blocks are used as teacher data to train VAE. The learning conditions are summarized in table II. In this case, we prepared a photo showing a road (the upper left image in Fig.3) and prepared the teacher data by converting the part showing only the road (the upper right image in Fig.3) into a block. The VAE was trained in MATLAB, and the weights and parameters output from the training were used to control the VAE mounted on FPGA.

2) FPGA Structure: The board used was ZYNQ-7010 manufactured by DIGILENT. In this design, $X(16)$, $W(16 \times 2)$, and $b(16)$ were prepared as input data and $Z(16)$ as output

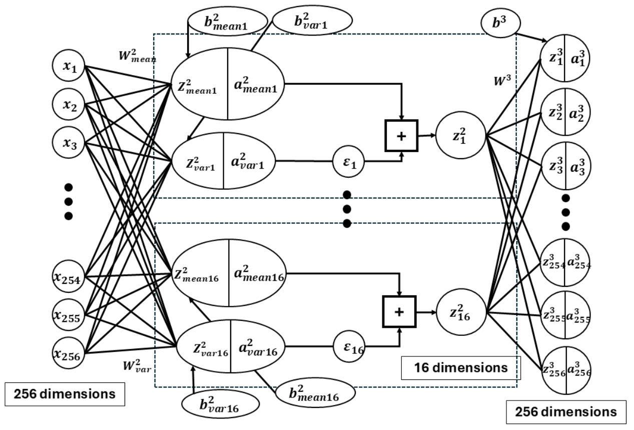


Fig. 2: Structure of 256 × 16 × 256VAE

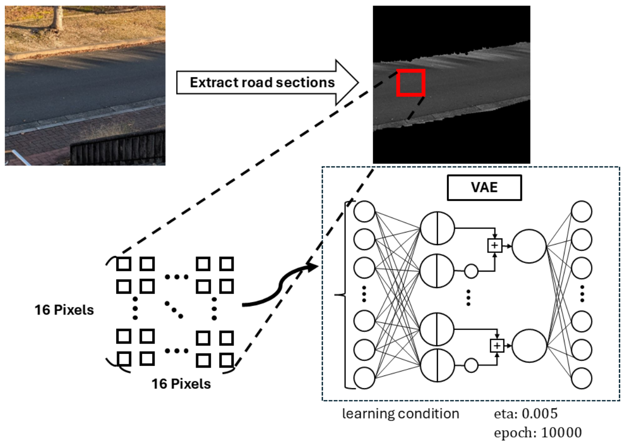


Fig. 3: How to study VAE

as shown in Fig.4 (the number of dimensions in parentheses). The unit enclosed in the red box, where $Output_1$ is the output, performs the operation of multiplying the input and weight parameters, as shown in Equation 4.

$$Output_1 = X_1 \times W1_1 \tag{4}$$

Sixteen such units are shown in the blue box. The final Z output is the computation of Equation 5.

$$Z_1 = \sum_{i=0}^{16} X_i \times W1_i + b1 \tag{5}$$

Initially, we wanted to put 256 input and 256 output operations on the FPGA, but there was a capacity limitation. Therefore, we designed the VAE of 256 × 16 × 256 to have 16 inputs for X to make it easier to realize the VAE of 256 × 16 × 256 that we designed this time.

TABLE II: VAE learning conditions

epoch	10000
eta	0.0005
Layer2(number of latent spaces)	16

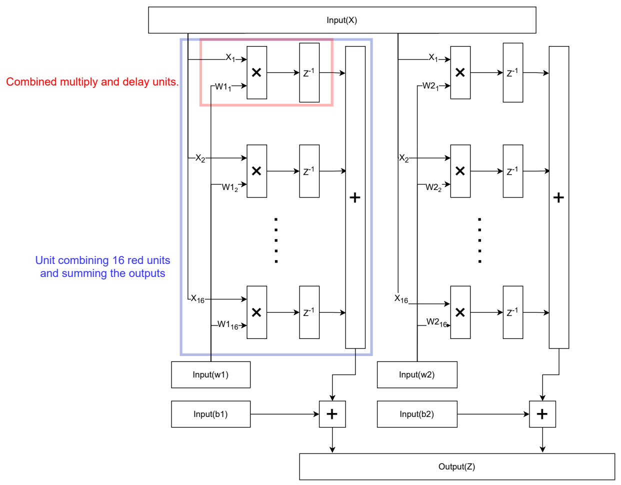

Input(X)
Combined multiply and delay units.
Unit combining 16 red units and summing the outputs
Input(w1)
Input(b1)
Input(w2)
Input(b2)
Output(Z)

Fig. 4: FPGA structure

TABLE III: FPGA Resource Utilization

Resorce	Utilization	Available	Utilization[%]
LUT	3301	17600	18.76
LUTRAM	62	6000	1.03
FF	3297	35200	9.37
DSP	64	80	80.0
IO	12	100	12.0
BUFG	1	32	3.13

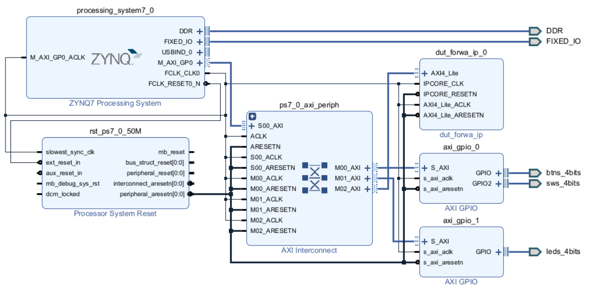


Fig. 5: SoC FPGA Configuration Diagram

3) SoC FPGA Structure: Fig.5 shows an overview of the SoC FPGA system structure. The processing **processing_system7_0** performs various processes through the bus, and plays the role of CPU. The **dut_forwa_ip_0** part is the part of the FPGA created this time. The resource utilization evaluation after implementation is shown in Table III.

Next, an overview of the SoC FPGA processing is shown in Fig.6. The SD card contains weights and parameters stored in CSV files and image data in RAW format. The CPU reads these data. The CPU then processes the input data according to the created FPGA and sends the data to the FPGA. The FPGA executes the data as soon as it is stored and stores the output results. The CPU reads the output result and processes the output data. These processes are repeated.

This VAE is $256 \times 16 \times 256$, and the FPGA structure created in II-A2 is 16×2. The part of the FPGA that can be processed by running the designed FPGA once is shown in Fig.7. The encoder assigns the computation of the two outputs to z^2_{mean} and z^2_{var}. Since the FPGA can process 16 out of 256 input dimensions in a single use, it is necessary to use the FPGA 16 times to compute a single latent space. The output z^2_{mean} and z^2_{var} are designed to be stored in b. Since there are 16 latent spaces, the encoder calculation runs a total of $16 \times 16 = 256$ times. When computing the decoder part, only one FPGA run is needed to obtain 2 out of 256 dimensions of the output of the third layer Z^3_i. Therefore, the FPGA is run 128 times in the decoder. This kind of control was created using a CPU.

The z^2_{mean}, z^2_{var}, and z^3 obtained from the output are used to calculate a^2_{mean}, a^2_{var}, and a^3 using the active functions in the software. Furthermore, z^2 is also calculated on the software using a^2_{mean} and a^2_{var}. Finally, PSNR is measured by comparing the input and output a^3 values.

B. Experimental Procedure

In this case, we prepared three types of images shown in Fig.8. Fig.8a is an image without any falling objects. Fig.8b and Fig.8c are images with falling objects, and were prepared for comparison of anomaly detection. The size of the image is 512×512, and the image is grayed out when passed through VAE. The image is processed by dividing it into $16^times16$ blocks, and the PSNR is measured. The PSNR is compared between the MATLAB output and the FPGA output. The PSNR output from the FPGA is written to a CSV file, and the data is presented in MATLAB.

III. Experiments and Discussions

A. Experimental Results

First, the SoC FPGA execution screen is shown in Fig.9. The CSV file stored in the SD card is read by pressing button 0. The image is passed through the VAE by pressing buttons 1 to 3. Fig.9 shows the state after button 0 is pressed and the parameters are acquired.

1) Comparison of output of image 1 : Fig.10b shows the output in MATLAB and Fig.10c shows the output in FPGA.

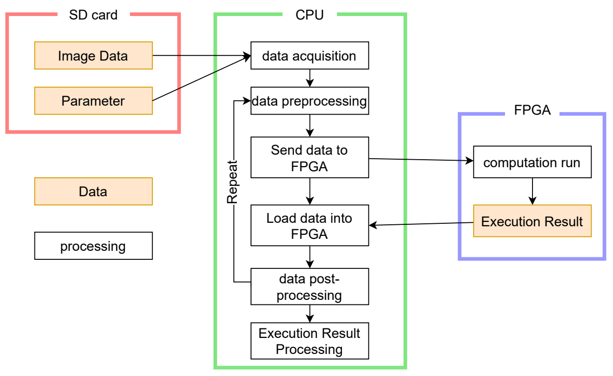


Fig. 6: SoC FPGA Processing Overview

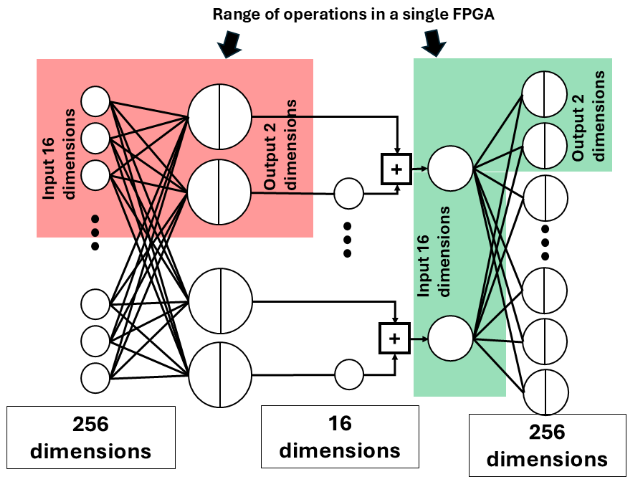
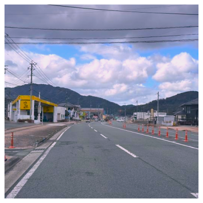
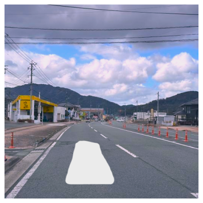


Fig. 7: Image of FPGA use


(a) Image 1 Original image


(b) Image 2 With falling objects

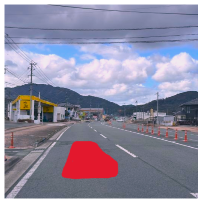

(c) Image 3 With falling objects

Fig. 8: Evaluation image to be tested this time

In MATLAB, the PSNR around the road area shows a value of 25 or higher. In addition, parts of the sky and mountains also have a high PSNR. The FPGA output shows a lower overall PSNR compared to MATLAB. However, the PSNR of the road section is around 25, which is not a bad result.

2) Comparison of output of image 2 : Fig.11b shows the output from MATLAB and Fig.11c shows the output from FPGA. Both MATLAB and FPGA outputs show a worse PSNR in the area of the falling objects.

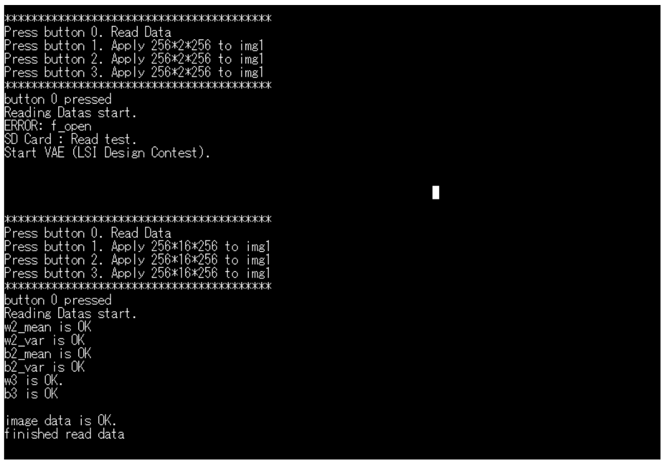
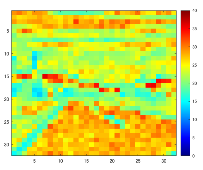

```
****************************************
Press button 0. Read Data
Press button 1. Apply 256*2*256 to img1
Press button 2. Apply 256*2*256 to img1
Press button 3. Apply 256*2*256 to img1
****************************************
button 0 pressed
Reading Datas start.
ERROR: f_open
SD Card : Read test.
Start VAE (LSI Design Contest).




****************************************
Press button 0. Read Data
Press button 1. Apply 256*16*256 to img1
Press button 2. Apply 256*16*256 to img1
Press button 3. Apply 256*16*256 to img1
****************************************
button 0 pressed
Reading Datas start.
w2_mean is OK
w2_var is OK
b2_mean is OK
b2_var is OK
w3 is OK.
b3 is OK

image data is OK.
finished read data
```

Fig. 9: SoC FPGA execution screen

(a) Image 1 to be tested


(b) PSNR output block by block in MATLAB

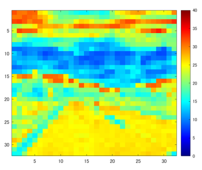
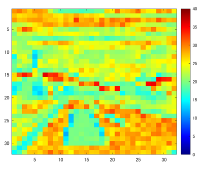

(c) PSNR output block by block in FPGA

Fig. 10: Processing results for image 1

(a) Image 2 to be tested


(b) PSNR output block by block in MATLAB

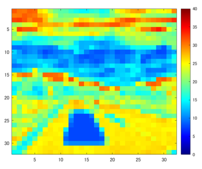

(c) PSNR output block by block in FPGA

Fig. 11: Processing results for image 2

3) Comparison of output of image 3 : Fig.12b shows the output in MATLAB and Fig.12c shows the output in FPGA. Unlike the output in image 2, the PSNR in the area where the falling object is located is conversely improved.

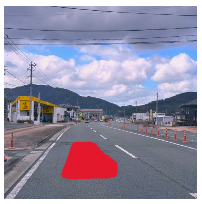
4) Summary of results: Comparing the output images, there were differences between the MATLAB and FPGA output results. However, since the PSNR was higher for the road section in FPGA than for other sections, we do not think that there was a failure.

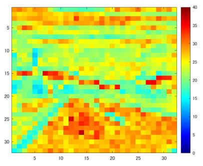
In addition, it was found that the anomaly detection was affected by the color of the falling object. The PSNR worsened when the object was white, while the PSNR improved when the object was red.

B. Consideration

From the results obtained, the following three points are discussed.

1) The point that PSNR is high except for roads
2) MATLAB and FPGA outputs are different


(a) Image 2 to be tested


(b) PSNR output block by block in MATLAB

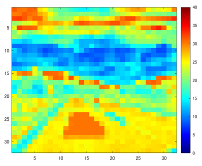

(c) PSNR output block by block in FPGA

Fig. 12: Processing results for image 3

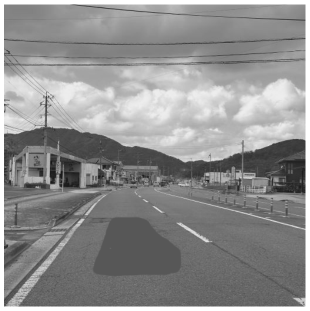


Fig. 13: Image 3 converted to grayscale

3) PSNR is improved even for red objects

(1) can be considered from two points of view: the effect of grayscaling and the effect of VAE characteristics. In particular, when the sky portion is grayscaled, it has fewer features than the road portion. Therefore, the VAE created in this study can represent them, and it is thought that the image restoration capability of the VAE can be demonstrated.

(2) is considered to be caused by the variables used and the fixed-point error. However, since the detailed output has not been confirmed, we will conduct further verification to determine the cause.

(3) is thought to be caused by grayscaling. When the red color was converted to grayscale, it was almost the same as the color of the road (Fig.13). This suggests that the color scale should be used to correctly identify falling objects.

1) Evaluating adoption in edge computing: We will evaluate whether it can be adopted as edge computing.

First, we evaluate image compression. This time, the PSNR threshold is set at 25, and judgments are made on the basis that images above the threshold are compressible and those below the threshold are uncompressible. The number of blocks for which the threshold is 25 or higher, the capacity before compression, the capacity after compression, and the compression ratio are shown in Table IV. The size calculation is shown in the formula (6). Since the original image takes values from 0 to 255, each pixel is 8 bits. Since the latent space is compressed data, 8-bit fixed-point numbers are used. The B denotes the number of blocks above the threshold.

$$size = B \cdot 16 \cdot 8[bit] + (32 \cdot 32 - B) \cdot 16 \cdot 8[bit] \qquad (6)$$

The compression ratio for all the images is close to 70%. By sending compressed images to the cloud for processing, it is possible to reduce the amount of traffic from the base station to the processing server and to improve the processing speed in the cloud.

Next, anomaly detection is evaluated. As shown in the experimental results, whether an abnormality can be detected depends on the color of the falling object. However, since abnormality detection is possible when the object is white, we believe that it is possible if the processed image is a color image. If the recognition is performed on the area where the vehicle is going to run, it is possible to avoid accidents by immediately notifying the vehicle of the recognition results.

TABLE IV: image compression effect

No.	Blocks	Before[bit]	After[bit]	Com.ratio[%]
1	346	2097152	1432832	68.32
2	305	2097152	1511552	72.08
3	349	2097152	1427072	68.05

Finally, we evaluate real-time performance, which is important in edge computing. The system took about 4 to 5 seconds to input an image and output the PSNR. If the system were to continue at this rate, an accident could occur before an abnormality judgment could be made. The reason for the long processing time is thought to be that data was written and read many times using a time-consuming bus. In this case, the circuit that could be mounted on the FPGA is a part of VAE, and the FPGA is executed many times, and data is transferred using the bus each time. In fact, to process a 512×512 image in VAE, the FPGA is run 393216 times. However, it is highly possible that this problem can be solved by using a higher-precision FPGA.

IV. Conclusions and Future Prospects

In this study, we implemented VAE on FPGA and constructed a system that compresses road images and detects anomalies. The results show that image processing in edge computing using FPGAs is feasible.

Although the system has some problems, we believe that they can be solved with appropriate improvements. Therefore, we believe that the system has potential for future industrial applications.

This contest allowed us to experience LSI development using FPGAs. I felt the future potential of FPGAs and recognized that FPGAs are necessary devices for efficient computation in today's world where AI is used on a daily basis. I would like to continue to develop FPGAs that can demonstrate their capabilities, and since technologies for developing FPGAs, such as high-level synthesis, are evolving day by day, I would like to challenge myself in various ways.

Acknowledgment

Thank you to the LSI Design Contest Executive Committee for organizing this contest, and to the organizers, co-sponsors, sponsors, and supporters for their support. I would also like to thank all the professors who provided various kinds of support even though they were not assigned to my laboratory.

References

[1] H. Morikawa, *5G Jisedai Ido Tsushin Kikaku no Kanosei*, Iwanami Shoten, Tokyo, 2020.

[2] Y. Tanaka, N. Takahashi, and R. Kawamura, "IoT Jidai o Hiraku Edge Computing no Kenkyu Kaihatsu," *NTT Giho Journal*, vol. 27, no. 8, pp. 59-63, 2015.

[3] H. Yokota, S. Oda, T. Kobayashi, D. Ishii, T. Ito, and A. Isozumi, "IoT no Missing Link o Tsunagu Edge Computing Gijutsu," *NEC Giho*, vol. 70, no. 1, 2017.

[4] Ministry of Internal Affairs and Communications, "Reiwa 6-nenban Joho Tsushin Hakusho," 2024.

[5] T. Yamamoto, A. Hashimoto, and H. Okamoto, "Heikin Gazou ni Taisuru VAE Ijou Kenshi no Tekiyo ni Yoru Doro Rakka-mono Kenshutsu," in *Proc. Annu. Conf. Jpn. Soc. Artif. Intell.*, vol. 35, 2021.

[6] ”LSI Design Contest,” Available: http://www.lsi-contest.com/. Accessed: 2025-01-31.

焙煎状態の自動品質評価システム
―VAE を用いた食品加熱プロセスの最適制御―

1st Sizumasa Kojo
Kyusyu Politechnic College
Department of Electronics and Information Tecnology
Fukuoka, Japan
2317114@kyushu-pc.ac.jp

2nd Masaya Furuta
Kyusyu Politechnic College
Department of Electronics and Information Tecnology
Fukuoka, Japan
2317132@kyushu-pc.ac.jp

I. はじめに

パンの焦げ目を自動検知するシステムは、食品製造における品質管理の効率化と高精度化を目的とした重要な技術である。従来の目視による検査方法は簡便であるものの、検査者の熟練度に依存し、判断のばらつきが生じる問題が発生した。また、大量生産の現場においては、一貫性のある品質管理を人力で行うことが困難であると指摘されている [1]。この課題に対応するため、本研究では画像処理技術と機械学習アルゴリズムを融合したシステムを開発した。

本システムは、焼き加減を判別するために、パン表面の色相や明度などの特徴量を抽出し、それを基に適切な焦げ目の状態をリアルタイムで判定する。動作の流れを図 1 に示す。

本研究の成果は、食品製造における品質保証の新たな標準を提供するものであり、特に大規模な製造ラインにおける効率的な品質管理の実現に寄与すると考えている。加えて、本システムの導入により、作業者の負担軽減や食品ロスの削減にもつながることが期待される。

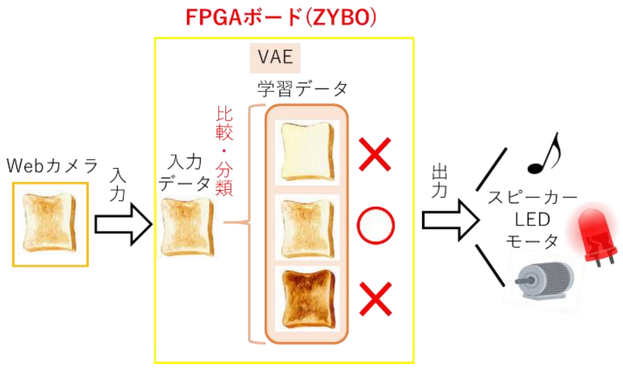


図 1. 動作の流れ

II. 概要

A. 装置概要

図 2(a)に示した装置は、現段階ではプロトタイプの概念実証を目的としたものである。設計プロセスにおいては、図 2(b)に示すように 3 次元 CAD ソフトウェア Tinkercad を使用し、精密な寸法計測と構造解析を実施した。

本装置は、オーブン型のデザインを採用しており、外形には黒と透明のアクリル板を使用した。また、本装置にはスイッチ、LED、モータ、スピーカー、カメラ、FPGA ボード、自作基板が搭載されている。自作基板はスイッチ、LED、モータ、スピーカーを制御するためのものであり、その回路図を図 3、4 に示す。

スイッチを押すことによりパンの焦げ目の判別を開始し、適切な焦げ目になると自動で扉が開く仕組みとなっている。また、LED でも視覚的に状況を把握することが可能である。

本装置には加熱機能がなく、パンの焦げ目を生み出すことはできない。そのため事前に焦げ目の異なるパンを用意し、USB カメラから読み取って焦げ目の判別をしている。今後は、パンを加熱して焦げ目を生み出す環境下での運用が必要不可欠である。

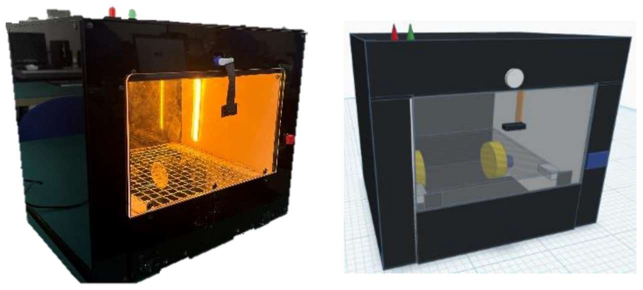

(a)製作した装置　(b) CAD 設計図
図 2. 装置の外観

図 3 に示すのは、スピーカー、DC モータ、および RC サーボモータの接続回路図である。各デバイスの特性に応じた電源供給と信号制御が設計されており、効率的な動作と安全性を確保している。

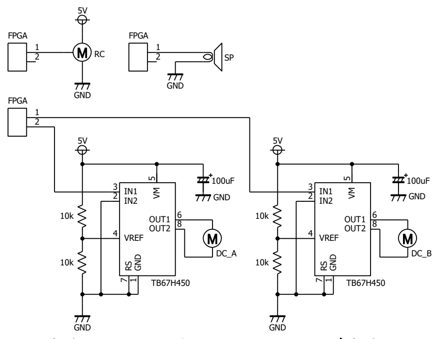


図 3. アクチュエータ回路図

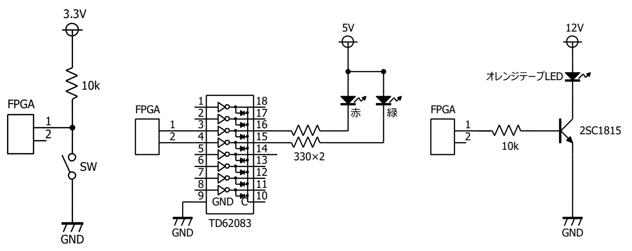


図 4. 入力・出力制御回路

B. *システム概要*

本システムの開発環境を以下に示す。

- ZYBO Zynq-7020
- Vivado 2022.2
- Xilinx Vitis 2022.2
- MATLAB R2022b

本システムの焦げ目認識と自動出力制御のアルゴリズムでは、Web カメラから取得した画像を入力として VAE (Variational Autoencoder) に供給し、あらかじめ学習させた焦げ目の教師データと比較・分析をし、誤差を取得することで焦げ目の識別を行う。その後、VAE の出力値に基づき LED、スピーカー、各種モータの制御信号を自作基板に送信する(図 5 参照)。

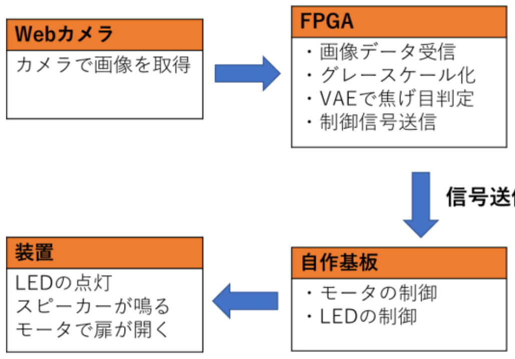
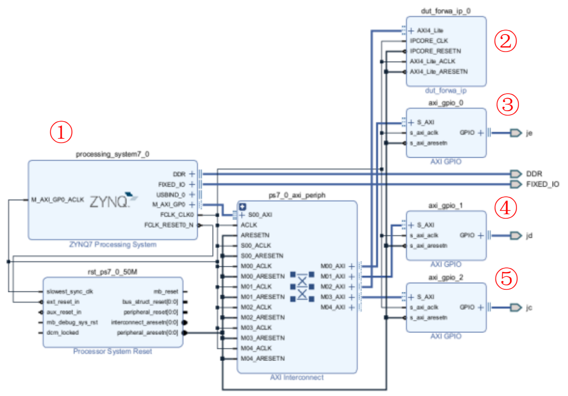


図 5. 処理の流れ

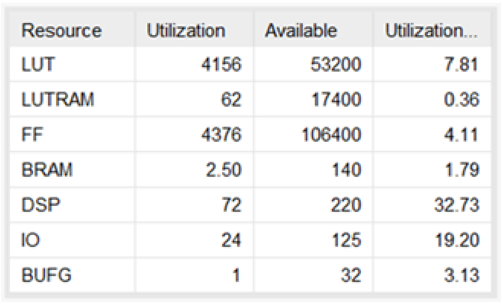
図 6 に本システムの FPGA 内部ブロック図を示す。主要な IP ブロックの機能は以下の通りである。また、表 1 にハードウェアサイズを示す。



① CPU のブロック
② VAE を行うブロック
③ SW (入力部) のブロック
④ LED、テープ LED 制御のブロック
⑤ DC モータ、RC モータ、スピーカー制御のブロック

図 6. FPGA のブロック図

表 1. ハードウェアサイズ



Resource	Utilization	Available	Utilization...
LUT	4156	53200	7.81
LUTRAM	62	17400	0.36
FF	4376	106400	4.11
BRAM	2.50	140	1.79
DSP	72	220	32.73
IO	24	125	19.20
BUFG	1	32	3.13

C. *画像前処理の手法*

食パンを同じトースターで加熱すると焦げ目は同じところに出るが、食パンを置く位置によって、カメラから読み取るときに焦げた位置がずれて処理に問題が発生した。そのため、パンを置く位置を固定することとした(図 7 参照)。

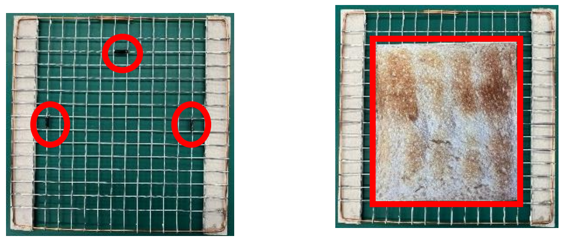


図 7. パンの配置指定

本研究では、VAE に前処理された画像データを供給する。パンの焦げ目は中央に現れやすく、カメラとの距離の影響も考慮する必要があるため、パン全体の画像ではなく、中央部分のみを切り取った画像を使用することとした(図 8 参照)。

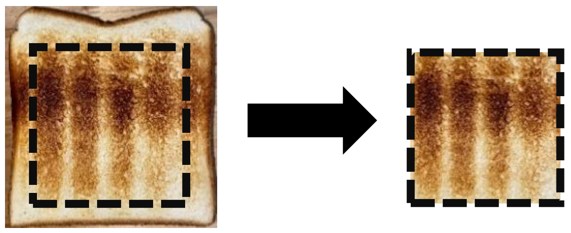


図 8. パンの中央画像抽出

また、FPGA(ZYBO)で処理をするため画像ノイズを考慮する必要が発生し、以下の画像処理技術を実装して VAE モデルの入力としての最適化を図った。

① 解像度正規化
入力画像を 64×64 ピクセルの[0, 1]範囲に正規化することで、計算負荷を最適化し、リアルタイム処理への対応を可能とした。正規化でネットワークの学習効率が向上した。

② グレースケール変換
カラー画像をグレースケールに変換することで、モデルの計算効率を向上させ、特徴抽出の精度を高めることを実現した。

III. 実験

A. *実験内容*

本実験では、MATLAB を用いてシミュレーションを行った。そして、本実験での VAE モデルを図 9 に示す。本実験では、入力層、出力層を 64×64(4096)で中間層は 128 の構成で行うことにした。また、パンの画像は、同じトースターを使用して加熱したものを撮影し、加熱条件を統一している。

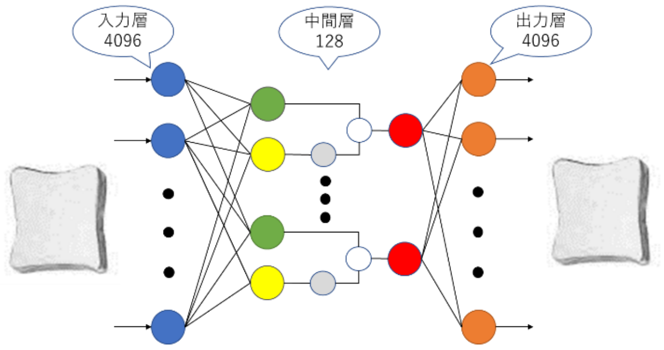


図 9. VAE モデル

VAE を用いて以下の実験を行った。実験 1 では、教師データ自体を入力した場合に、元の教師データが正確に再現・出力されるかを確認する。次に、実験 2 では教師データにないデータを入力した場合に、教師データに近い画像を出力し、差分を取れるか確認する。

実験 1 で使用した教師データを図 10 に示す。そして、実験 2 でも図 10 の教師データを使用し、入力データには 6 種類の焦げ目の違う画像を使用した。その画像を図 11 に示す。1 枚目と 6 枚目は教師データとは別につくった適切な焦げ目、2~5 枚目は白いパンから適切な焦げ目になる直前までの過程の画像である。

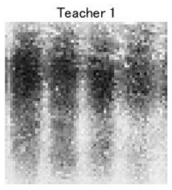


図 10. 実験で使用した教師データ

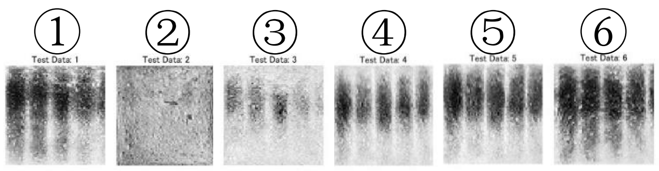


図 11. 実験 2 で使用した入力データ

B. *実験結果*

実験 1 の結果を図 12 に示す。教師データと出力データを比較すると、出力データにはノイズのようなものが含まれており、教師データを正確に再現できていないことが確認された。このとき、学習率は 0.0001、学習回数は 2000 で行った。このときの学習回数とエラー数の関係を示したグラフを図 13 に示す。このグラフから、学習回数 2000 回のときではエラー数が多く VAE が十分に収束できていないと考えられる。そのため学習回数 10000 回に増やし、再度学習を行った。

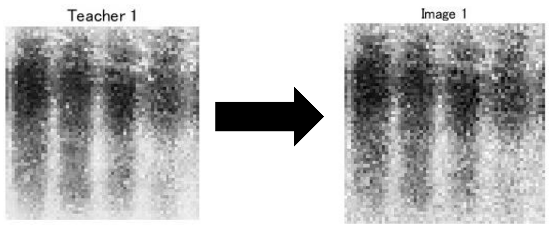


図 12. 入力データと出力データ (1)

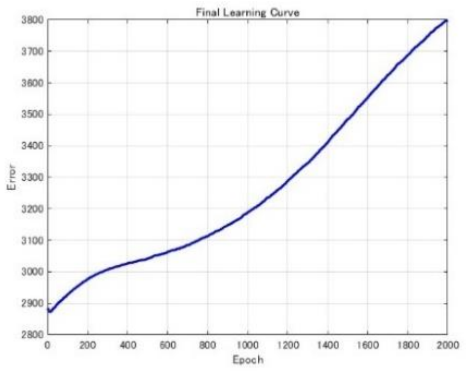


図 13. エラー数と学習回数 (1)

学習回数を増やしたときの実験結果を図 14 に示す。教師データと出力データを比較したところ、正しく出力されていることが確認できた。また、このときの学習回数とエラー数の関係を示したグラフ(図 15)からエラー数が安定していることがわかる。

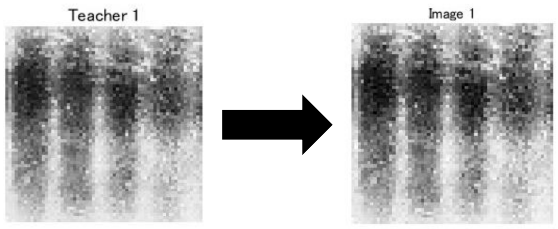


図 14. 入力データと出力データ (2)

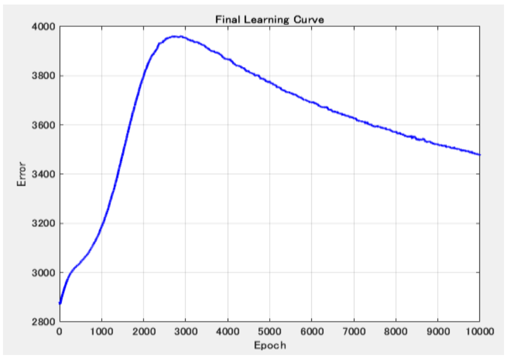


図 15. エラー数と学習回数 (2)

実験 1 では、学習率を 0.0001、学習回数を 10000 に設定することで、教師データを入力した場合に元の教師データを正確に再現できることが確認された。

次に、実験 2 では、実験 1 の結果から得られた最適な学習率（0.0001）と学習回数(10000)を使用した。実験 2 の結果を図 16 に示す。どの入力データに対しても、教師データに近い画像を出力することが確認できた。この結果を基に、1 枚目と 6 枚目の画像を適切な焦げ目のパン、2～5 枚目の画像をまだ焼けてないパンと判定するために、入力データと出力データの差分を算出した。

差分を算出した結果を図 17 に示す。図 17 から 1 枚目と 6 枚目の差分は小さくなっており、2～5 枚目の差分は大きいことが確認できた。これから差分が、500 以下の場合に「適切な焦げ目」と判定する基準を設定した。

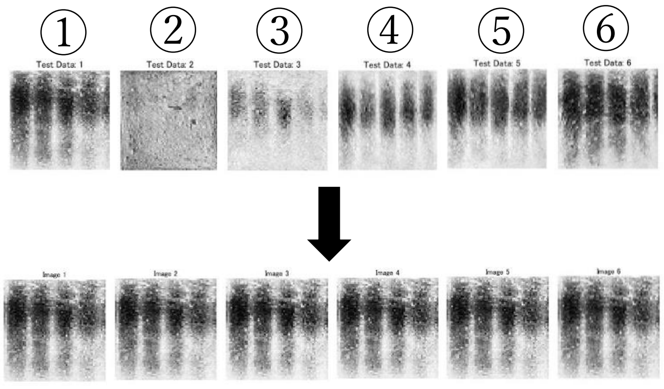


図 16. 入力データと出力データ(3)

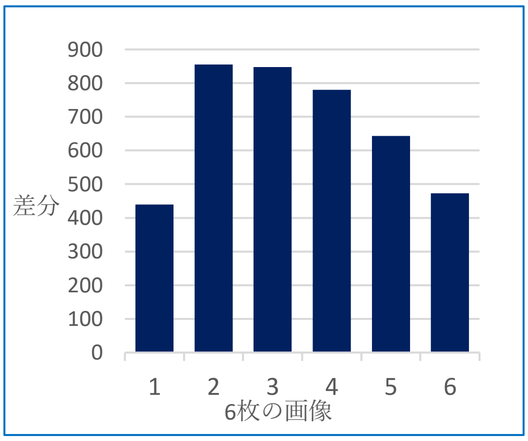


図 17. データの差分

以上の結果から、学習率を 0.0001、学習回数を 10000 回、差分の合計値を 500 以下にすることで目的とする出力を再現できるようになった。

IV. おわりに

本研究では、パンの焦げ目を自動で判定する装置の開発を行った。本装置は、焼成過程においてパン表面の特徴を解析し、適切な焦げ目を判定する機能を備えている。しかし、現段階では特定の焦げ目を 1 種類しか判定することができず、利用者ごとの好みに応じた焼き加減の選択ができないという課題がある。そのため、ユーザーが好む焼き加減を設定可能とする機能の追加が必要とされる。また、小麦の全粒粉 100%の食パン(図 18) の適切な焦げ目を入力画像として使用したみたが、差分が 500 以上になり。適切な焦げ目と判定することができなかった。そのため、白いパンだけでなく、別種類のパンにも対応していく必要がある。

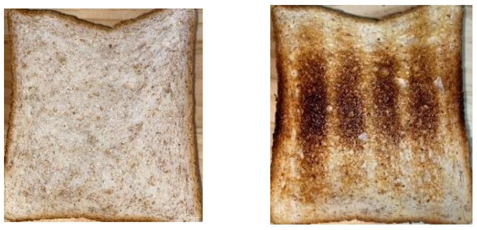


図 18.　小麦の全粒粉 100%の食パン

また、加熱を実際に行う際、装置のカメラやその他の部品が高温環境にさらされることで、性能の劣化や破損のリスクが懸念される。そのため、装置に使用する素材を耐熱性の高いものに変更し、長時間の過熱環境に耐え得る設計が求められる。

加えて、焦げ目の判定精度を向上させるため、異なる焼き加減やパンの種類に対応したデータセットの拡充と、高精度な機械学習アルゴリズムの適用が考えられる。さらに、装置の動作に係るエネルギー効率を向上させることで、商業利用や家庭用機器としての実用性を高めることも重要である。これらの改良を通じて、より多様なニーズに対応可能な装置の実現を目指している。

参考文献

[1]　わが国の品質管理実践革新の可能性と品質コストが果たす役割に関する考察
https://www.waseda.jp/fcom/soc/assets/uploads/2015/01/wcom4[Accessed: Jan. 30, 2025].

HW/SW Co-Design for a Variational AutoEncoder targeting Anomaly Detection on FPGA

Tuan-Phong Tran, Thien-Duy Ho, Tung-Bach Nguyen, Xuan-Tu Tran, Duy-Hieu Bui
VNU Information Technology Institute
144 Xuan Thuy Road, Cau Giay District, Hanoi, Vietnam
Email: duytper@gmail.com

***Abstract*—Variational Autoencoder (VAE) is a generative model based on autoencoders, utilizing a latent space with probabilistic distribution to generate new data. VAE is applied in various fields, such as image generation, data compression, and text generation. This work presents the design and implementation of a VAE for anomaly detection on FPGA by using co-design techniques to optimize the VAE model in software first and then accelerate the encoder part in hardware to improve performance. The base Variational Autoencoder (VAE) is from the work in [1] on Kaggle, which uses the MVTec AD dataset for anomaly detection. To adapt the model for FPGA deployment, we redesigned the reference model by reducing kernel size and the number of filters, removing the batch normalization layer, and replacing the leaky ReLU activation with ReLU. These modifications reduced the model size by a factor of 10 while only resulting in a slight accuracy drop. The proposed model is then retrained with a 16-bit fixed-point representation instead of a 32-bit floating point. Due to the time limitation, the hardware accelerator for the encoder was designed using High-Level Synthesis (HLS) to accelerate the development time, while the rest of the proposed VAE model was performed in software on the Pynq-Z2 board. The hardware implementation results of the encoder show that it runs more than 180 times faster than the software one while occupying a small area of the programable logic on the Pynq-Z2 development board. The full system of the encoder IP integrated with the Zynq SoC occupies less than 2500 slices on Pynq-Z2 FPGA.**

Keywords— Variational Autoencoder, High-Level Synthesis, Hardware Description Language

I. INTRODUCTION

Maintaining product quality is extremely important in the manufacturing industry. If anomalies are not detected in time, they can lead to equipment damage, reduced operational efficiency, and potentially negatively impact employees' health and morale. Automated anomaly detection systems help reduce costs by enabling early issue detection, which minimizes maintenance expenses and decreases production downtime caused by failures. As a result, anomaly detection methods have become essential in various fields, including healthcare, manufacturing, and food quality monitoring.

Manual inspection methods rely on human observation and judgment, such as visual product inspection, manual sorting, or checking basic parameters using simple tools like scales or measuring devices. Although still widely used, these methods can be inaccurate and prone to human error, especially under high workloads or when high precision is required. Similarly, traditional machine learning approaches, such as classification models, require manual feature engineering, which is time-consuming and can lead to suboptimal results, particularly when dealing with highly dynamic and complex data.

Our project uses variational autoencoders (VAE) as the core deep learning model to apply anomaly detection in hazelnuts. The goal is to detect defects or anomalies in hazelnuts that may indicate production issues, such as damage or irregularities in size, shape, or color. VAE models are well-suited for anomaly detection due to their ability to learn complex data distributions and generate new data points from these learned distributions. By training the VAE on a dataset of normal hazelnuts, the model learns to represent the typical characteristics of a hazelnut. It can identify deviations from this learned normal distribution when presented with new data, thereby detecting anomalies.

The software implementation of VAE presents several challenges. One of the main challenges is the high complexity of the original VAE model, which contains approximately three million parameters, making hardware deployment difficult. On the hardware side, optimization is challenging due to resource constraints, as VAEs typically have millions of parameters, making deployment on FPGAs or embedded systems difficult. Latency constraints are also critical in real-time applications, requiring an efficient inference process to ensure timely processing. Moreover, memory bandwidth poses another challenge, as managing large models in memory-limited environments requires careful optimization.

We applied a Hardware/Software co-design strategy, where the FPGA handles computationally intensive tasks while lighter tasks are executed in software. In our implementation, the VAE model—specifically the Encoder block—is designed as a hardware IP core, while the sampling layer and decoder are implemented in Python on the Pynq-Z2 FPGA development board. By adopting this co-design approach, we optimize system performance and efficiency, enabling the deployment of large-scale VAE models on resource-limited FPGA platforms while reducing latency and improving real-time processing capabilities.

To address these challenges, we redesigned the reference VAE model in [1] to reduce the number of parameters from 3 million to approximately 300 thousand for hazelnut anomaly detection while maintaining the model accuracy. Our optimization facilitates easier hardware deployment without compromising performance. By combining model complexity reduction, synthetic anomaly data generation, and FPGA implementation, we successfully tackled key challenges in applying deep learning for defect detection in hazelnuts.

The rest of this paper is organized as follows. Section II presents our proposed VAE model's architecture. The reference and proposed architecture will be evaluated in terms of model size, performance, and accuracy. The proposed hardware architecture for the encoder is depicted in Section III. After that, Section IV presents the hardware implementation and the system integration of the encoder model with the Zynq processing system on Pynq-Z2 FPGA development board. Finally, there are some conclusions and perspectives in Section V.

II. Proposed Variational AutoEncoder Architecture

We first started with a reference VAE model on Kaggle, which uses the MVTec Anomaly Detection (MVTec AD) dataset. This model has over 3 million parameters with an accuracy of approximately 90%. To optimize this model for hardware implementation, we used the HW/SW co-design approach to optimize the model in terms of the number of parameters and throughput while maintaining similar accuracy. Our proposed model is more suitable for hardware implementation on FPGA. This section presents our proposed VAE model for hazelnut anomaly detection along with the optimization techniques to implement the target model more efficiently on FPGA.

A. *Hazelnut anomaly detection dataset*

The MVTec Anomaly Detection (MVTec AD) dataset [3] is a well-known anomaly detection benchmark designed to evaluate algorithms used in industrial applications, such as defect detection in manufacturing processes. This dataset, provided by MVTec Software GmbH, consists of images categorized into object and texture types, totaling 1,535 images. These images are divided into training and test sets, with approximately 1,000 images used for training and 500 images designated for testing. The training set includes both normal (defect-free) and defective samples, while the test set contains images with previously unseen anomalies, allowing for a comprehensive evaluation of model performance. Some samples of the MVTec AD dataset are displayed in Fig. 1.

Regarding annotations, each image in the MVTec AD dataset is labeled with ground truth information about the location of defects. These annotations are provided as either pixel-wise segmentation masks for localizing defects or image-level labels indicating the presence of anomalies. This dual annotation format supports localization tasks (identifying the exact location of defects) and classification tasks (distinguishing between normal and defective objects).

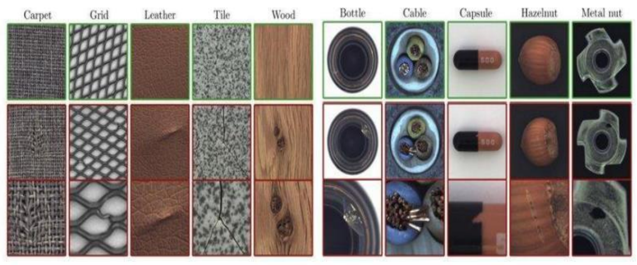


Fig. 1. MVTec Anomaly Detection dataset's samples [2].

B. *The proposed Variational AutoEncoder architecture*

1) *Original architecture*

Fig. 2 illustrates the architecture of the reference model in [1]. It contains three main parts: the encoder, the decoder, and latent space sampling. The entire VAE model consists of over 3.5 million parameters, with the encoder responsible for over 1.5 million parameters and the decoder responsible for 2.0 million. The latent space introduces a learnable sampling mechanism from the mean and the variant from the encoder. The model effectively compresses and reconstructs images while ensuring smooth transitions in the latent space through the reparameterization trick [xxx].

The encoder compresses input images in a 32-bit floating point (FP32) format into a latent representation through four convolutional layers. Each convolutional layer applies 3×3 filters to the input feature maps, followed by Batch Normalization to stabilize training and accelerate convergence. Finally, the Leaky ReLU activation function is used for non-linearity, enabling the network to learn complex patterns while mitigating the vanishing gradient problem. The number of filters for the convolution layers is 64, 128, 256 and 512, respectively.

Instead of directly mapping to a latent representation, the encoder outputs two vectors, mean and log variance, defining a Gaussian distribution in the latent space. The sampling step ensures the model learns a continuous latent space, which is crucial for generating smooth interpolations.

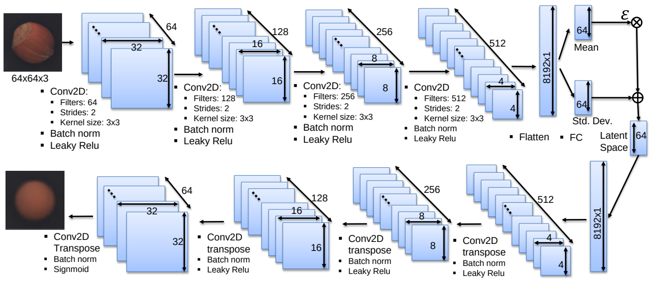


Fig. 2. The reference VAE architecture in [1].

The decoder reconstructs images from the sampled latent vector. It begins with a dense layer that expands the latent space into a shape matching the encoder's last feature map, followed by a reshaping operation. The decoder then employs transposed convolution layers to upsample the feature maps. The first transposed convolution layer restores spatial resolution to 8×8×256, followed by batch normalization and activation. The successive layers progressively increase the spatial dimensions to 16×16×128 and 32×32×64 while reducing the number of filters. The final layer generates an output of shape 64×64×3 using a sigmoid activation function, ensuring that pixel values remain in the valid range.

The model's accuracy on the training and test sets is shown in Fig. 3. Both training and validation results reaches SS 90%. Meanwhile, the loss function for the training and test sets is approximately 0.001 and 0.002, respectively.

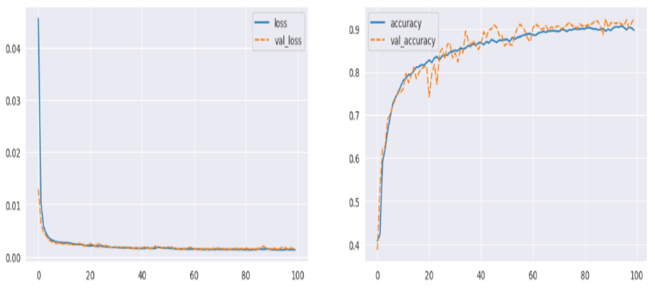


Fig. 3. The accuracy and loss of the reference model with the hazelnut dataset.

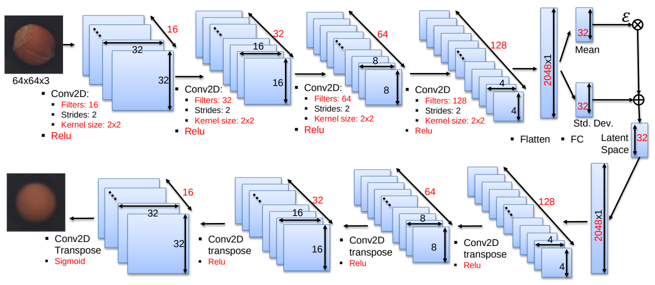


Fig. 4. The proposed VAE model's architecture.

2) *Proposed architecture*

FPGA has limited logic elements, memory, and DSP blocks. Additionally, real-time applications require low latency, necessitating optimized inference processes for rapid decision-making. Excessive resource usage can also increase power consumption, impacting the performance of edge devices. Therefore, optimizing and fine-tuning the model architecture maximizes parallel processing capabilities, reduces memory bottlenecks, saves energy, and ensures hardware-friendly implementation on FPGA.

The reference VAE model faces challenges related to resource constraints and suboptimal execution time when being implemented on FPGA because it contains many parameters with floating point operation. Additionally, in the anomaly detection phase, the output of the VAE model only provides a reconstructed image without highlighting the anomalous regions. The reference work in [1] relies on an existing library [5] that identifies anomalies from the reconstructed image to obtain an image with detected anomalies. We attempted to install this library on our PYNQ-Z2 board, which is used for this project, but found it infeasible due to its limited memory (only 512MB DRAM available). The library requires more than 1GB of DRAM to run. We replaced the anomaly detection method with a custom Python program to predict and detect anomalies. While this custom solution is less effective than the pre-existing library due to time constraints, it still ensures anomaly detection capability.

The proposed architecture of the proposed model is illustrated in **Error! Reference source not found.**. To reduce the model size, we decreased the kernel size from 3×3 (in the original model) to 2×2 and reduced the number of filters in both the encoder and decoder blocks. A smaller kernel reduces the number of convolution operations, while fewer filters decrease the total number of parameters, optimizing bandwidth and processing speed. Batch normalization was also removed because it requires statistical computations (mean, variance) and complex arithmetic operations (division, square root), which are not suitable for hardware implementation on FPGA. Additionally, Leaky ReLU was replaced with ReLU since FPGA implementations typically prioritize simple operations. ReLU only requires a comparison and assignment, whereas Leaky ReLU involves a multiplication with a small coefficient (αx) when $x \leq 0$, increasing latency and consuming more hardware on FPGA.

After research and development, we successfully created a new VAE model with approximately 300 thousand parameters and nearly the same accuracy as the original model, which was 89%. The training loss stays around 0.001. The proposed model maintains accuracy and improves effectiveness for hardware deployment. Fig. 5 shows train and test results in terms of the accuracy of the proposed model with the hazelnut dataset.

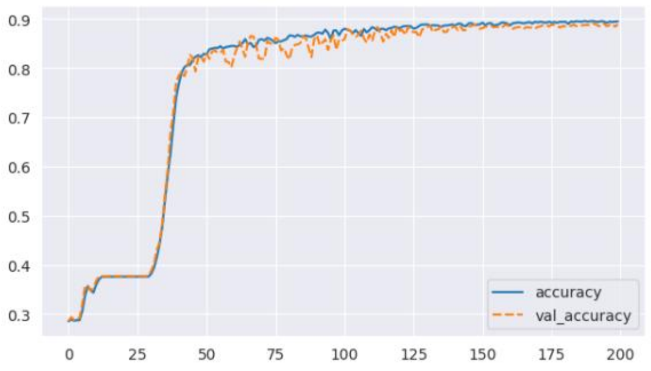


Fig. 5. Train and test accuracy of the proposed model on the hazelnut dataset.

Since the reference model is implemented in software using the TensorFlow framework, processing floating-point data during training and performing calculations within the network is straightforward and efficient on computers. However, floating-point computation is costly in resource-constrained devices and hardware implementation on FPGA. Therefore, we proposed to use quantization-aware training to convert all input images and model parameters, including weights and biases, to 16-bit fixed-point numbers [5] with 2 bits for the integer part and 14 bits for the fractional part. This reduces the proposed model's size and computation complexity. Fixed-point computation is equivalent to integer computation, which is less complex and more efficient than floating-point one. After converting to fixed-point format, we updated the weights and biases accordingly and tested the model to compare its accuracy when working with floating-point and fixed-point representation. Fig. 6 shows the comparison of the reference model with our proposed one. The reference model is 30 times larger than the proposed model in terms of parameter size.

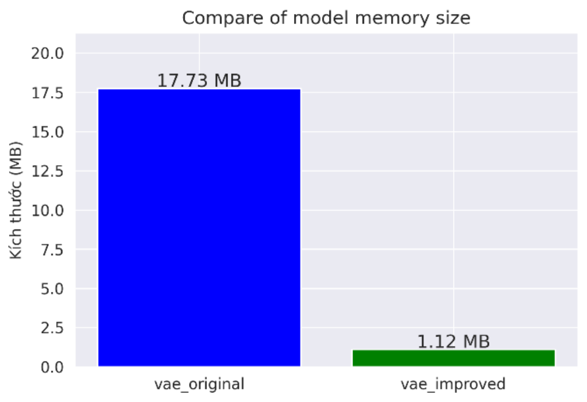


Fig. 6. Comparison of the parameter size between the reference model and the proposed model.

III. HARDWARE ARCHITECTURE

After optimizing the model to reduce the model size and computation complexity using the fixed-point reprentation, we continue applying the HW/SW co-design approach to optimize the system. Due to the time limitation, we decided to implement the encoder using High-Level Synthesis on FPGA while the rest is implemented in software using Python on Pynq-Z2 development board. This section presents our hardware architecture and its integration with the Zynq processing system.

A. Our design flow

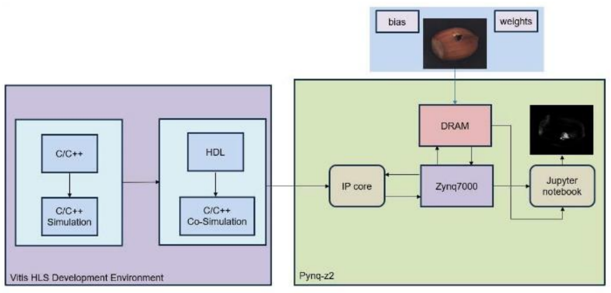


Fig. 7. Our design flow to implement the VAE model on Pynq-Z2 development board.

Fig. 7 presents our design flow to implement the proposed VAE model on Pynq-Z2 development board. Firstly, we train the model and extract parameter files from the trained model, including weights and biases in Tensorflow. These parameters are essential for reproducing the model on another platform, specifically in C++. The model is then translated from its Deep Learning framework into C++ source code, enabling smooth integration into embedded systems. Next, the Vitis HLS tool converts the C++ code into Verilog generates an IP core. Finally, the IP core is integrated into the Zynq-7000 SoC on the Pynq-Z2 board. The image data (already converted to a fixed-point format in the first step) is loaded into DRAM along with the kernel weights and biases via the Zynq processor. The extracted data from DRAM is then reconstructed into an image using Python running on Pynq-Z2 development board.

B. High-Level Synthesis and Pynq-Z2 development board

High-Level Synthesis (HLS) is a technology that converts high-level C/C++ code into hardware description languages such as VHDL or Verilog for execution on an FPGA. One of its main advantages is the significant reduction in development time, as writing C/C++ code is much faster than developing RTL code in Verilog or VHDL. Additionally, HLS allows for easier optimization, enabling quick modifications to improve algorithm performance. It also integrates seamlessly with tools like Xilinx's Vivado, which supports exporting RTL code for FPGA implementation. However, HLS also has some drawbacks, such as potentially suboptimal hardware compared to hand-written RTL designs, leading to resource utilization and performance inefficiencies. Moreover, the generated RTL code may not be fully optimized or easily controllable, making fine-tuned hardware design more challenging.

For our project, we chose HLS due to the time limitation in implementing the proposed model using HDL. HLS provides a powerful approach to implementing complex algorithms and computations on hardware using high-level languages like C++, making it a suitable choice for accelerating development while maintaining efficiency. To deploy our design, we use the Pynq-Z2 FPGA development board, which is based on the Xilinx Zynq-7000 dual-core processor and is designed to run PYNQ – a framework that simplifies FPGA programming with Python. The Pynq-Z2 features a Zynq XC7Z020-1CLG400C FPGA, which integrates an ARM Cortex-A9 processor with FPGA fabric, along with 512MB DDR3 RAM, microSD storage, and multiple interfaces such as HDMI, USB, Ethernet, PMOD, and Arduino headers. It is a versatile platform for various applications, including real-time image and video processing, machine learning, IoT, and embedded systems. Additonally, tts ease of programming with Python makes it an ideal choice for education and research.

C. Hardware and software design

1) Proposed hardware architecture

The proposed hardware architecture is presented in Fig. 8. The system is divided into three stages: the first stage is initialization, the second stage is data processing, and the final stage is storage and result retrieval. In the first stage, the input image data is loaded into DDR3 memory along with the weights and biases of the VAE model. Next, in the second stage, the Cortex-A9 transfers the configuration of the IP along with the address of the image data, weights, and biases in DDR3 Memory to the encoder IP to activate the hardware accelerator. The encoder uses the AXI4 master interface to load data into the Input Mem and Weights/Bias Mem in the programable logic via the AXI bus. The Conv2D (2x2) block then performs the convolution operation. A Finite State Machine (FSM) controls the entire processing flow. In the final stage, the processed data is stored in the Output Mem and then transferred to DRAM in the processing system through the AXI4 Master Interface. The processing system continues to run the rest of the anomaly detection process in Python.

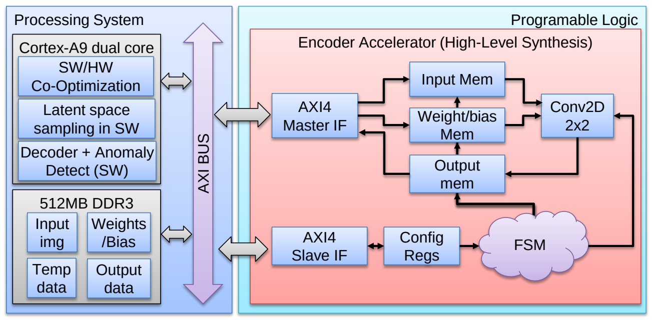


Fig. 8. The proposed hardware architecture for the encoder module.

The proposed hardware architecture has been implemented in C++ with the optimization for High-Level Synthesis with Vitis HLS 2021.2. The AXI4 master and slave interface is automatically inferred through the HLS pragmas. The proposed hardware architecture has been tested in C++, co-simulated with the generated hardware module from HLS and generated into a Vivado IP that can be integrated in to a system-on-chip.

2) Integration of the encoder IP into Zynq-7000 SoC in Vivado

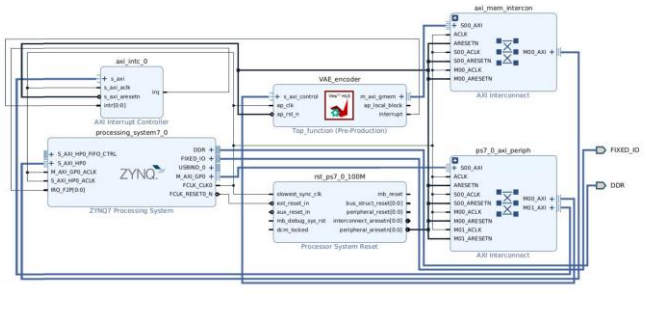


Fig. 9. VAE encoder module's integration to the Zynq processing system on Pynq-Z2 development board.

Fig. 9 shows the system architecture of the encoder IP with the Zynq-7000 via the AXI4 interface. The proposed encoder IP has two interfaces: one master interface for direct memory access and one slave interface for configuration. The two interfaces are connected to the AXI bus. The Zynq processor controls the encoder IP and can directly access the DRAM memory for the input data, weights, and bias. This arrangement reduces the workload on the processor during the encoder execution.

Additionally, the system's interrupt mechanism in the connection between the Zynq7000 SoC and the IP improves communication efficiency between the IP core and the Zynq processor (Cortex-A9). Instead of the processor continuously polling the IP's status, the IP proactively sends an interrupt to the processor when the encoder's computations are complete. This reduces the processor workload and enhances system performance by allowing the processor to respond only when an interrupt is triggered.

IV. Hardware implementation results and evaluation

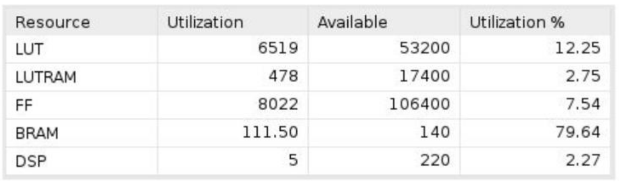
A. *Synthesis results*

During the hardware design process using HLS, we first develop a software model to simulate and verify the algorithm on a computer. Once the algorithm is confirmed to work correctly, the next step is to convert this model into C/C++ code suitable for execution in the HLS tool. When writing C/C++ code for HLS, we use integer data types, minimize complex loops, and optimize memory usage. The code is then synthesized into RTL (Verilog/VHDL) for deployment on FPGA. Finally, after obtaining the RTL code, the design is integrated and tested on the FPGA to ensure it meets system requirements.

The proposed encoder IP has been successfully implemented using Vitis HLS 2021.2 and integrated into Zynq-7000 SoC using Vivado 2021.2. It has been successfully synthesized, implemented, and validated on Pynq-Z2 development board. The software used Python to run on Linux on Pynq-Z2 development board.

Fig. 10 shows the resource utilization of the proposed design on Zynq XC7Z020 FPGA. The proposed design occupies only over 7300 flip-flops and about 5700 LUTs, 7% and 11% of the total flip-flops and LUTs on the targeted FPGA Chip, respectively. It also occupied 5 DSP slices for the convolutional computations and 111.5 BRAM tiles. The total number of slices occupied is less than 2500, about 18.5% of the total number of slices on Zynq XC7Z020 FPGA. Fig. 11 shows the proposed system mapped into FPGA resource after placement and routing.

The encoder IP utilizes about 6,000 flip-flops and 4,650 LUTs. It occupies about 2000 slices in total (about 15% of the total number of slices) with 111.5 BRAM tiles. This shows that our proposed encoder IP has a small area. The only drawback is the number of BRAM tiles which can be furthur improve in the near future.



Resource	Utilization	Available	Utilization %
LUT	6519	53200	12.25
LUTRAM	478	17400	2.75
FF	8022	106400	7.54
BRAM	111.50	140	79.64
DSP	5	220	2.27

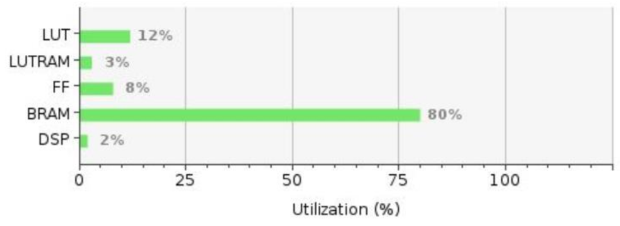


Fig. 10. Resources Utilization.

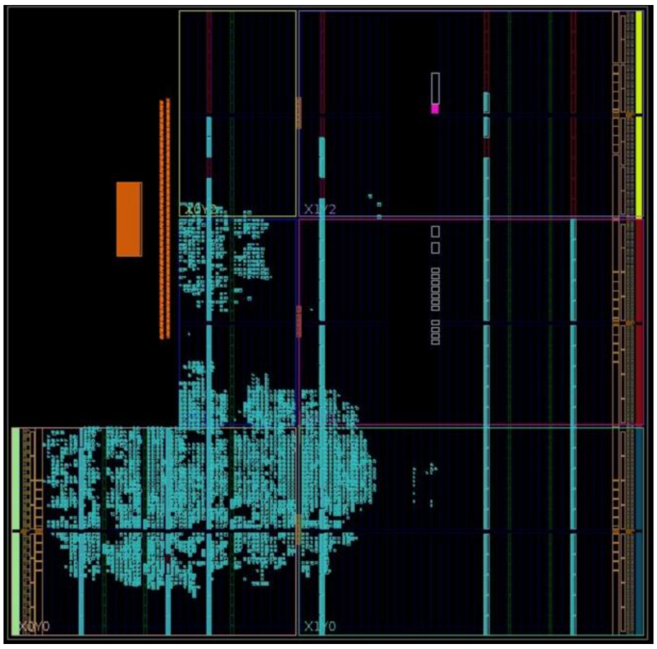


Fig. 11. The proposed encoder design after placement and routing on FPGA.

B. *Runtime between encoder hardware and software*

After implementing the encoder IP into FPGA, we integrate the hardware IP and the software using Pynthon on Pynq platform for Pynq-Z2 development board. Pynq platform allows to program the programable logic and control the IPs through its python interface. This allow us to use Python to build the driver for the encoder IP and integrate into the rest of the anomaly detection application for hazelnut.

Fig. 12 illustrates the execution time comparison between software (SW) and hardware (HW) implementations of the encoder in the proposed VAE model. The results show a significant improvement in execution speed when using hardware acceleration. Specifically, the software implementation takes 5.12 seconds, whereas the hardware implementation only requires 0.028 seconds. This translates to an approximate 180x speedup, reducing the execution time of the encoder to just 3.76% of the software's execution time.

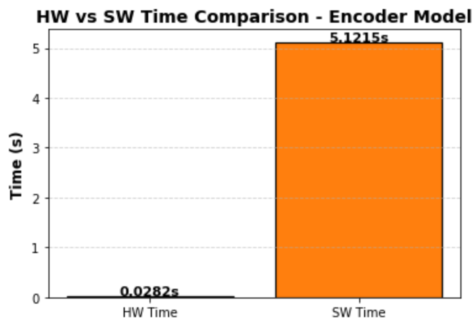


Fig. 12. Comparison of the execution time between the hardware encoder and the software encoder on the Pynq-Z2 development board.

C. *Model Results: Hardware Encoder - Software Decoder*

Fig. 13 shows the final results of the model using the hardware encoder's output as input for the software decoder that produces generated images consistent with those obtained from a fully software-based implementation. This confirms that the hardware encoder preserves essential latent representations, ensuring reliable reconstruction by the software decoder.

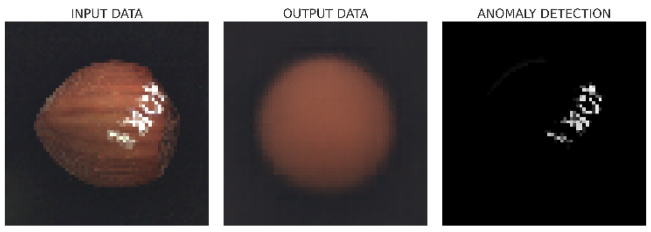


Fig. 13. Reconstructed image using the software-based decoder and anomaly detection prediction.

V. CONCLUSIONS AND FUTURE WORKS

Variational AutoEncoder models face many challenges when implemented on FPGA. The encoder and decoder use the convolution 2D and convolution 2D transpose for data processing, while the latent space sampling is complicated. In this work, we proposed using a hardware/software co-design approach to implement a VAE for hazelnut anomaly detection. We have significantly reduced the number of parameters compared to the reference work from 3.5 million to about 300 thousand (approximately 10 times smaller). Additionally, we have converted the model to operate and compute in fixed-point format to accelerate hardware execution and increase the throughput. Due to the time limitation, only the encoding part was implemented into FPGA using High-Level Synthesis. The proposed encoder IP has been successfully integrated into Zynq-7000 SoC with low hardware utilization. The testing software shows that the encoder IP can run much faster than the software one. The hazelnut anomaly detection has been implemented on the Pynq-Z2 development board using the Pynq framework. The test results show that it maintains the accuracy of the software model. In the near future, we would like to optimize the proposed system in terms of block RAM to reduce the block RAM usage and implement the decoder into FPGA. The latent space sampling can be implemented in software to reduce the system's complexity.

REFERENCES

[1] Jiří Raška, "MVTect - HazelNut - Variational AutoEncode II." Retrieved from https://www.kaggle.com/code/jraska1/mvtect-hazelnut-variational-autoencode-ii

[2] Yajie Cui, Zhaoxiang Liu, Shiguo Lian, "A Survey on Unsupervised Anomaly Detection Algorithms for Industrial Images." Arxiv 2022. URL: https://arxiv.org/abs/2204.11161

[3] Paul Bergmann, Kilian Batzner, Michael Fauser, David Sattlegger, Carsten Steger: The MVTec Anomaly Detection Dataset: A Comprehensive Real-World Dataset for Unsupervised Anomaly Detection; in: International Journal of Computer Vision 129(4):1038-1059, 2021, DOI: 10.1007/s11263-020-01400-4.

[4] Francof2a, "Fixed Point Precision Neural Network for MNIST dataset," GitHub,[Online].Available:https://github.com/francof2a/fxpmath/blob/master/examples/Fixed_Point_Precision_Neural_Network_for_MNIST_dataset.ipynb[Accessed: Mar. 2, 2025]

[5] Van Looveren et al. Alibi Detect: Algorithms for outlier, adversarial and drift detection. URL: https://github.com/SeldonIO/alibi-detect

[6] PYNQ-Z2 Reference Manual v1.0. URL: https://dpoauwgwqsy2x.cloudfront.net/Download/pynqz2_user_manual_v1_0.pdf

[7] PYNQ: Python productivity for Adaptive Computing platforms. URL: https://pynq.readthedocs.io/en/latest/index.html

Low Resource CNN Variational Autoencoder By Utilizing BRAM and Shift Operation

Randy Revaldo Pratama*†, Naufal Afiq Muzaffar *‡, Steven Tjhia*§, and Nana Sutisna*
*School of Electrical Engineering and Informatics, Bandung Institute of Technology, Indonesia
Email: {13222012,13222025,13522103}@std.stei.itb.ac.id, nsutisna@itb.ac.id

***Abstract*—This paper presents a low-resource Variational Autoencoder (VAE) optimized for hardware implementation using a Convolutional Neural Network (CNN). To reduce computational costs, two key methods are introduced. First, shift operations replace traditional multiplications, significantly reducing hardware resource usage. Second, BRAM is utilized for output storage by efficiently computing addresses based on layer operations. Additionally, a Look-Up Table (LUT) is employed for activation functions, reducing computational complexity, especially for the sigmoid activation and the exponential function used in reparameterization. The entire network is implemented in Verilog-HDL on a PYNQ-Z1 FPGA board, utilizing a fixed point representation with 10-bit integer and 10-bit fractional precision for each layer. Experimental results demonstrate a high degree of similarity in which the average SSIM value of ShiftCNN is 0.3311, meanwhile the conventional CNN is 0.3397. Moreover the average Mean Square Error (MSE) of the reconstructed image using shiftCNN is a little bigger than without using it around 0.0554, while without using shiftCNN is around 0.0541. That degree of similarity between FPGA-based results and Python simulations validate the effectiveness of the proposed approach.**

***Index Terms*—Variational Autoencoder, FPGA, ShiftCNN, Verilog**

I. Introduction

Recently, technology has advanced rapidly, with zeros and ones flowing through digital veins. The artificial intelligence and machine learning industry demand a high performance and cost-effective architecture in order to take the lead in the industry. Among these, there is Variational Autoencoder (VAE), enabling efficient data compression, generation, and representation learning [1], which are critical for staying competitive in the ever-evolving AI industry.

Variational Autoencoders (VAEs) utilizing principles from neural networks. Unlike traditional autoencoders [1], which focus solely on reconstructing input data, VAEs introduce a probabilistic framework that enables them to generate new data samples by sampling from a learned latent space [1]. This makes VAEs particularly powerful for tasks such as image generation, anomaly detection, and data compression.

At the core of a VAE is the idea of encoding input data into a latent space and then decoding samples from this latent space to reconstruct the input or generate new data [1]. The training process involves optimizing both the reconstruction error and a regularization term. This balance between reconstruction and regularization is achieved through the use of the reparameterization trick, which allows for efficient gradient-based optimization [2].

While VAEs are typically implemented using high-level frameworks like TensorFlow or PyTorch, there is growing interest in deploying these models on hardware platforms such as Field-Programmable Gate Arrays (FPGAs). FPGAs offer the advantage of reconfigurability and parallelism [3], making them well-suited for accelerating the computationally intensive operations [3] involved in VAE inference. By leveraging hardware description languages like Verilog, we can design architectures tailored to the specific requirements of VAEs enabling efficient and low-latency implementations [4].

The integration of VAEs with FPGAs and Verilog opens up exciting possibilities for real-time applications, such as edge-based image generation [5], autonomous systems [6], and low-power embedded devices [7]. This combination of advanced machine learning techniques and hardware optimization represents a promising direction for the future of AI and embedded systems [8].

However, implementing VAE inference, especially for image generation in hardware components, often demands significant computational resources, particularly due to the intensive multiplication operations [9]. Additionally, the convolution operation also takes a lot of LUT [10] needed for both input and output, corresponding to the image size. To address this challenge, we propose utilizing ShiftCNN [11], a method that replaces multiplications with shift operations and additions, significantly reducing resource consumption. We also propose sequential operation by utilizing BRAM in order to reduce the LUT needed for this network. This paper explores the feasibility of implementing ShiftCNN and utilizing BRAM within a VAE inference on an FPGA, aiming to optimize performance and resource efficiency.

II. Proposed Design

A Variational Autoencoder is an enhancement of a traditional Autoencoder that mainly consists of three parts, which is Encoder, Latent Space, and Decoder [1]. It differs in the loss function and the way its latent variables are learned [1], making it capables to generating new data instances by modeling raw data into a probability distribution [1]. The architecture that we created is trained to reconstruct MNIST datasets, which are 28x28 pixels for both the input size and output size. For this type of data, Convolutional Neural Networks are mainly used for pattern recognition [12].

A. *Variational Autoencoder Architecture*

Our architecture is built in purpose to lower the resource used for the model implementation, that in general, the encoder has four layers starting from Convolutional 2D layer, Max Pooling layer, Dense layer with 100 neurons, and lastly Dense layer with 50 neurons. After that, we continue with two Dense layers with 2 neurons to calculate the z mean and z log variance score which then will be used to calculate the z sample score a.k.a. the latent space itself. As we can see, we use 2 neurons for the Dense layer because the target latent dimension is 2 so we want two pair of z mean and z log variance. On the decoder part, we have four layers starting from Dense layer with 50 neurons, Dense layer with 100 neurons, Convolutional 2D Transpose layer a.k.a Deconvolutional layer, and lastly Convolutional 2D layer.

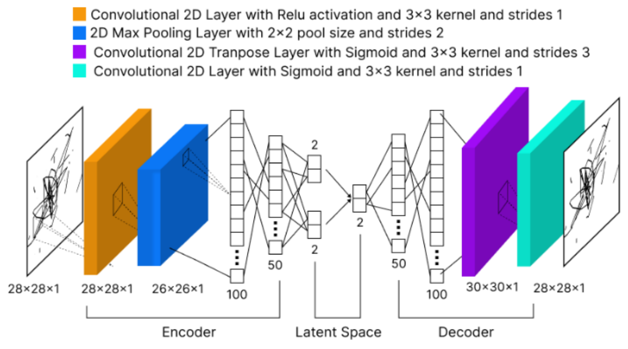

Convolutional 2D Layer with Relu activation and 3×3 kernel and strides 1
2D Max Pooling Layer with 2×2 pool size and strides 2
Convolutional 2D Tranpose Layer with Sigmoid and 3×3 kernel and strides 3
Convolutional 2D Layer with Sigmoid and 3×3 kernel and strides 1
28×28×1
28×28×1
26×26×1
100
50
2
2
2
50
100
30×30×1
28×28×1
Encoder
Latent Space
Decoder

Fig. 1. Variational autoencoder architecture.

The purpose of the Dense layer with 50 and 100 neurons is to minimize the number of weights needed so it doesn't take too much resources when implemented in the FPGA while also maintaining the model performance. We also utilize the Max Pooling layer to reduce the output neurons that are connected to the next layer.

B. *FPGA Architecture*

Our architecture mainly consists of CNN networks that can be considered expensive for hardware implementation. While we already minimized the architecture, looking at how the network has no depth, the resource needed to build the network still takes up a lot of computing resources, considering how convolution and deconvolution operation is done mathematically. Implementing mathematical operation in a hardware can be done either combinationally or sequentially. By nature, Convolutional Neural Network has kernels as its trainable properties [12]., and the process is done by multiplying its kernel with the pooled-input matrix then adding all the results before sliding to the next pooled-input.

Looking at this operation, it is very normal to think that this can be implemented by taking the pooled window as its input and multiplying it with its kernel. However, as the input grows, either for the pooled size or the bit size, it will take as much resource as the total of pool size times the bit size of each number, making it a non-resource friendly to be

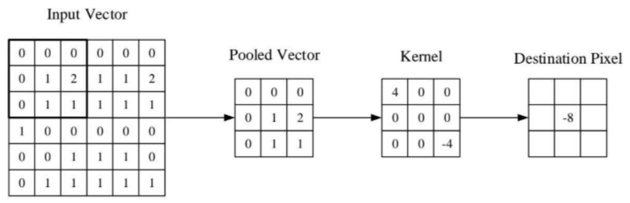


Fig. 2. Convolutional Neural Network [12].

implemented. This is where our novelty comes into action. We built a network that utilizes the usage of Block RAM to save space and reduce the need to use LUTRAM and FF to store all the data needed, while maintaining the speed by using shift-based operation for the multiplication. It works by making the process of operation sequentially, by taking one input each time, making it only needs to use logic-resources to do one operation at a time. This type of operation makes the network scalable, for both the input and the training parameter as long as the BRAM needed is enough to save all the training parameters and the output for each operation. The use of shift operation is to reduce the need of a multiplier block that will take up a lot of computing resources and reduce the time needed to do the operation.

C. *Process Development*

Starting from training the VAE model with Tensorflow in python, we extract the weights and biases from it. The weights will be quantized first so the value can be derived from the sum of the power of two. With that, the implementation in the FPGA will only utilize shift and addition operations.

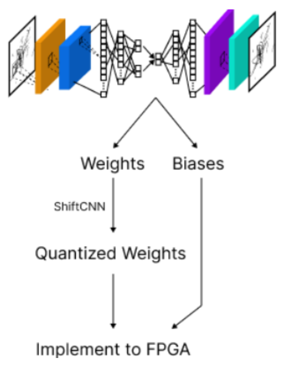


Fig. 3. Process Development.

III. Implementation and Engineering Considerations

This project's development is divided into two parts, which is the training part and the FPGA implementation part. The training part consists of building the network and training the model that is fully done in python using Tensorflow-Keras framework, so that the model can be used in the FPGA implementation part for the inference process. For the implementation part, it is developed fully using Verilog-HDL, which is divided into a smaller component that can be used in the top level entity. We used fixed-point format for the implementation, with the precision of 10 bits integer and 10 bits fraction, to see the resources utilization by making each layer holding the same amount of bit-length for the operation and not optimizing it.

A. *Python Implementation*

We train our model using Tensorflow with Adam optimizer. The architecture we adopt for training is already explained in the proposed design. After the training is complete, we save the weights and biases in .npy format. After another pre-processing, the saved weights and biases are then used to perform VAE inference.

a) Model Architecture:

- Encoder

TABLE I
ENCODER LAYERS

Layer	Input Shape	Output Shape
Conv2D	(28, 28)	(26, 26)
MaxPooling2D	(26, 26)	(13, 13)
Flatten	(13, 13)	(169)
Dense (100 neurons)	(169)	(100)
Dense (50 neurons)	(100)	(50)
[z mean] Dense (2 neurons)	(50)	(2)
[z logvar] Dense (2 neurons)	(50)	(2)
[z sample] Sampling (2 neurons)	(2) & (2)	(2)

- Decoder

TABLE II
DECODER LAYERS

Layer	Input Shape	Output Shape
Dense (50 neurons)	(2)	(50)
Dense (100 neurons)	(50)	(100)
Reshape	(100)	(10, 10)
Conv2DTranspose	(10, 10)	(30, 30)
Conv2D	(30, 30)	(28, 28)

b) ShiftCNN Weights Format:
The saved weights in the .npy file will be quantized with ShiftCNN and then converted to binary format and then saved to .coe file. The first bit represents the sign of the weight: 1 indicates it has negative value and 0 indicates it has positive value. The second bit represents the direction of the shift operation: 1 indicates that the shift direction is right and 0 indicates that the shift direction is left. The remaining bits represent the shift amount for the operand.

- **Sign Bit:** X_ _____
  X = 1 → The weight has a negative value
  X = 0 → The weight has a positive value
- **Shift Direction Bit:** _ X_ _____
  X = 1 → Right shift
  X = 0 → Left shift
- **Shift Amount Bit:** _ _ XXXX
  XXXX → Shift amount

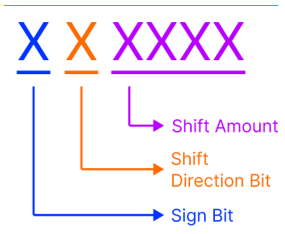


Fig. 4. ShiftCNN Weight Format.

B. *FPGA Implementation*

Our design is implemented using a PYNQ-Z1 FPGA that can use Jupyter-notebook for the input and output flow that is controlled by a FSM Controller. The way it sends data from Jupyter-notebook to FPGA is done sequentially, making it really fits with our proposed design.

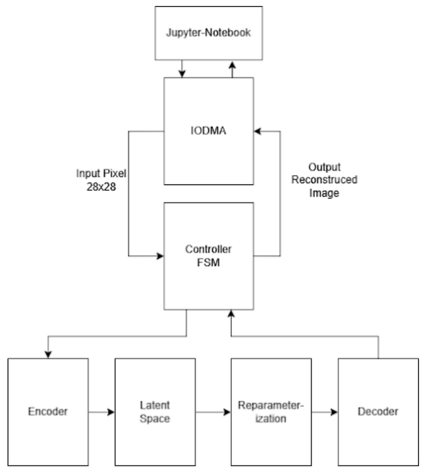


Fig. 5. FPGA implementation.

a) Convolution:
There are two modes of convolution that are being used in this project, Convolution 2D - Valid a.k.a Convolution, and Convolution 2D Transpose - Valid a.k.a Deconvolution. Those functions are made to behave like tensorflow function which has the same principle as how convolution operation usually works.. It differs on the output size parts, with convolution will reduce the size of input and deconvolution will increase the size of input. The implementation for this module is done by using FSM, which will save the input to its own BRAM, then doing the address calculation needed for the input and kernel for current index calculation. The multiplication operations are done using Shift operator, with the format of the kernel-weights that stated before.
Fig. 6. shows how convolution operation is implemented. It works by calculating the address of operation one by one, and then save it to the BRAM output everytime the kernel-size calculation is already satisfied, with the output address calculated previously. This way, we don't need to go back and forth to save and get the calculated data from BRAM output. However, this method only applies for convolution operation,

because for deconvolution, each calculated result needs to be saved to BRAM output because it needs to get the result from previous calculation from a specific address, so that it can get the right results. Additionally, deconvolution operation has a different boundary from convolution in the 2 outer loops, which is input size.

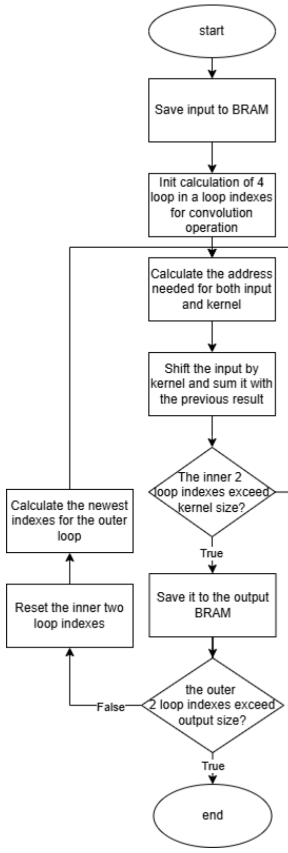

start
Save input to BRAM
Init calculation of 4 loop in a loop indexes for convolution operation
Calculate the address needed for both input and kernel
Shift the input by kernel and sum it with the previous result
Calculate the newest Address for the inner loop
The inner 2 loop indexes exceed kernel size?
False
True
Calculate the newest indexes for the outer loop
Reset the inner two loop indexes
Save it to the output BRAM
the outer 2 loop indexes exceed output size?
False
True
end

Fig. 6. Flowchart of Convolution Operation.

b) Matrix Multiplication:
Mathematically, matrix multiplication is done by multiplying the rows of multiplier with cols of multiplicand, then summing it for the value of that row x cols index. In verilog, we implemented it with the same principle as the convolution operation, by calculating the address of multiplier and multiplicand, multiplying the data using shift operation, then summing all the results and saving it to the BRAM output with specific address.

c) ReLU (Rectified Linear Unit):
ReLU is an activation function that is used the most in our network architecture. It results the same value as the input, if and only if the input is greater or equal to zero. Other than that, this function will result in zero. We implement it sequentially by using an enable signal that will start the comparing operation if it is 1. By implementing it sequentially, we have access to reset, enable, and done signals that are being used on the higher level module.

d) Look-Up Table (LUT):
Look-Up Table is one way to implement a function without the need to do calculation. It works by quantizing the function with resolution as the input size. We used LUT to implement exponential function and sigmoid activation function, for the calculation of reparameterization tricks that are being fed to decoder parts, and the activation function of the last 2 layers of decoder. The output for the exponential function has the same amount of bits as the input, which will make the input bound smaller than the actual input. For sigmoid function, considering the output range only from 0 to 1, we used 11 bits for the output to lower the to save space of the LUT. However, the output still has the same bit length as the input, by concating 9 bit to the MSB of sigmoid output.

e) Full Network Implementation:
The full network is implemented using the previous basis function, that for each layer will have their own FSM controller to control the operation flow of the basis function. From the previous chapter, we quantized the weight so that FPGA only need to do shift and addition operation, however we quantized it with resolution of 2, so that for one layer, will has 2 weights that after the shift operation, both results need to be added up along with bias before being fed to the activation function. For the reparameterization module, the randomness a.k.a epsilon is calculated using python, then is fed to the network along with the input pixel. All in all, the network will accept both input pixel and epsilon, and use the exponential LUT module that has been made before to calculate the input of the decoder layer.
Fig. 7. presents an overview of the full network implementation from the encoder's perspective. Notably, the decoder and latent space share the same implementation as the encoder shown in Fig. 7.

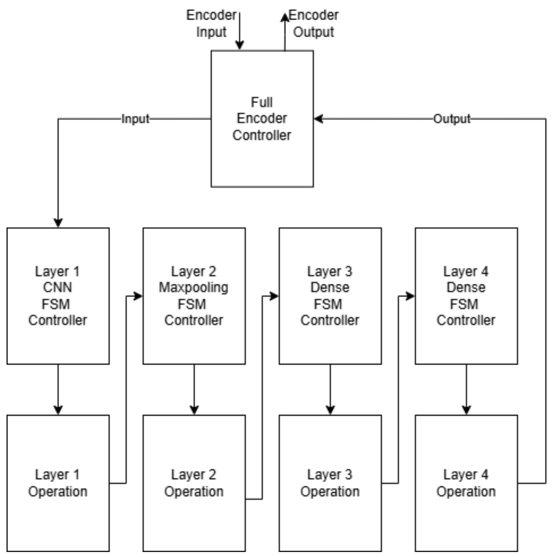


Fig. 7. General Overview of full network implementation.

f) AXI DMA and Jupyter-Notebook:
Xilinx-ZYNQ based FPGA uses IP core AXI DMA protocol to connect the data processing module with the system memory. AXI DMA can be controlled by an integrated Jupyter Note-

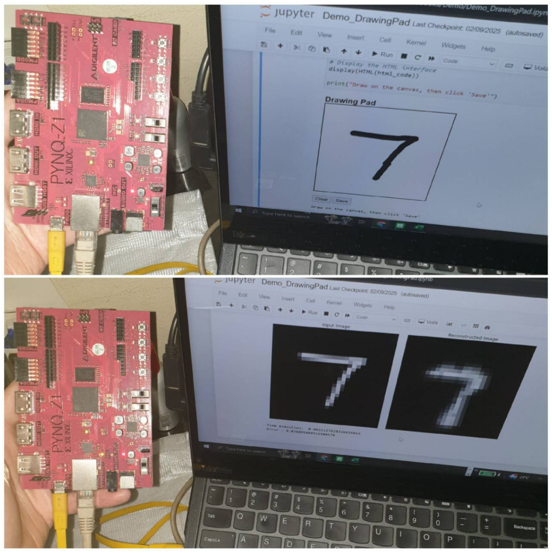


Fig. 8. Hardware setup using PYNQ-Z1 FPGA Board.

book on PYNQ-Z1 which provides convenience in managing the data, memory buffers and showing the received processed data.

The data received by the processing module is a normalized data in integer representation, with the following conversion flow. For example the pixel data from an image being sent is the number 255. This value then normalized by dividing it by 255. Next, the normalized data is converted into a fixed-point format with 10 bits for the integer part and 10 bits for the fractional part. Finally, the data is converted back to its decimal form, resulting in an input value of 1024.

IV. Results and Analysis

A. Model Analysis of Shift CNN Architecture

We used the *Tensorflow Keras MNIST* dataset as a test data set to evaluate our result. Three evaluation metrics are used to test the Shift CNN models: *Mean Squared Error* (MSE), *Structural Similarity Index* (SSIM), and *Binary Cross Entropy* (BCE). The MSE value ranges from 0 to 1 (because the image pixel value is normalized) indicating the error of one image to the other. Meanwhile the SSIM value ranges from -1 to 1: -1 indicates that one image is the exact inverse of the other, 0 indicates that there is no similarity, and 1 indicates that images are perfectly identical. BCE value ranges from 0 to infinite, smaller BCE value means smaller error and zero means the images are identical.

Below is the visualization of MSE, SSIM, and BCE value distributions from input image compared to reconstructed without shiftCNN, input image compared to reconstructed with shiftCNN, and reconstructed without shiftCNN compared to reconstructed with shiftCNN.

The result shows us that the average MSE of the reconstructed image without using shiftCNN is around 0.0541, meanwhile the average MSE of the reconstructed image using shiftCNN is a little bigger than without using it around 0.0554.

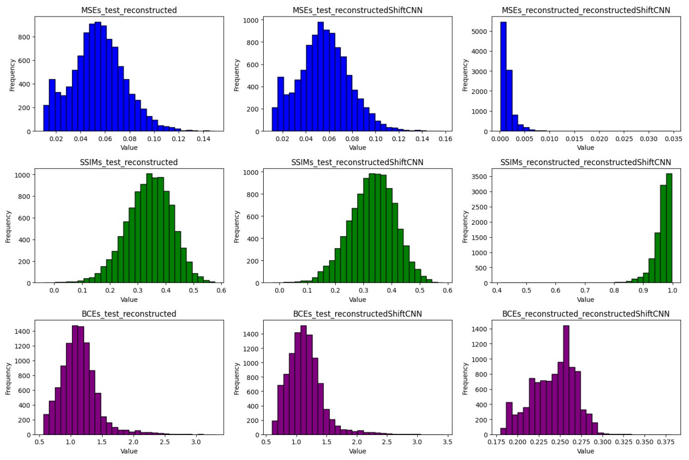


Fig. 9. (From top to bottom) MSE, SSIM, and BCE Histograms.

So as with the SSIM value, the average SSIM value without using shiftCNN is around 0.3397, meanwhile the average SSIM value of the reconstructed image using shiftCNN is around 0.3311. As we can see, the MSE and SSIM value of the reconstructed image with or without using shiftCNN only has a slight difference. Moreover, the average MSE value between the reconstructed image without shiftCNN and the reconstructed image using shiftCNN is very small around 0.0016, and the average SSIM value between the reconstructed image without shiftCNN and the reconstructed image using shiftCNN is 0.9615. Hence, the result of the reconstructed image with or without using shiftCNN doesn't have a significant difference.

B. FPGA Design Verification

The design is verified by comparing the error of reconstructed image from FPGA implementation result with python Shift-CNN result. The error is calculated using MSE for the verification process that can be seen in Table 5. Also, the resource utilization and the performance metrics both can be seen in Table 3 and Table 4. From resource utilization data, we can see that the network only uses 7072 LUT and 6705 FF, which can be considered small considering how CNN operation works. Also, this implementation doesn't optimize the bit width usage for each layer operation, making it more resource-friendly if we fully optimized all the operation done for each layer.

As shown in Table 5, both the FPGA implementation and the Python implementation using Shift-CNN yield comparable results in terms of the mean squared error (MSE) calculated from the reconstructed image. Considering the design implementation, the execution time presented in the table demonstrates sufficient efficiency for the inference phase with a 28×28 pixel input.

Conclusion

The hardware implementation utilizing BRAM and shift operations demonstrates excellent performance, as evidenced by the execution time required for each operation and the

TABLE III
UTILIZATION

Resource	Utilization	Available	Utilization(%)
LUT	7072	53200	13.29
LUTRAM	157	17400	0.90
FF	6705	106400	6.30
BRAM	97	140	69.29
DSP	44	220	20.00
BUFG	1	32	3.13

TABLE IV
PERFORMANCE METRICS

Clock Frequency	On - Chip Power	Worst Negative Slack (WNS)
100 MHz	1.628 W	0.227 ns

TABLE V
COMPARISON FROM IMPLEMENTED DESIGN WITH PYTHON

Value	MSE Value			FPGA Execution Time (s)
	No Shift CNN Python	Shift CNN Python	Shift CNN FPGA	
0	0.0660	0.0726	0.0719	0.0020
1	0.0121	0.0129	0.0146	0.0022
2	0.0855	0.0924	0.0924	0.0019
3	0.0701	0.0745	0.0788	0.0018
4	0.0534	0.0577	0.0583	0.0020
5	0.0875	0.0896	0.0897	0.0021
6	0.0767	0.0807	0.0794	0.0019
7	0.0336	0.0345	0.0345	0.0018
8	0.0751	0.0768	0.0769	0.0019
9	0.0512	0.0528	0.0528	0.0021

strong similarity between the FPGA and Python implementation results, with average SSIM value of ShiftCNN and conventional CNN are 0.3311 and 0.3397. Moreover, the shift CNN maintains high accuracy even with small quantization, with MSE for around 0.0554, while without using shiftCNN is around 0.0541. Additionally, by carefully optimizing the bit width used in each layer operation, the architecture can be further refined to reduce resource consumption, while without optimizing it, the resources used already really small considering the use of convolution operation, which can be seen in Table III. Lastly, the architecture still maintains its speed considering how the process is implemented, with execution time around 0.0019 to 0.0021 seconds.

ACKNOWLEDGMENT

We would like to express our sincere gratitude to Nana Sutisna, S.T., M.T., Ph.D. for his invaluable support and constructive feedback to this project.

REFERENCES

[1] A. Asesh, ”Variational Autoencoder Frameworks in Generative AI Model,” 2023 24th International Arab Conference on Information Technology (ACIT), Ajman, United Arab Emirates. 2023. pp. 01-06, 2023.

[2] D. P Kingma and M. Welling, ”Auto-Encoding Variational Bayes,” arXiv preprint arXiv:1312.6114, 2013. [Online]. Available: https://arxiv.org/abs/1312.6114v11. [Accessed: Jan. 28, 2025]

[3] J. C. Porcello, “Scaling up deep learning for AI using fpgas,” 2024 IEEE Aerospace Conference, pp. 1–13, Mar. 2024. doi:10.1109/aero58975.2024.10521423

[4] Z. Que, M. Zhang, H. Fan, H. Li, C. Guo and W. Luk, ”Low Latency Variational Autoencoder on FPGAs,” in IEEE Journal on Emerging and Selected Topics in Circuits and Systems, vol. 14, no. 2, pp. 323-333, June 2024, doi: 10.1109/JETCAS.2024.3389660.

[5] A. Vahdat and J. Kautz, “NVAE: A deep hierarchical variational autoencoder,” in Proc. Adv. Neural Inf. Process. Syst., vol. 33, 2020, pp. 19667–19679

[6] Govorkova, E., Puljak, E., Aarrestad, T. et al. Autoencoders on field-programmable gate arrays for real-time, unsupervised new physics detection at 40 MHz at the Large Hadron Collider. Nat Mach Intell 4, 154–161 (2022). https://doi.org/10.1038/s42256-022-00441-3

[7] M. Isik, M. Oldland, and L. Zhou, ”An Energy-Efficient Reconfigurable Autoencoder Implementation on FPGA,” arXiv preprint arXiv:2301.07050, Jan. 2023. [Online]. Available: https://arxiv.org/abs/2301.07050

[8] D. Marculescu, D. Stamoulis, and E. Cai, ”Hardware-Aware Machine Learning: Modeling and Optimization,” arXiv preprint arXiv:1809.05476, Sep. 2018. [Online]. Available: https://arxiv.org/abs/1809.05476

[9] X. Yang, S. Chaudhuri, L. Likforman, and L. Naviner, ”MinConvNets: A new class of multiplication-less Neural Networks,” arXiv preprint arXiv:2101.09492, Jan. 2021. [Online]. Available: https://arxiv.org/abs/2101.09492

[10] Y. Ma, Q. Xu, and Z. Song, ”Resource-Efficient Optimization for FPGA-Based Convolution Accelerator,” *Electronics*, vol. 12, no. 20, p. 4333, Oct. 2023. [Online]. Available: https://doi.org/10.3390/electronics12204333

[11] D. Gudovskiy, “ShiftCNN: Generalized Low-Precision Architecture for Inference of Convolutional Neural Networks,” arXiv preprint arXiv:1706.02393, Jun. 2017. [Online]. Available: https://arxiv.org/abs/1706.02393. [Accessed: Jan. 28, 2025]

[12] K. O’Shea and R. Nash, “An Introduction to Convolutional Neural Networks,” Dec. 2015.

Design and Implementation of Point Cloud Data Generation Circuit Using Variational Autoencoder

Noritomo Okamoto, Shun Matsumoto, Kazuma Mori, Itsuki Karube
Department of Electrical Engineering, Faculty of Engineering, Chiba University, Japan
Graduate school of Engineering, Chiba University, Japan
Email: 21t1061h@student.gs.chiba-u.jp

Abstract: We designed and implemented a circuit for a point cloud generator using VAE. The system can generate point cloud data 2.5 times faster than the CPU. In addition, we have developed an optical reconstruction system for the created point cloud data through holography.

Keywords—Variational Autoencoder, Point Cloud, FPGA, Pipelining, Holography

1. Introduction

Point cloud data is three-dimensional (3D) and used in various fields such as architecture, autonomous driving, XR contents, etc. However, making point cloud data requires modeling and surveying techniques, which are time-consuming and labor-intensive. Therefore, this project developed a system for generating new point clouds at high speed.

2. Point Cloud and Holography

Point cloud data is a data format that represents an object as a set of points. Each point is represented by 3D spatial coordinates (x, y, z) and has 3D information.

There are a wide range of applications for point cloud data, and this article describes one example of its use in holography.

Holographic 3D display is a 3D imaging technology that provides the depth and parallax information of scene without using customized glasses.

A computer-generated hologram (CGH) is created from point cloud data and displayed on an optical element. By irradiating the CGH with a laser, the 3D image floating in the air can be projected.

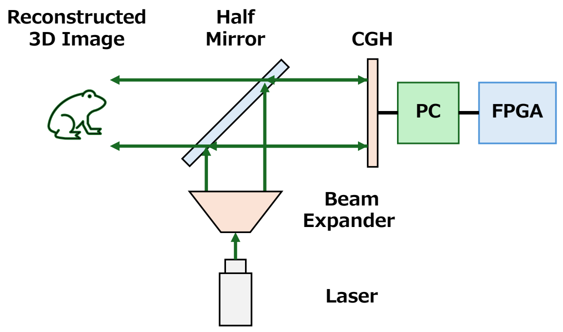


Fig. 2.1. Image Reconstruction from CGH

The objective of this project is to generate point clouds that interpolate transitions between different point cloud data via VAE. As an example, we developed a system in which VAE learns point clouds of tadpoles and frogs, estimates the growth process of frogs by controlling latent variables, and outputs these point clouds.

We used Blender to create training data (Fig. 2.2.), the number of training data was increased by randomly scaling and rotating the data in python code.

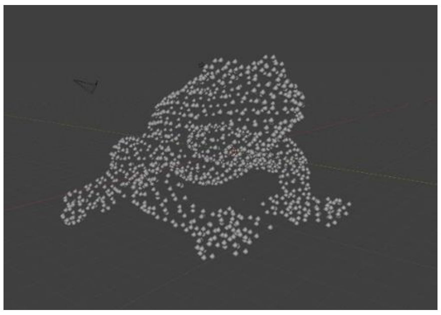


Fig. 2.2. Point Cloud on Blender

3. Model

We are using VAE to do the generation. The full structure of our model is shown in Fig. 3.1. The details for our model will be described in the following subsections.

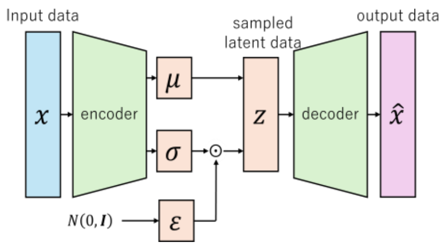


Fig. 3.1. Structure of VAE

3.1. Encoder

We will first introduce the encoders we used in our model. The structure is shown in Fig. 3.2.

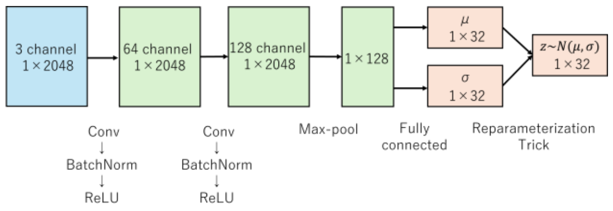


Fig. 3.2. Structure of Encoder

We used 1D convolutional networks with filter size 1, which is equivalent to a fully connected layer in 2D. Then we use a fully connected layer to get the mean and variance for the latent Gaussian distribution.

3.2. Sampling

Let μ and σ denote the output from the encoder with input x, then we sample z from $q(z|x) = \mathcal{N}(\mu, \sigma)$ and pass it to the decoder.

3.3. Decoder

For the decoder, it takes the sampled latent variable z as input and the output $\hat{x}$ is a $2{,}048 \times 3$point cloud matrix. In this project, we use a fully connected layer.

3.4. Loss Function

Assume we have N data points $x^{(1)}, \dots x^{(N)}$, the loss function $\mathcal{L}$ is defined to be:

$$\mathcal{L}(\theta, \phi; x^{(i)}) = \mathbb{E}_{q_\phi(z|x^{(i)})}[\log p_\theta(x^{(i)}|z)] \\ -D_{KL}\left(q(z|x^{(i)}) \parallel p_\theta(z)\right) \qquad (3.1)$$

The first term is the reconstruction loss and the second term is the latent loss. In this system, we use chamfer distance as reconstruction loss and KL divergence as latent loss.

3.4.1. Chamfer Distance

The Chamfer distance between two sets X, Y of point clouds is defined to be:

$$CD(X,Y) = \sum_{x \in X} \min_{y \in Y} \|x - y\|_2^2 \\ + \sum_{y \in Y} \min_{x \in X} \|x - y\|_2^2 \qquad (3.2)$$

For each point, the algorithm of Chamfer distance finds the nearest neighbor in the other set and sums the squared distances up.

3.4.2. KL Divergence

KL divergence is a measure of the difference between two probability distributions P and Q. In this system, $p(z)$ is the standard Gaussian, $q(z|x)$ is the distribution generated by the program, and $D_{KL}(q(z|x) \parallel p(z))$, the difference between them, is the loss function.

In this model both $p(z)$ and $q(z|x)$ are Gaussian; in this case, the resulting estimator for this model and datapoint $x^{(i)}$ is:

$$-D_{KL}(q(z|x) \parallel p(z)) \\ = \frac{1}{2} \sum_{i=1}^{z_{dim}} (1 + \log(\sigma_i)^2 - (\mu_i)^2 - (\sigma_i)^2) \qquad (3.3)$$

4. Circuit Structure and Operation

For implementation, we used the Zynq UltraScale+ MPSoC ZCU104[1] provided by Xilinx. Table 4.1 shows the specifications and development environment of the PL (Programmable Logic) part. In addition to the PL part, which rewrites circuits in HDL, this evaluation board has a PS (Processing System) part, which has an ARM CPU core. Table 4.2. shows the specifications and development environment. In this design, the Python library associated with PYNQ (Python Productivity for Zynq)[2] is used for FPGA rewriting and communication between the CPU and FPGA.

Table 4.1. PL Part Specification and Development Environment

Logic Cell	504,000
FF (Flip-Flop)	460,800
LUT (Look Up Table)	230,400
Block RAM	11 [Mb]
DSP Slice	1,728
Development	Vivado 2024.1

Table 4.2. PS Part Specification and Development Environment

CPU	ARM Cortex-A53
Memory	2.0 [GB]
OS	PynqLinux, based on Ubuntu 22.04
Compiler	g++ 11.2.0
Language	Python version:3.10.4

In this design, among the neural network training and point cloud generation, the circuit for point cloud generation is implemented in FPGA. The input values, weights, and bias values are obtained by training with an emulator.

The flow of the entire system is as follows.

1. The latent variable data is sent from the PS part to the PL part.
2. The start computation signal is sent from the PS part to the PL part.
3. The PL part receives a start computation signal and begin computation.
4. The PS part receives a signal from the PL part to end the computation.
5. The PS part sends a signal to receive the output value.

Fig. 4.1. shows an overall view of the configured circuit. The circuit is divided into two parts. AXI (Advanced eXtensible Interface) communication, and the calculation circuit, which controls data and performs decoder operations.

4.1. AXI Communication Circuit

For the communication of this system, we used the AXI4-Lite standard. Table 4.3. shows the addresses and control signals of AXI4-Lite.

Table 4.3. AXI4-Lite Addresses and Control Signals

Address	Control Signals
x00	Reset Signal (1bit) Start Computation Signal (1bit)
x04	End Computation Signal (1bit)
x08	Send Data to FIFO A (32bit)
x0C	Receive Data from FIFO B (32bit)

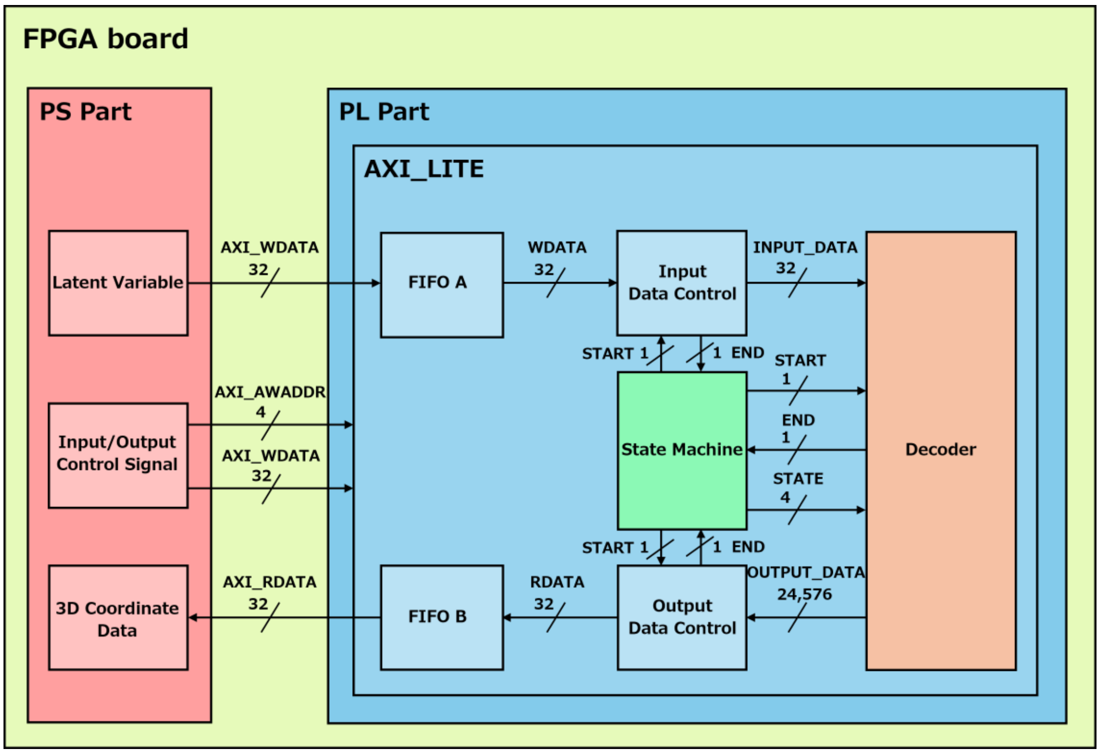


Fig. 4.1. Overall View of the Configured Circuit

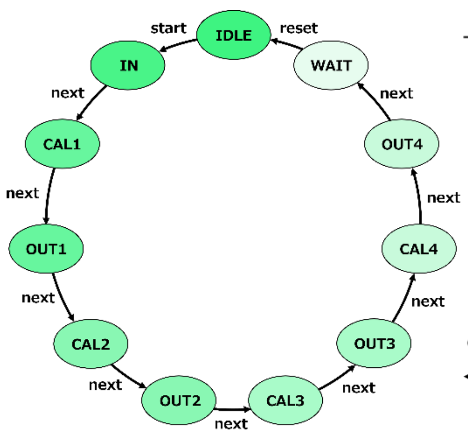


Fig. 4.2. View of the State Machine

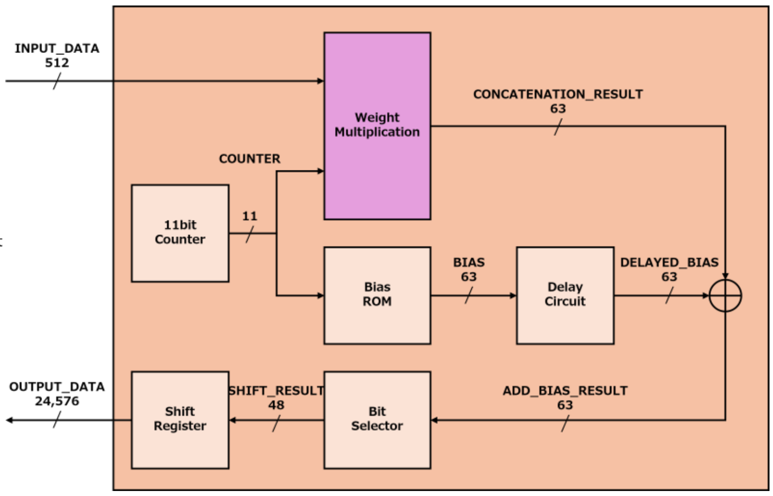


Fig. 4.3. View of the Decoder Circuit

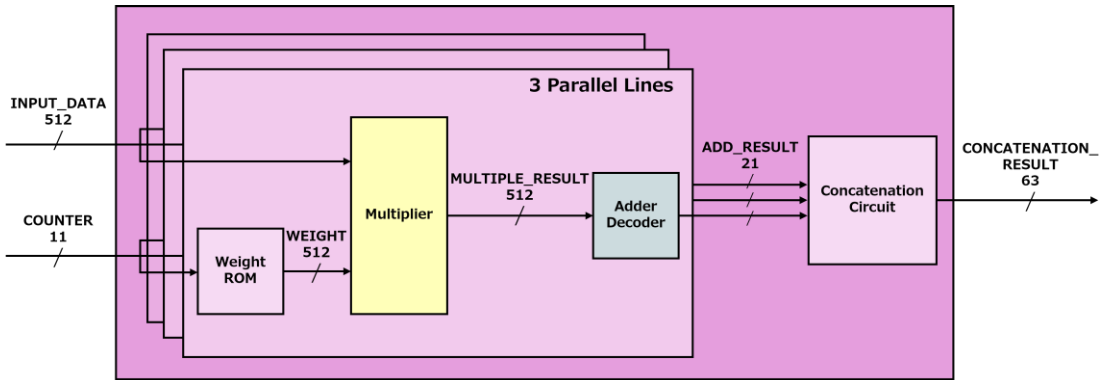


Fig. 4.4. View of the Weight Multiplication Circuit

In this system, the latent variables are 32-dimensional, and each dimension has 16 bits. Therefore, two 16-bit signals are connected and converted into a signal 32-bit signal in the PS part. FIFO A, which holds the input values to the calculation circuit, has a bit width of 32 bits and a bit depth of 16, and FIFO B, which holds the output values from the calculation circuit, has a bit width of 32 bits and a bit depth of 3,072 (=2,048 points × 3 × 16 bits ÷ 32)the final output value is 16 bits per dimension, the signal received from the PL part is divided in the PS part.

4.2. Calculation Circuit

The calculation circuit consists of four circuits: "Input Data Control", "Output Data Control", "State Machine", and "Decoder".

In "Input Data Control", the input data are combined because the decoder cannot perform operations unless all 512 (=32 × 16) bits of data are available.

In "Output Data Control", the 24,576-bit data which output from the decoder circuit is divided into 32 bits to enable data communication between PL and PS part.

In "State Machine", "Decoder" and "Output Data Control" are divided into four times (to reduce the number of wires. The details will be explained in §5.1), and it is responsible for managing their states.

In “Decoder Circuit”, it reconstructs the input 32-dimensional latent variables into a 6,144-dimensional output layer of 2,048 3D spatial coordinates (x, y, z). Fig. 4.3. shows view of the Decoder Circuit. The Decoder Circuit receives 512 bits of data, and it is multiplied with the corresponding weights data in the Weight Multiplication Circuit. The weight data used in this process is extracted from the Weights ROM. The sum of the multiplication results of 32 dimensions is calculated and output from the Weight Multiplication Circuit. The output data from the Weight Multiplication Circuit is added to the corresponding bias data. As with the weights data, the bias data is extracted from the Weights ROM and delayed matching the timing of addition. The data is concatenated in the Shift Register Circuit and output as 24,576-bit OUTPUT_DATA. 24,576 bits is the 3D data for 512 points (512 points × 3 dimensions × 16 bits). By performing the above process 4 times, the output layer of 6,144 dimensions, which is the 3-dimensional data for 2,048 points, is reconstructed.

4.2.1. Weight Multiplication Circuit

This section describes Weight Multiple Circuit in detail. The Weight Multiplication Circuit performs multiplication and addition of the input 32-dimensional data and the corresponding weights. Fig.4.4. shows the view of the Weight Multiplication Circuit. The Multiplier Circuit multiplies the input 32-dimensional data with the corresponding weights. The weight data is retrieved from ROM. The weight data stored in one address of the weight ROM is 512 bits of data for 32 dimensions with 16 bits per dimension. The input data and weight data are multiplied by 16 bits, resulting in a 32-bit multiplication result, of which the appropriate 16 bits are selected for the multiplication result in consideration of the addition process. The result of the multiplication of 32 dimensions is concatenated and output as 512 bits. Adder Decoder Circuit calculates the sum of the 32-dimensional data output from the Multiplier Circuit. Since it adds 32 times, a maximum of 5 overflows occurs ($32 = 2^5$). To prevent overflow, the calculation is performed in 21 bits by inserting the upper 5 bits. The above processes are performed in three parallel operations. The three 21-bit data are concatenated in a Concatenation Circuit and output in 63 bits.

5. Design Innovation

5.1. Reducing the Number of Wires

In this design, input is 512 (=32 dimensions × 16 bits) bits, and the output is 98,304 (=2,048 points × 3 × 16 bits) bits. The neural network is shown in Fig. 5.1.

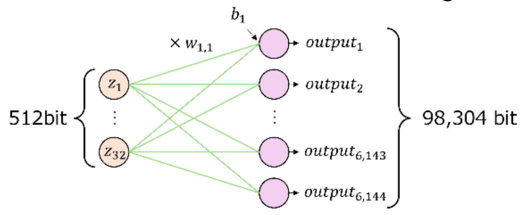


Fig. 5.1. Neural Network of Decoder Circuit

This circuit could not be implemented on the FPGA board due to the insufficient number of wires. Therefore, instead of obtaining all output values at once, we changed the specification to reduce the number of output wires by dividing the operation into multiple times. The neural network shown in Fig. 5.1. can be represented by the matrix operation shown in Fig. 5.2.

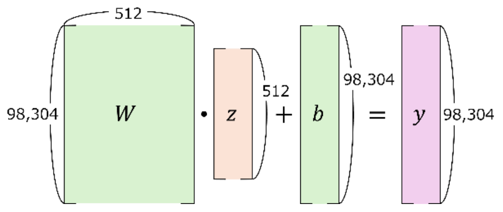


Fig. 5.2. Diagram of Matrix Operations in the Decoder Circuit

Here, we divide the matrix operation into 4 parts as shown in Fig. 5.3.

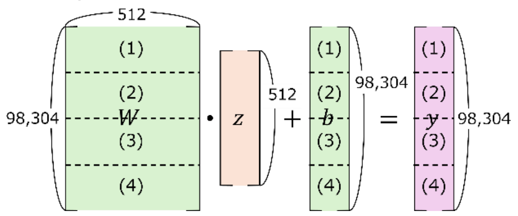


Fig. 5.3. Diagram of Fig. 5.2. Divided into 4 Parts

With this division, we can see that the output of (1) is obtained from the overall input value, the weights of (1), and the bias of (1). Therefore, to obtain the output of (1), we can design a circuit to compute the neural network shown in Fig. 5.4.

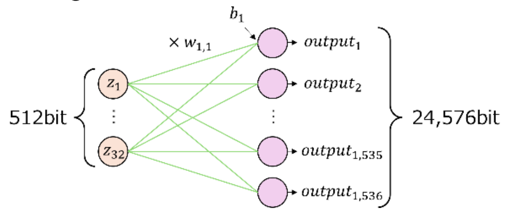


Fig. 5.4. Neural Network to Obtain (1) in Fig. 5.3.

By constructing the circuit that performs the calculation shown in Fig. 5.4. and updating the values of weights and biases, we were able to reduce the output wires to a quarter of the original circuit configuration, enabling implementation on the FPGA board.

5.2. Pipelining and Parallelization

In this design, the two operations, multiplication and addition, were parallelized, and the entire Decoder circuit was pipelined.

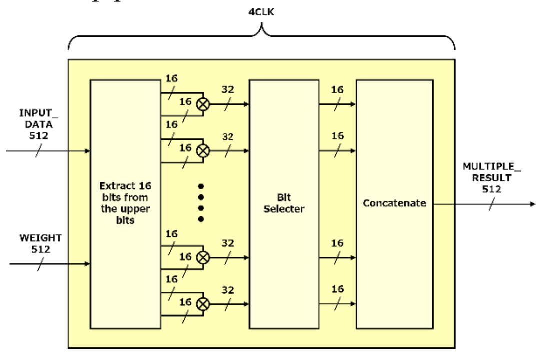


Fig. 5.5. View of the Multiplier Circuit

Fig. 5.5. shows the parallelization of the multiplier circuit. To calculate the one-dimensional data of the output layer, it is necessary to multiply the 32-dimensional data of latent variable data by weights and add biases. 32 multiplications are performed in parallel in one clock cycle by storing the weight data for 32 dimensions at one address in the Weight ROM.

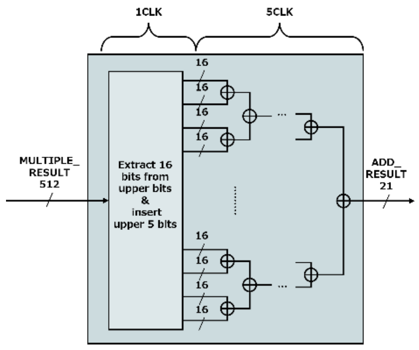


Fig. 5.6. View of the Adder Circuit

Fig. 5.6. shows the parallelization of the Adder Decoder Circuit. When all 32 multiplication results are added together, not added in sequence but instead are calculated in a tournament fashion. This makes it possible to perform the additional process in 5 clocks.

Furthermore, by processing the Multiplier Circuit and Adder Decoder Circuit in three parallel, it is possible to calculate the three dimensions of the output layer at once. Therefore, the calculation of 1,536 (=2,048 points × 3 dimensions ÷ 4) dimensions can be performed in 512 processes instead of 1,536 processes. Another advantage of the three parallel processing is that the 6,144-dimensional data in the output layer are the x-coordinate, y-coordinate, and z-coordinate of the first point of the point cloud data, starting from the first dimension, making it possible to obtain the three-dimensional coordinate data for one point in one calculation time.

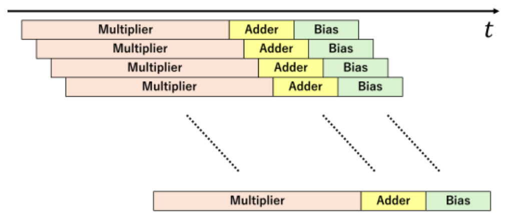


Fig. 5.7. Pipeline for Decoder

Fig. 5.7. shows how the entire Decoder Circuit is pipelined. The calculation required 17 clocks per 3 dimensions, and the calculation of the output layer of 1,536 dimensions required $(1{,}536 \div 3) \times 17 = 8{,}704$ clocks, but by making the entire Decoder Circuit pipeline-processable, processing can be done in $17 + (1{,}536 \div 3) = 529$ clocks.

6. Evaluation

6.1. Resource Usage

Table 6.1. shows the resource utilization and maximum frequency of the designed circuit. Note that Xilinx's Vivado2024.1. was used for logic synthesis and placement and routing.

Table 6.1. Resources and Maximum Frequency

Resources	Utilization	Utilization [%]
LUT	12,347	5.36
LUTRAM	118	0.12
FF	56,495	12.26
BRAM	113	36.22
DSP	96	5.56
BUFG	4	0.74
Maximum Frequency [MHz]	140.04	

6.2. Performance Comparison with CPU

We compare the results of computation time on an emulator running on a CPU with those on an FPGA. Table 6.2. shows the execution environment of the emulator. The emulator performs calculations in floating-point mode.

Table 6.2. Emulator Execution Environment

CPU	Intel Core i9-11900K 3.50GHz
Memory	64.0 [GB]
OS	Windows 10 Enterprise 64bit
Language	Python version:3.12.2

Table 6.3. shows the computation times of 1, 10 and 100 calculations performed by CPU and FPGA, where the process from the input of 32-dimensional latent variables to the output of 2,048 objects is 1 time. The computation time is the average of five measurements.

Table 6.3. CPU and FPGA Computation Time

Times	CPU [ms]	FPGA [ms]
1	67.9	25.8
10	645	254
100	6,327	2,540

Table 6.3. shows that the FPGA can compute about 2.5 times faster than the CPU. If we try to generate object points in real time under the present conditions, the frame rate is about 15fps (frame per seconds) for the CPU, while it is about 39fps for the FPGA. While 15fps makes the motion seem unnatural and slow, 39fps is smooth and practical.

7. Result

7.1. Generated Point Cloud Data

Fig. 7.1. shows the point cloud data obtained by inputting latent variables into the designed circuit and plotting them in a 3D coordinate system. The point cloud data in Fig. 7.1. (b), (c), and (d) are obtained latent variables by dividing the latent variables of a tadpole and an adult frog into two groups.

Fig. 7.1. (b), (c), and (d) show that the point cloud data interpolating the growth process from tadpoles to adult frogs can be generated.

7.2. Holographic Reconstruction with Generated data

We generated a CGH using the point cloud output from the designed circuit and conducted optical experiments. Fig. 7.2. shows the reconstruction 3D image obtained from the optical experiment. In this way, the point cloud data can be applied to holography.

8. Conclusion

As an application of VAE, we focused on “point cloud” and designed and implemented a point cloud data generation circuit that complements two point clouds. In designing the circuit, we considered how to process a huge amount of output data within a given circuit resource, and as a result, we were able to realize the desired system by employing a method of dividing outputs using a state machine. Furthermore, by implementing parallelization and pipelining in the decoder circuit, we were able to achieve a speedup of about 2.5 times faster than that of the CPU.

In the future, we would like to make further improvements aiming at generating reconstructed images with higher accuracy. In the circuit design, since the results showed that there is enough margin in the number of resources used, such as DSP, we aim to further increase the speed by introducing more parallelization of the operations that perform multiplication. In addition, to focus on the application to holography, by designing a circuit to create CGH from point cloud data, we want to create a circuit that works from point cloud generation to CGH creation in a single FPGA board.

Appendix

Transition from Tadpole to Frog: https://sites.google.com/view/ito-shimobaba-lab/conceal/lsi2025_frog

Reference and Links

[1] Zynq UltraScale+ MPSoC ZCU104 Evaluation Kit https://www.xilinx.com/products/boards-and-kits/zcu104.html

[2] PYNQ|Python Productivity to AMD Adaptive Coompute platforms http://www.pynq.io/

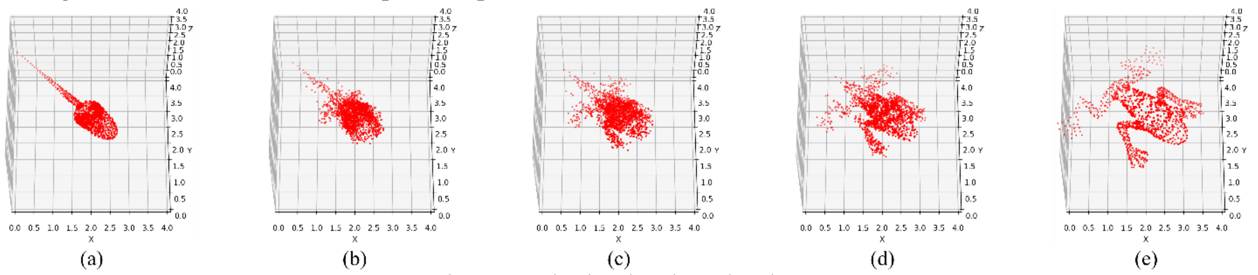


Fig. 7.1. Obtained Point Cloud Data

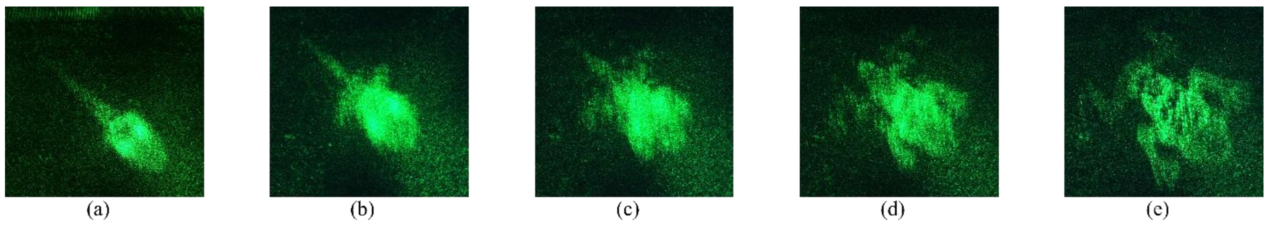


Fig. 7.2. 3D Image Obtained from the Optical Experiment

SVAE Implementation on FPGA for Arrhythmia Classification

Rafael Aditya Cahyo W, Ahmad Hafidz Aliim, Maritza Humaira
Electrical Engineering
School of Electrical Engineering and Informatics
Bandung, Indonesia
13221066@std.stei.itb.ac.id, 13221055@std.stei.itb.ac.id, 13221026@std.stei.itb.ac.id

***Abstract*—This research presents an implementation of a Stacked Variational Autoencoder (SVAE) on FPGA for arrhythmia classification using ECG signals. The design utilizes the PYNQ-Z1 board featuring a Xilinx Zynq-7000 SoC, employing a custom 16-bit fixed-point data format with 1-bit sign, 4-bit exponent, and 11-bit mantissa. The SVAE architecture consists of multiple stacked encoders and decoders, processing 10-dimensional input data through progressive dimensional transformations to a 1-dimensional latent space, ultimately providing binary classification for normal and arrhythmic patterns. The implementation achieves a maximum frequency of 100 MHz with 40 clock cycles per operation, resulting in an execution time of 400 ns and a throughput of 80 Mbps. Hardware utilization remains efficient at 4.46% LUTs, 1.88% Flip-Flops, and 8.64% DSPs. Testing on the MIT-BIH Arrhythmia Database demonstrated a classification accuracy of 91.5% across 200 samples, establishing the viability of FPGA-based SVAE implementation for real-time arrhythmia detection.**

***Keywords*—PYNQ-Z1, Arrythmia Classification, SVAE, hardware acceleration**

I. Introduction

Heart rhythm irregularities, known as cardiac arrhythmias, represent a serious medical problem that requires quick identification to ensure appropriate medical intervention. The primary tool for identifying these conditions is the electrocardiogram (ECG), which captures electrical signals from the heart. Although conventional ECG analysis typically relies on software systems, these approaches often struggle with speed limitations and challenges in real-time monitoring. Field Programmable Gate Arrays (FPGAs) have emerged as a promising hardware-based alternative, offering advantages through their ability to process multiple tasks simultaneously and adapt to different configurations. This potential has been demonstrated through practical applications, such as a successful FPGA implementation that extracts key features from ECG signals using time-domain analysis techniques [1], highlighting the practical benefits of hardware-based approaches in heart monitoring systems.

In recent years, machine learning techniques, especially artificial neural networks (ANNs), have been increasingly applied to ECG signal classification tasks. The parallel structure of ANNs aligns well with FPGA architectures, enabling efficient real-time processing. A study implemented a fully parallel ANN-based arrhythmia classifier on a single-chip FPGA, achieving an accuracy of 97.66% in classifying arrhythmias [2].

Building upon these successes with basic neural networks, researchers have begun exploring more sophisticated deep learning architectures for ECG analysis. Variational Autoencoders (VAEs), a class of generative models, have shown particular promise in learning latent representations of complex data distributions. The Stacked Variational Autoencoder (SVAE) extends this concept by incorporating stochastic processes, enhancing the model's ability to capture temporal dynamics inherent in sequential data like ECG signals. Given the demonstrated success of implementing neural networks on FPGAs, extending this hardware acceleration approach to SVAEs for arrhythmia classification could leverage the strengths of both advanced machine learning models and high-performance hardware architectures, potentially leading to more accurate and efficient diagnostic tools.

II. Related Works

Previous research has explored deep learning techniques for the classification of arrhythmias, emphasizing the effectiveness of residual and variational autoencoder (VAE) architectures. Lu et al. [3] investigated the use of deep networks to annotate arrhythmias, employing feature extraction through residual layers and latent representation through VAE, followed by classification with a bidirectional recurrent neural network. They used ECG data from the VFDB and MIT Normal Sinus Rhythm databases, which were preprocessed by resampling, removal of baseline drift, and conversion to 2D grayscale images. Similarly, Belen et al. [4] proposed an uncertainty estimation framework for the detection of atrial fibrillation using VAE to improve the reliability of classification. Their method involved residual feature extraction, transposed convolution for decoding, and confidence evaluation through repeated sampling of network outputs. These works highlight the potential of combining residual networks and VAEs in achieving a robust arrhythmia classification.

In another work, Fahad et al. [5] developed a deep learning model, Convolutional Neural Networks (CNN), to identify arrhythmias using ECG signals. Since the older arrhythmia detection methods are not completely accurate and reliable, the testified CNN model can successfully detect and distinguish arrhythmias with a testing accuracy of 98.3% and a validation accuracy of 83.7%, respectively. Zhong and Sun [6] proposed a personalized arrhythmia detection system PerDetect based on an unsupervised autoencoder. The system trains a separate BiLSTM-based autoencoder BiAE for each patient for arrhythmia detection. BiAE only needs to use the patient's normal heartbeat for training. Experiments show that the system only needs a small amount of ECG training data (within five minutes) to achieve good performance and the accuracy of their method on MIT-BIH Arrhythmia Database is 97%.

Nithya and Rani [7] have investigated the application of a Stacked Variational Autoencoder (SVAE) for the automatic diagnosis of arrhythmia from ECG signals. Furthermore, the augmented data set is used for training the model, to resolve the imbalance in the classes. The proposed model reached an overall accuracy of 98.96% and sensitivity of 97.32%. SVAE classified twelve classes of cardiac arrhythmia including normal sinus rhythm.

III. Proposed Design

A. Dataset

The model in this project is trained using the MIT-BIH Arrhythmia Database, a widely recognized benchmark for arrhythmia classification research [8]. The database contains 48 half-hour excerpts of two-channel ambulatory ECG recordings from 47 subjects, collected by the BIH Arrhythmia Laboratory between 1975 and 1979. The recordings were sampled at 360 Hz with 11-bit resolution, providing high-fidelity ECG signals suitable for detailed analysis. Each record is annotated by two or more cardiologists, resulting in approximately 110,000 beat annotations that serve as a reliable reference for model training.

This database was chosen for its extensive and diverse representation of arrhythmias, including clinically significant but rare cases that are crucial for training robust classification models. By providing a well-annotated and high-quality dataset, the MIT-BIH Arrhythmia Database enables the development of deep learning models capable of handling complex ECG patterns and ensuring accurate arrhythmia detection in real-world scenarios.

B. Data Preparation

The data preparation process for the arrhythmia classification model is using Jupyter Notebook computing program [9]. It involves four key stages: preliminaries, data visualization, preprocessing, and data balancing. In the preliminaries, essential libraries such as wfdb for ECG signal processing and imbalanced-learn for handling class imbalance were installed. Supporting tools like numpy, tensorflow, and matplotlib were also imported to enable numerical computations, model building, and data visualization.

Data visualization followed, where ECG signals were plotted to examine their structure and identify significant features. For instance, an ECG record was loaded, and 3,000 samples were visualized to understand the waveform's characteristics and identify R-peaks, as shown in Fig. 1. This step provided

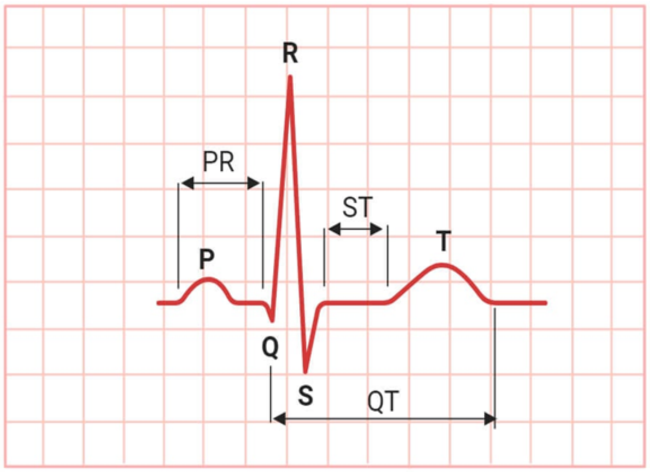


Fig. 1. ECG Measures Signal

insights into the dataset's composition and guided subsequent preprocessing tasks.

In data preprocessing, annotations from the ECG records were extracted to locate R-peaks and classify beats as normal or arrhythmic. R-R intervals were calculated and normalized, then segmented into windows of 10 beats. Each segment was labeled as normal (if all beats were normal) or arrhythmic (if at least two ventricular beats were present). This segmentation approach ensured the model received structured and meaningful input data.

Finally, data balancing addressed the inherent class imbalance, as normal beats outnumber arrhythmic ones. Using SMOTE (Synthetic Minority Oversampling Technique), additional synthetic samples of the minority class were generated, effectively equalizing the class distribution. This step improved the model's ability to learn from both normal and arrhythmic cases, enhancing its predictive accuracy. These meticulous preparations laid the groundwork for an effective arrhythmia classification model.

C. Data Type

In the proposed design, a custom data type was designed to balance precision and hardware efficiency, consisting of a 1-bit sign, a 4-bit exponent, and an 11-bit mantissa. This configuration is a tailored floating-point representation optimized for the specific requirements of the system.

The primary reason for adopting this custom data type is to improve computational accuracy without incurring excessive hardware overhead. The 11-bit mantissa provides sufficient precision to represent the fine-grained variations in the processed data, which is critical for tasks like arrhythmia classification that rely on subtle distinctions in ECG signals. Meanwhile, the 4-bit exponent is adequate because the range of values in the data is relatively limited, and the application does not require handling extremely large or small numbers. This keeps the hardware resources efficient, as fewer bits for the exponent reduce complexity while still ensuring the data range is fully covered. Additionally, the 1-bit sign allows for representation of both positive and negative values, which is essential for capturing the full spectrum of ECG signal amplitudes.

D. Model

To perform identification, the proposed design utilizes a Stacked Variational Autoencoder (SVAE). This type of Autoencoder is a Deep Generative Model used for learning latent representations, featuring multiple stacked encoders. The architecture of the implemented SVAE is shown in Fig. 2.

In the proposed SVAE model, different activation functions are used at various stages to enhance learning and stability. ReLU is used in the first and third encoder layers, where the representation size is reduced, promoting sparsity and efficient representation learning by setting negative values to zero, which helps filter out less significant features. These layers are described by Eq. 1. Meanwhile, Softplus is used to the variance in second encoder (latent space), to ensure numerical stability by preventing negative values while maintaining sensitivity to small variations. This layer is described by Eq. 2.

$$\mathbf{y_1} = \text{ReLU}(\mathbf{W_1 x} + \mathbf{b_1}) \tag{1}$$

$$\mathbf{y}_\sigma = \text{Softplus}(\sqrt{\sigma_\mathbf{i}{}^2} \times \epsilon_i) \tag{2}$$

With

$\mathbf{W}$ = Weight
$\mathbf{b}$ = Bias
$\mathbf{y}$ = Encoder Output
σ = Variance
ϵ = Random variable

The proposed Stacked Variational Autoencoder (SVAE) architecture employs a series of stacked encoders and decoders to effectively process and classify arrhythmia patterns. The encoder section consists of multiple dense layers that progressively transform the 10-dimensional input data through a sequence of dimensional transformations, ultimately compressing it into a 1-dimensional latent space representation. This progressive compression through stacked layers, rather than direct dimensional reduction, is crucial for preserving important feature information and preventing underfitting. The latent space serves as a compact yet informative representation where different arrhythmia classes can be effectively separated into distinct regions. Following the latent space, the decoder section mirrors this structure in reverse, gradually expanding the dimensionality through dense layers until reaching the final 2-dimensional output layer for classification purposes.

This architecture culminates in a two-dimensional output layer that functions as a confidence-based classifier. This output provides probability scores indicating the likelihood of whether the analyzed heartbeat belongs to either the normal or arrhythmic category. The relative magnitude of these output values serves as a confidence measure - a significantly higher value in one output node compared to the other strongly indicates the classification of the heartbeat pattern. This probabilistic approach allows the model to not only classify the heartbeats but also quantify the confidence level of its classification decision, providing valuable diagnostic information for clinical interpretation.

E. Hardware Implementation

The hardware implementation of a Stacked Variational Autoencoder (SVAE) for arrhythmia classification utilized the PYNQ-Z1 board. The PYNQ-Z1 is a development platform designed around the Xilinx Zynq-7000 SoC, which integrates a dual-core ARM Cortex-A9 processor with programmable logic based on the Artix-7 FPGA architecture. This integration allows for efficient hardware-software co-design and execution of computationally intensive tasks like deep learning model inference.

The PYNQ-Z1 provides 13,300 logic slices, 220 DSP slices for high-speed arithmetic operations, and 630 KB of block RAM for efficient data handling. Its 512 MB DDR3 memory operates with a 16-bit bus at 525 MHz, ensuring sufficient bandwidth for managing large datasets. Additionally, the board is equipped with multiple I/O features, such as HDMI, Ethernet, and USB, enabling real-time data streaming and external device connectivity [10].

A key feature of the PYNQ-Z1 is its support for the PYNQ framework, which simplifies programming by enabling the use of Python for hardware control and interaction. This makes it an accessible platform for deploying advanced machine learning models while leveraging its robust hardware capabilities. The board's energy-efficient design, coupled with its ability to handle parallel processing tasks in the FPGA fabric, makes it highly suitable for biomedical applications, including real-time arrhythmia detection systems.

IV. Implementation

A. Implementation Steps

The FPGA implementation of the SVAE design for arrhythmia classification was conducted using the AXI Lite interface, with Vivado serving as the primary tool for hardware generation. The implementation process involved generating essential files, including Hardware Handoff File (`.hwh`), Tool Command Language Script (`.tcl`), and Bitstream File (`.bit`), which were subsequently deployed to the FPGA board via a wireless local area network. The design was further integrated and tested within a Jupyter Notebook environment. To ensure compatibility with the fixed-point architecture, a fixed-point translator was incorporated into the source code, along with prepared test datasets derived from the data preprocessing stage. This methodology ensured a seamless transition from design to deployment, facilitating efficient validation of the proposed architecture.

B. Synthesis Result

From the synthesis performed in Vivado using the PYNQ-Z1 board, we obtained utilization results for LUT, Flip Flop, and DSP as shown in Table I.

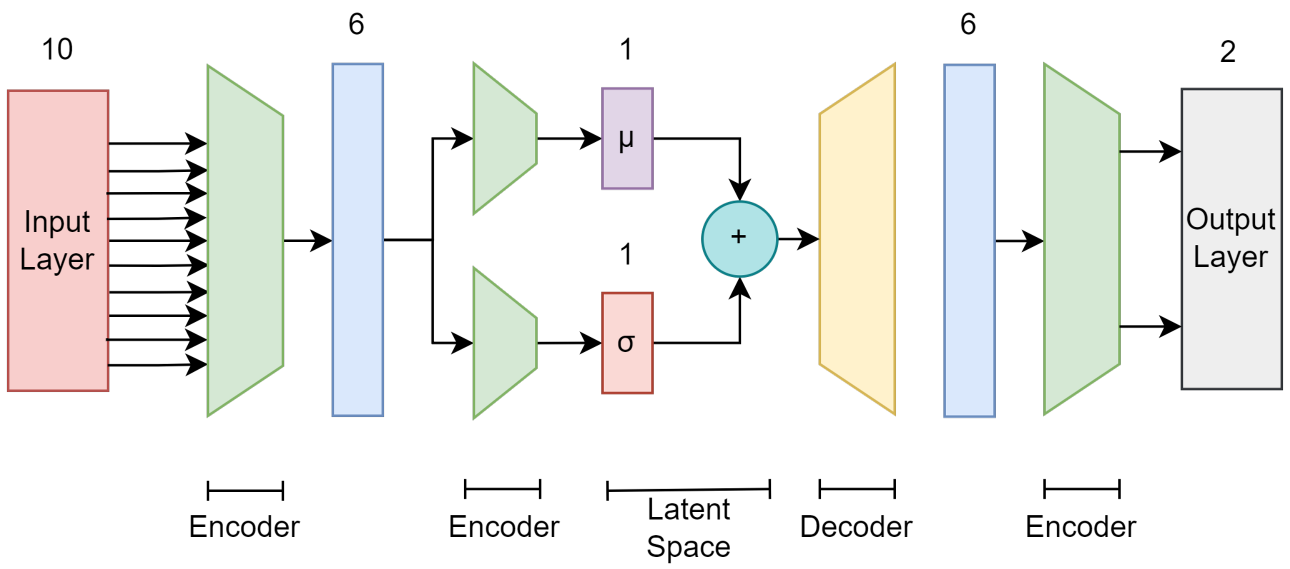


Fig. 2. Architecture of Stacked-VAE

TABLE I
SYNTHESIS RESOURCES RESULT

Parameters	Used	Utilization
LUT	2375	4.46%
FF	2001	1.88%
DSP	19	8.64%

TABLE II
IMPLEMENTATION PROCESS CLOCK CYCLE

Process	Clock Cycle
Encoder 1	12
Encoder 2	8
Lambda	8
Decoder	3
Encoder 3	8
Offset	1
Total	**40**

TABLE III
SYNTHESIS PERFORMANCES RESULT

Parameters	Value
#Clock Cycle	40
Frequency	100 MHz
Excecution Time	400 ns
Throughput	80 Mbps
Power	1.433 W

Performance measurements of the design were conducted. The synthesis results show that the design has a maximum frequency of 100 MHz. The number of clock cycles required was calculated for each progress step, resulting in a total of 40 clock cycles as shown in Table II. The value of maximum frequency 100 MHz is obtained by assigning the value of clock cycle period in the constraint file (`.xdc`). This process goes through several “trials and errors” to acquire the least value of period. From these two data points, execution time and throughput can be calculated using Eq. 3 and Eq. 4.

$$Execution\ Time = \frac{\#CCs}{f_{max}(MHz)} \tag{3}$$

$$Throughput = \frac{\#bits}{Execution\ Time} \tag{4}$$

As previously discussed, the output of this design consists of two confidence values with a 16-bit fixed-point format, making the total output size 32-bit. The performance results obtained are presented in Table III.

C. Implementation on FPGA

The implementation on the FPGA uses a Jupyter Notebook. This notebook includes importing the overlay, a custom fixed-point translator function, and several pieces of data from the dataset. Jupyter Notebook was also used to perform comparisons with the dataset and display the output of arrhythmia detection, which is located on the board’s Processing System (PS). Meanwhile, the arrhythmia detection model was implemented on the board’s Programmable Logic (PL). This implementation is shown in Fig. 3.

Testing on the FPGA produced 17 prediction errors out of 200 samples, demonstrating a model accuracy of approximately 91.5%, as shown in Fig. 4.

V. CONCLUSION

The implementation of the Stacked Variational Autoencoder (SVAE) on FPGA for arrhythmia classification demonstrates

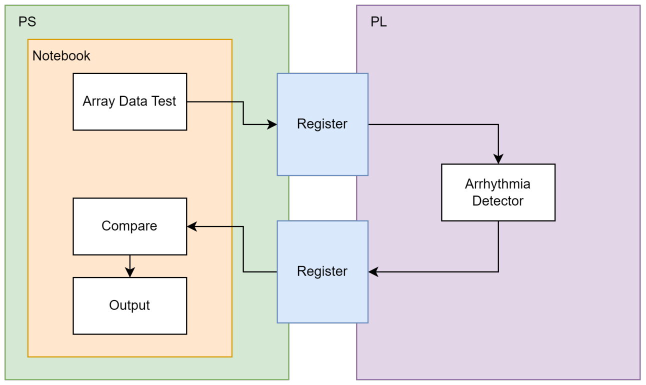
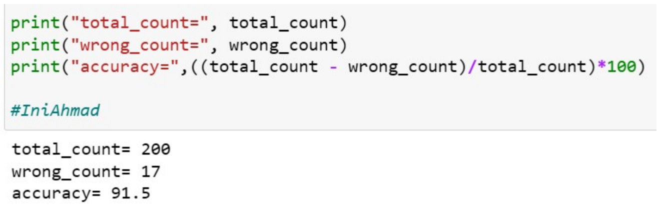


Fig. 3. Diagram of FPGA Implementation



```
print("total_count=", total_count)
print("wrong_count=", wrong_count)
print("accuracy=",((total_count - wrong_count)/total_count)*100)

#IniAhmad
```

```
total_count= 200
wrong_count= 17
accuracy= 91.5
```

Fig. 4. Accuracy of FPGA Implementation

promising results in terms of hardware efficiency and processing speed. The design achieves significant performance metrics with a maximum frequency of 100 MHz and throughput of 80 Mbps, while maintaining modest resource utilization on the PYNQ-Z1 board. The custom 16-bit fixed-point data format proves sufficient for maintaining computational accuracy while optimizing hardware resources. While the achieved accuracy of 91.5% in real-world testing indicates room for improvement, it establishes the feasibility of implementing complex deep learning architectures like SVAE on FPGA platforms for real-time arrhythmia detection.

VI. Future Works

Future work could focus on optimizing the architecture to enhance classification accuracy while maintaining efficient hardware utilization and high-speed processing. Currently, the proposed model can classify only two types of arrhythmias: normal and Premature Ventricular Contraction (PVC). Expanding the model to classify a broader range of arrhythmia types would improve its clinical relevance and practical application.

Additionally, a fully integrated system could be developed to enable end-to-end real-time arrhythmia detection. This system would incorporate an ECG sensor for data acquisition, followed by analog-to-digital conversion. The pre-processing stage, handled by the Processing System (PS) of the FPGA, would extract key features such as the timing of ten peak values and normalize the input data. The processed signals would then be sent to the Programmable Logic (PL), where the trained SVAE model would perform classification. The output could be presented as a real-time indicator or logged for further analysis, facilitating improved monitoring and diagnosis.

References

[1] N. Madiraju, N. Kurella, and R. Valupadasu, "Fpga implementation of ecg feature extraction using time domain analysis," 02 2018.

[2] K. Danisman, "Fully parallel ann-based arrhythmia classifier on a single-chip fpga: Fpaac," *Turkish Journal of Electrical Engineering and Computer Sciences*, vol. 19, pp. 667–687, 01 2011.

[3] W. Lu, J. Shuai, S. Gu, and J. Xue, "Method to annotate arrhythmias by deep network," in *2018 IEEE International Conference on Internet of Things (iThings) and IEEE Green Computing and Communications (GreenCom) and IEEE Cyber, Physical and Social Computing (CPSCom) and IEEE Smart Data (SmartData)*, 2018, pp. 1848–1851.

[4] J. Belen, S. Mousavi, A. Shamsoshoara, and F. Afghah, "An uncertainty estimation framework for risk assessment in deep learning-based afib classification," in *2020 54th Asilomar Conference on Signals, Systems, and Computers*, 2020, pp. 960–964.

[5] R. A. Mohammed Fahad, S. Purohit, S. Bhatnagar, D. M. Kotambkar, V. M. Thakre, and K. Kapoor, "Development of deep learning model for cardiac arrhythmia detection from ecg," in *2024 IEEE Recent Advances in Intelligent Computational Systems (RAICS)*, 2024, pp. 1–6.

[6] Z. Zhong and L. Sun, "Perdetect: A personalized arrhythmia detection system based on unsupervised autoencoder," in *2023 7th Asian Conference on Artificial Intelligence Technology (ACAIT)*, 2023, pp. 914–919.

[7] S. Nithya and M. M. S. Rani, "Stacked variational autoencoder in the classification of cardiac arrhythmia using ecg signals with 2d-ecg images," in *2022 International Conference on Intelligent Innovations in Engineering and Technology (ICIIET)*, 2022, pp. 222–226.

[8] G. Moody and R. Mark, "Mit-bih arrhythmia database v1.0.0," https://physionet.org/content/mitdb/1.0.0/, Feb 2005, accessed: 2024-02-24.

[9] T. Fu, "Github - tonyfu97/tinyrhythmanalyzer: Tinyml-powered ecg arrhythmia detection on arduino nano 33 ble sense, comes with a video tutorial," https://github.com/tonyfu97/TinyRhythmAnalyzer, 2023, accessed: 2025-01-28.

[10] S. Bobrowicz, "Pynq-z1 reference manual - digilent reference," https://digilent.com/reference/programmable-logic/pynq-z1/reference-manual, 2016, accessed: 2025-01-28.

Design and Implementation of Angle Completion Circuit Using VAE

Makoto Sekiguchi, Shun Nishijima, Yu Yamazawa, Soma Kato

Graduate school of Engineering, Chiba University

Chiba, Japan

Mail:24wm4307@student.gs.chiba-u.jp

Abstract- **We designed a circuit for VAE that generates images from unknown angles by learning images obtained from multiple angles and built a system on FPGA that outputs the images. We also achieved a circuit size that could fit on an FPGA evaluation board and a faster computation processing time.**

Keywords-Variational Autoencoder, FPGA, Pipelining,

Ⅰ. INTRODUCTION

The demand for 3D model creation has been steadily increasing in recent years, yet the production process remains complex and time-consuming. Efficient production methods are essential for the rapid generation of 3D models that support critical applications across healthcare, industrial design, and entertainment sectors. While 3D models enable visual communication and analysis in various fields, traditional creation methods require specialized expertise and considerable time investment, limiting their widespread application.

In response to these challenges, we aimed to design hardware capable of predicting and generating complete 3D models from a limited number of 2D images using VAE. This concept formed the foundation of our LSI design contest entry. By developing this hardware system, we sought to address the human resource and time constraints associated with 3D modeling by enabling visual generation of complete 3D models from images captured from limited viewpoints.

The system operates as follows: When images of an object captured from multiple angles are input into the system, a two-layer neural network (encoder) computes the latent space representation of the image set. By extracting latent variables from this space and inputting them into a two-layer decoder, the system can generate images of the object from arbitrary viewing angles. These output images can then be integrated to produce a complete 3D model in a short time. This technology enables anyone to create 3D models easily without specialized knowledge, significantly expanding accessibility and practical applications across various fields.

Ⅱ. LEARNING METHOD

An autoencoder consists of an encoder component that extracts essential features from input data and converts them into a low-dimensional latent representation, and a decoder component that reconstructs the original input data based on this latent representation. This architecture enables data compression and restoration, facilitating feature extraction and dimensionality reduction.

Figure 1 illustrates an example of an autoencoder model comprising three layers: an input layer, a hidden layer, and an output layer. In this example, both the input and output layers contain n units, while the hidden layer contains m units. The input and output layers of the autoencoder are defined by Equations (1) and (2), respectively:

$$x = (x_1, x_2, \dots, x_n)^T \qquad (1)$$

$$o = (o_1, o_2, \dots o_n)^T \qquad (2)$$

where x(n) represents the *nn* n-th input and x^(n) represents the *nn* n-th output value. The model depicted in Figure 1 adheres to Equations (3.1.3) and (3.1.4):

$$h = f(Wx + b) \qquad (3.1.3)$$

$$o = f(\tilde{W}h + \tilde{b}) \qquad (3.1.4)$$

where *WeW_e* We denotes the encoder weight matrix, *beb_e* be denotes the encoder bias vector, *WdW_d* Wd denotes the decoder weight matrix, *bdb_d* bd denotes the decoder bias vector, *h(n)h^{(n)}* h(n) represents the hidden layer output (latent representation), and *ff* f and *gg* g represent activation functions.

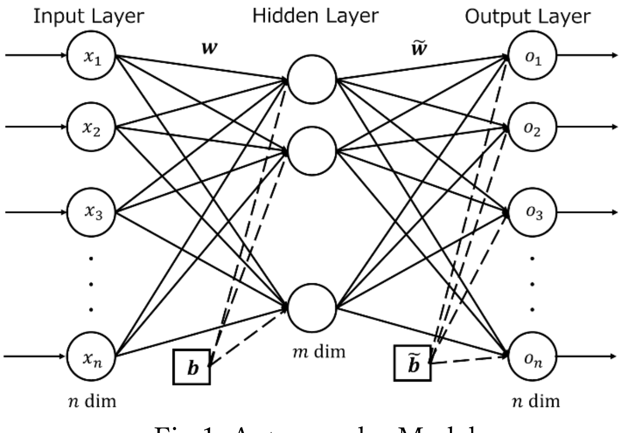


Fig.1. Autoencoder Model

Activation functions are mathematical operations that introduce non-linearity into neural networks, enabling them to learn complex relationships in data.

The Variational Autoencoder (VAE) represents an advancement over traditional autoencoders by integrating a probabilistic approach into the autoencoder framework, allowing it to learn the underlying structure of data and generate new instances. While conventional autoencoders output latent variables as discrete and fixed representations, VAEs encode latent variables as continuous and probabilistic distributions. This capability enables VAEs not only to reconstruct the original input but also to generate novel data similar to the input by manipulating variables within the latent space.

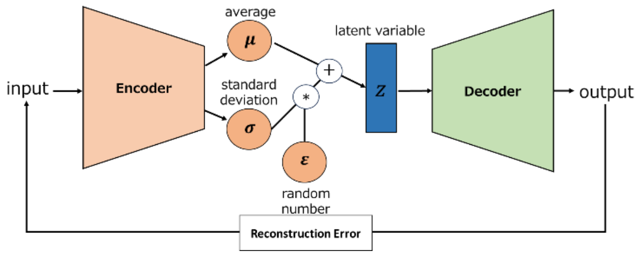


Fig.2. shows a rough outline of the VAE model.

Our system utilizes datasets comprising multiple viewpoints of arbitrary objects to train VAEs and derive the latent space. By inputting interpolated values between the latent vectors of any two viewpoints into the decoder, the system can generate images from previously unseen viewpoints that are not present in the original dataset. The implementation of such view synthesis requires datasets containing images of objects captured from multiple angles. Therefore, prior to model training, we created a comprehensive multi-view image dataset of the objects. The specific methodology is described below:

1. Record video footage of the subject from a fixed direction for a predetermined duration
2. Repeat this procedure three times (from three different angles)
3. Extract individual frames from the recorded videos using the OpenCV library in Python
4. Crop the extracted images to form perfect squares
5. Resize the images to the desired dimensions using OpenCV to create the dataset

The images used in this study are grayscale with 256 intensity levels and a relatively low resolution of 32×32 pixels. To achieve high-precision view synthesis, the training process must be highly accurate; consequently, we prepared 2,048 images for each viewing direction. The VAE was trained on this dataset using images of a particular mug as the subject. All components related to VAE training were implemented in Python. Table 1 presents the various parameters of the VAE designed specifically for this system.

Table. 1. Each parameter in learning

Epoch	30
Batch size	32
Activation Function	ReLU Function
Error function	KL divergence
Optimization Method	Adam
Learning start weight	All random
Learning start bias	All 0

Ⅲ. CIRCUIT DESIGN

The FPGA evaluation board used for implementation was a Zynq FPGA from AMD-XilinX. The evaluation board specifications and development environment are shown in Table 0. This evaluation board contains a SoC in addition to the FPGA. Table 2 shows the specifications of the PS part of the SoC.

Table. 2. FPGA evaluation board specifications

FPGA	Zynq UltraScale+ MPSoC ZCU104
Logic cell	504000
Block RAM[Mb]	11
DSPSlice	1728
Communication Protocol	AXI4-Lite
Development environment	Vivado 2023.2

Table. 3. Specifications of PS section

CPU	Cortex-A53
Memory [Gb]	2.0
OS	Pynq Linux based on Ubuntu 22.04
Compiler	g++ version
Development environment	Jupyter Notebook version 2024.11.0

The protocol used for communication between the PS and PL sections is AXI4 (Advanced eXtensible Interface). This system uses the AXI4-Lite type, which is easier to control than the full version of AXI4.

The two latent variables input to the decoder are 16 bits each, so the 32-bit input from the PS section is directly input to the main circuit (decoder) in this system. The output from the decoder is a 32 x 32 pixel image, with each pixel being 16 bits. The output is formatted to 32 bits in width in the data controller and returned to the PS section through the FIFO.

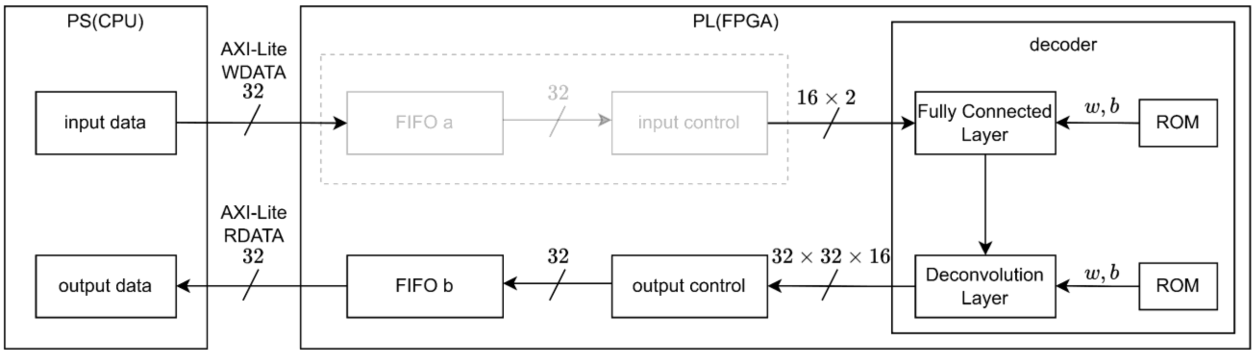


Fig. 3. Overall view of the system

The processing steps for image generation are as follows.

(1) The latent variable input from the CPU is sent to the decoder.

(2) The bias and weight data stored in the ROM and the data in (1) are all combined.

(3) Perform inverse convolution on the bias and weight data stored in the ROM and the data obtained in (2).

(4) The data obtained in (3) is sent to the CPU.

Processes (2) and (3) are explained below.

The all-combining layer, which processes (2), combines the data information and outputs new data. In this system, the data sent from the CPU is multiplied by the weight data and bias data stored in ROM. The weight data is used to transform each piece of data, and the bias data is used to bias all the data in the same direction. The formula is shown in Equation (3).

$$A = z_1 w_{11} + z_2 w_{12} + b_1 \tag{3}$$

The output result z_1 is the lower 16 bits of the input data, z_2 is the upper 16 bits of the input data, w_11 is the lower 16 bits of the weight data, w_12 is the upper 16 bits of the weight data, and b_1 is the bias data. This allows arbitrary transformation of the input data.

Before performing this calculation, a function is defined to multiply two 16-bit fixed-point data, and this function is used for subsequent multiplications. Since the data width of the input and weight data is 32 bits and the data width of the bias data is 16 bits, the calculation was established by dividing the input and weight data in two.

The circuit diagram is shown in Figure 3.

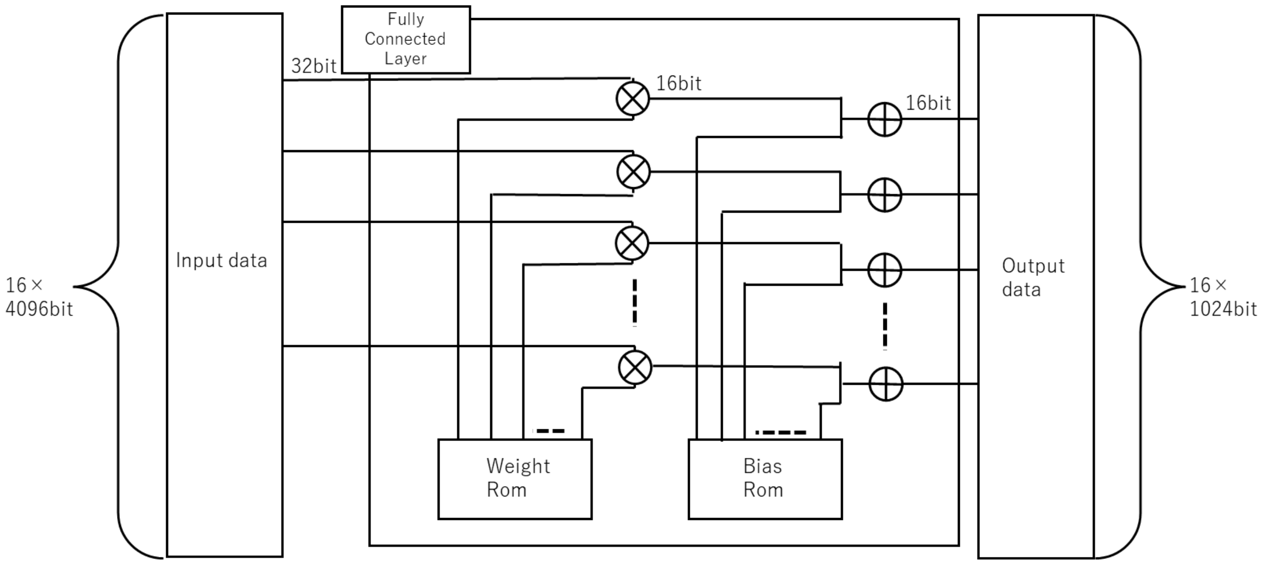


Fig. 4. Circuit diagram of all coupled layers

In (3), the inverse convolution layer creates kernel data by using weight data from ROM and performs inverse convolution processing on the results obtained in the all-combining layer using the kernel data. This generates a high-resolution feature map from a low-resolution feature map.

The calculation procedure is described below. The weight data for the inverse convolution has 144 bits, which are divided into nine data sets of 16 bits each and treated as a kernel matrix. Then, as shown in Figure 5, the product of A and each kernel matrix obtained in Equation (3) is calculated. This process is then performed on the array to the right of the first one. The result is then stored in the array two arrays to the right of the output data. If the previous data already exists, it is added to the output data. This process is performed on all 256 cells of data A to complete the convolution process. The output data is then combined into a single array and sent to the CPU. The data to be sent is a 16384-bit array.

The schematic of these calculations is shown in Figure 4.

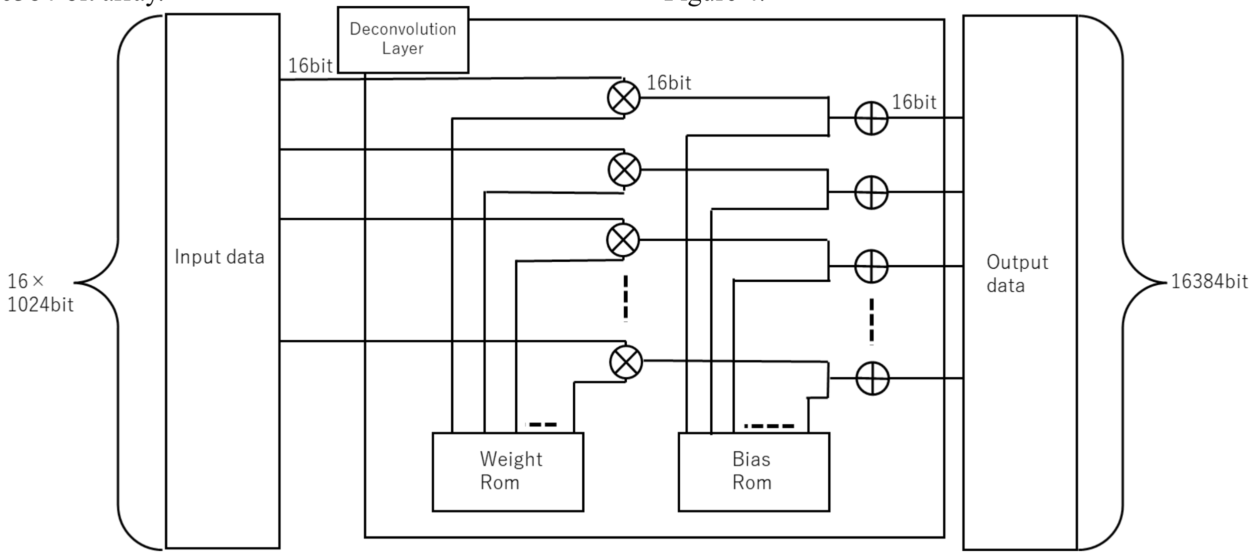


Fig. 5. Reverse convolution layer schematic

Pipelining is a method of reducing computation time by performing multiple instructions in parallel. In this system, five instructions are processed in parallel when performing inverse convolution processing.

When each of the following operations (1) to (7) is performed in one clock cycle, the pipeline processing operation is as shown in Figure 5.

(1) Reading weight and bias data
(2) Multiply input data and weight data
(3) Addition of bias data and the result of (2)
(4) Creation of kernel
(5) Multiply kernel and (3)
(6) Add (5) to the array for storage
(7) Store output data
(8) Output data molding
(9) Send end signal

The actual number of clocks required for this system is shown in Table 4.

Of these processes, pipeline processing is performed for steps (2) through (7). The number of clocks is about 256 times larger than that without pipeline processing because 256 (16 × 16) data are processed in parallel. Therefore, this process succeeds in greatly reducing the number of clocks.

However, this pipeline processing is controlled using multiple counts and requires accurate timing of operations, which is considered to be a high degree of difficulty. This time, the timing was calculated in detail before implementation.

Table. 4. Number of clocks per operation

	Processing	Clock
①	Read weight and bias data	1
②	Multiply input data and weight data	1
③	Addition of bias data and result of ①	1
④	Create kernel	1
⑤	Multiply kernel and ③	1
⑥	Add ⑤ to the array for storage	256
⑦	Store output data	1
⑧	Output data forming	1
⑨	Send end signal	1
Total		264

Fig. 6. also shows an image of the pipeline process.

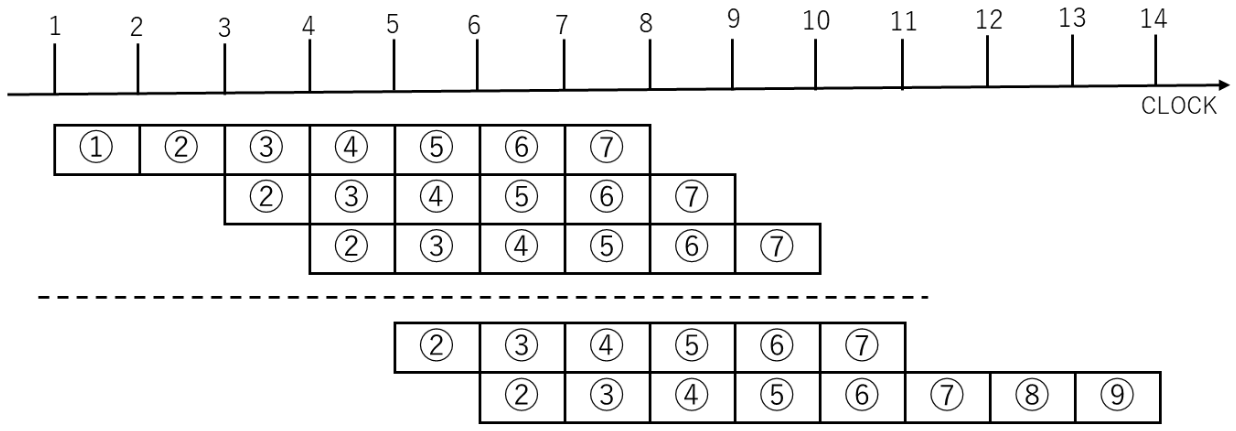


Fig.6: Diagram of pipeline processing

Ⅳ. SYSTEM EVALUATION

Table 5 shows the circuit scale of the designed circuit. The operating frequency of the FPGA was set at 100 MHz. A comparison of processing speeds on the actual device is also shown in Table 6.

Table. 5. Resources Used

Resource	Utilization	Utilization[%]
LUT	26494	11.50
LUTRAM	101760	0.01
FF	460800	3.81
BRAM	312	3.85
DSP	1728	0.64
BUFG	544	0.18

Table. 6. Comparison of calculation processing speed

Environment	Computation time [μs]
PC(Python)	139507
Simulation	97.34

As shown in the table, the speedup is approximately 1,400 times faster than when executed on a PC. Although a simple comparison cannot be made because of the different languages and implementations, it is believed that the speedup was achieved by using pipelining and other speedup techniques.

Ⅴ. OUTPUT RESULTS

Figure 7 and Figures 8 and 9 show the input and output images (left: input, right: output) generated by the software (Python). The output image is the input latent vector, which is the average value of the labeled data set. It can be seen that the output image is close to the input image. However, if we focus on the details, we can see that the contours are blurred. This can be attributed to the loss of contour information in the compression of encoder features, which is a characteristic of VAE.

Figure 10 shows the output images when the average values of the latent vectors of the left and center viewpoints are input and when the average values of the latent vectors of the center and right viewpoints are input. The image on the left side of Figure 10 is expected to look like the image seen from between Figure 7 and Figure 8, but the image is actually closer to the image seen from slightly left of center. Similarly, the image on the right side of Figure 10 is similar to the image viewed from slightly to the right of the center.

This confirms that the decoder can achieve the original goal of viewpoint completion by inputting the average value of the latent vectors for each viewpoint.

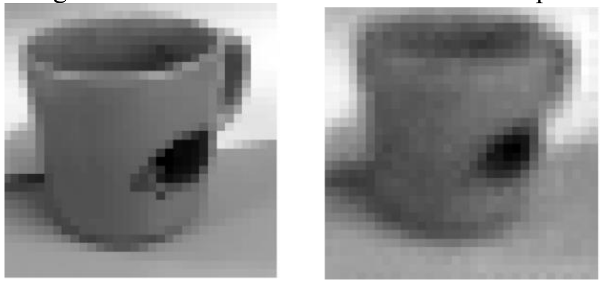


Fig. 7. Software input/output image (left side viewpoint)

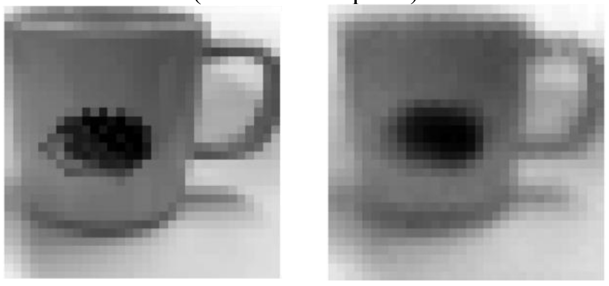


Fig. 8. Software input/output image (center viewpoint)

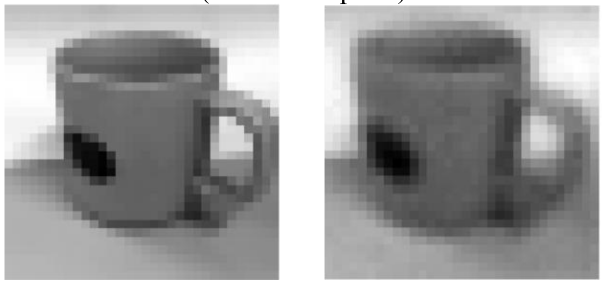


Fig. 9. Software input/output image (right side view)

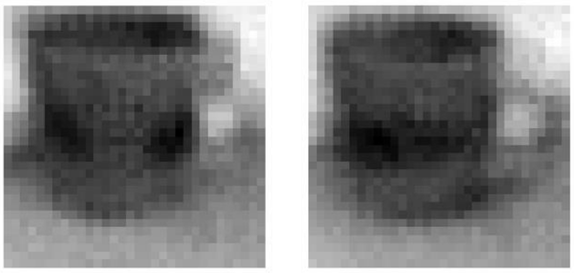


Fig. 10. Output image from average of latent vectors for each viewpoint
(Left: left viewpoint and center viewpoint
Right: right and center viewpoints)

Ⅵ. CONCLUSION

In this study, as an application of VAE to image generation, we designed and implemented a circuit for angle completion to generate images from an unknown viewpoint. The circuit design needed to handle 1024 dimensional features, so we focused on efficient memory access and optimal allocation of computing resources.

However, implementation issues remained, and we were unable to evaluate the processing time and circuit size on FPGA. In the future, it will be necessary to modify the circuit to ensure stable operation on an actual device, measure processing speed, and compare performance with an emulator. We also plan to optimize the circuit design by evaluating resource utilization.
As for the creation of 3D models, which is one of the objectives of this system, we intend to implement a function to generate 3D models from images obtained from multiple viewpoints. In addition, we will improve the resolution of the input data to achieve more accurate circuit design, thereby realizing a highly practical system.

We also aim to realize a system that enables efficient 3D modeling and image processing in industrial applications and research and development sites. We hope that this system will be widely used as a support tool for creative activities in various fields.

REFERENCES

[1] Xilinx, “ZCU104.” Available: https://japan.xilinx.com/products/boards-and-kits/zcu104.html. [Accessed: Dec. 10, 2024].

[2] AMD, “PYNQ | Python Productivity to AMD Adaptive Compute Platforms.” [Accessed: Dec. 10, 2024].

[3] 斎藤 康毅, *ゼロから作る Deep Learning ー Python で学ぶディープラーニングの理論と実装ー*, オーム社, 2018.

[4] D. P. Kingma and M. Welling, “Auto-Encoding Variational Bayes,” *arXiv preprint*, arXiv:1312.6114, 2013. [Accessed: Dec. 20, 2024].

[5] Xilinx Inc., “Vivado Design Suite: AXI リファレンスガイド (UG1037),” https://docs.xilinx.com/v/u/ja-JP/ug1037-vivado-axi-reference-guide. [Accessed: Dec. 20, 2024].

[6] AMD, AXI の基礎 1 - AXI の概要, https://adaptivesupport.amd.com/s/article/1087958?language=ja, [Accessed: Jan. 31, 2025]

また来年沖縄で、お会いしましょう。

©098free

コンテストに関してのお問合せ：
九州工業大学情報工学部情報・通信工学科尾知研究室内
LSIデザインコンテスト実行委員会事務局

TEL：0948-29-7667
http://www.lsi-contest.com