BinaryBrain Documentation

リリース *4.2.5*

Ryuji Fuchikami

2023 年 08 月 10 日

Contents:

本書は *BinaryBrain Ver4*: https://github.com/ryuz/BinaryBrain/tree/ver4_release のドキュメントです。

第 1 章

はじめに

1.1 概要

1.1.1 特徴

BinaryBrain は主に当サイトが研究中の LUT(Look-Up Table)-Network を実験することを目的に作成したディープラーニング用のプラットフォームです。

LUT-Network の評価を目的に作成しておりますが、それ以外の用途にも利用可能です。

以下の特徴があります

- ニューラルネットの FPGA 化をメインターゲットにしている
- バイナリネットであるも関わらず変調技術により Autoencode や回帰分析が可能
- 独自の Differentiable-LUT モデルにより、LUT の性能を最大限引き出したが学習できる
- 量子化＆疎行列のネットワークでパフォーマンスの良い学習が出来る環境を目指している
- C++ で記述されている
- GPU(CUDA) に対応している
- 高速でマニアックな自作レイヤーが作りやすい
- Python からの利用も可能

1.1.2 基本的な使い方

基本的には C++ や Python で、ネットワークを記述し、学習を行った後にその結果を verilog などに埋め込んで、FPGA 化することを目的に作成しています。

C++ 用の CPU 版に関してはヘッダオンリーライブラリとなっているため、include 以下にあるヘッダファイルをインクルードするだけでご利用いただけます。

GPU を使う場合は、ヘッダ読み込みの際に BB_WITH_CUDA マクロを定義した上で、cuda 以下にあるライブラリをビルドした上でリンクする必要があります。

また、BB_WITH_CEREAL マクロを定義すると、途中経過の保存形式に json が利用可能となります。

Python 版を使う場合は、一旦ビルドに成功すれば import するだけで利用可能です。

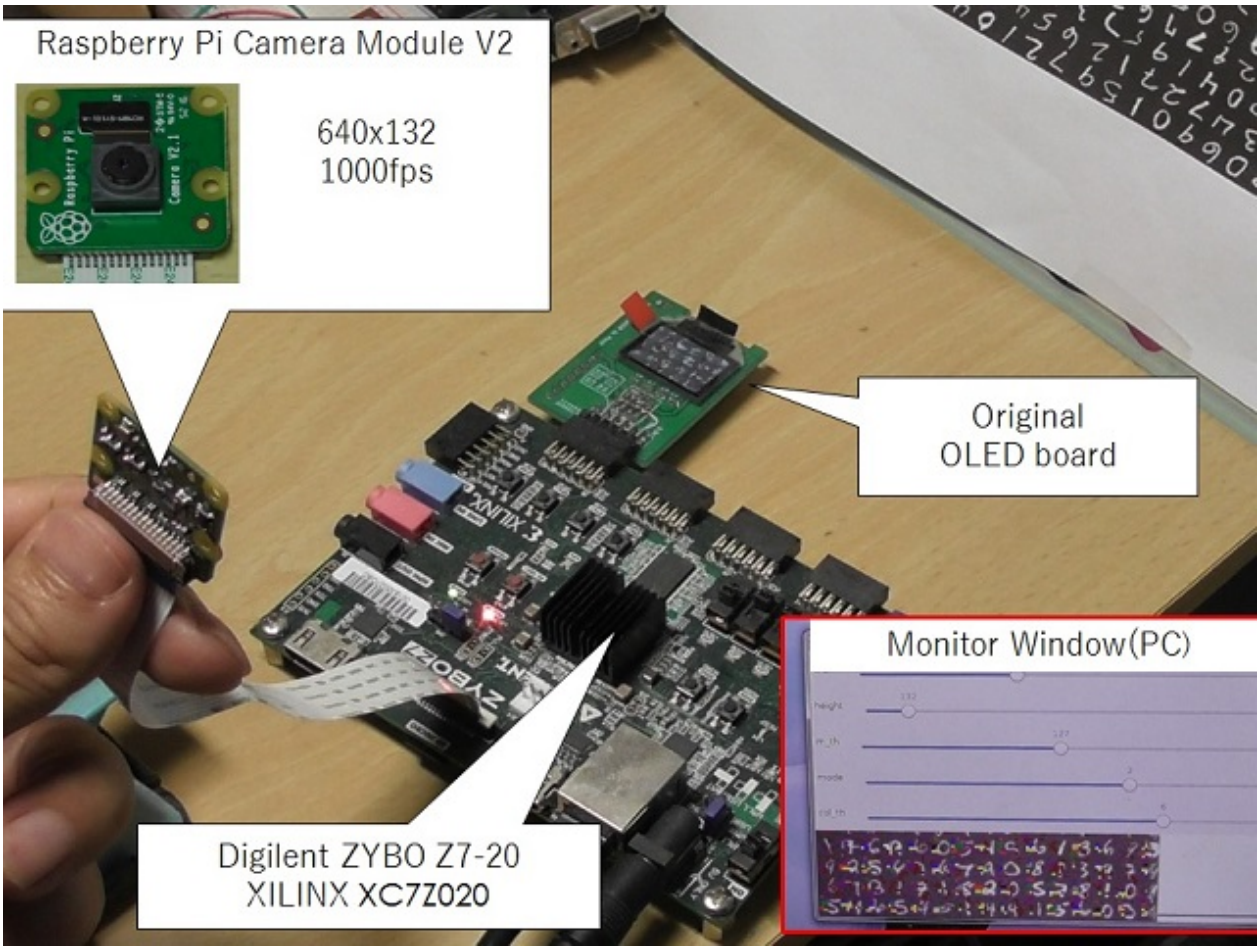
使い方は samples などを参考にしてください。

1.2 事例紹介

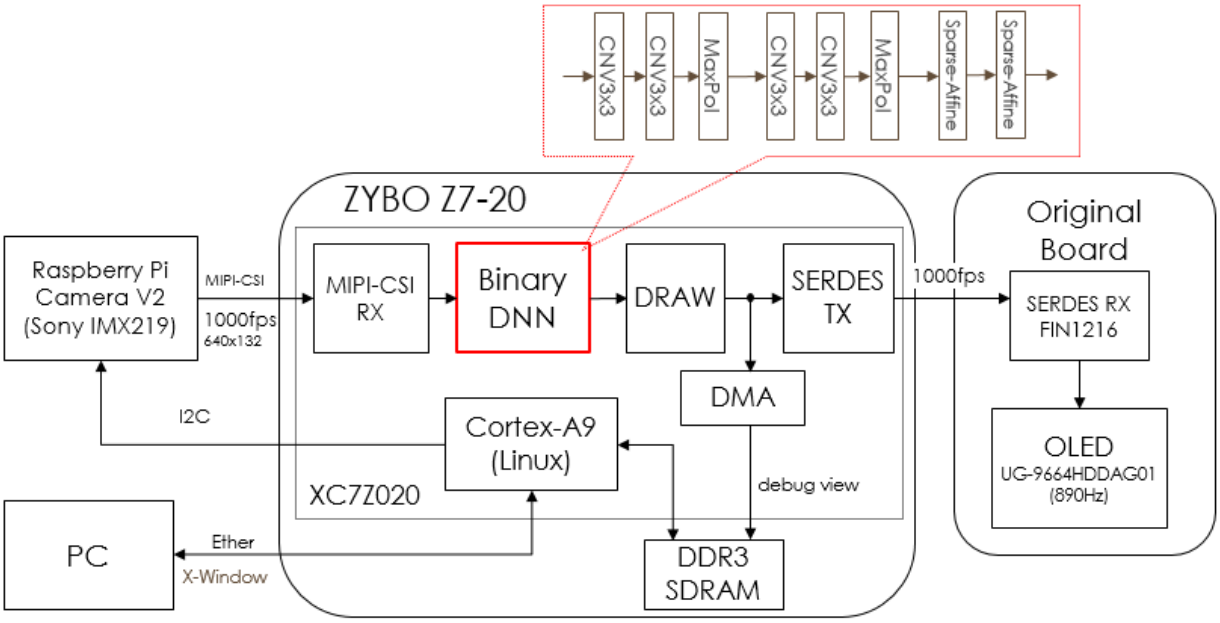
1.2.1 リアルタイム認識

1.2.1.1 実装事例

フルバイナリネットワークで、遅延数ミリ秒 (1000fps) での画像認識の例です。



下記のようなブロック図となっています。



1.2.1.2 FPGA リソース

いくつかの認識について実験したものを以下に示します。

XC7Z020CLG400-1						
tareget	LUT	BRAM	input-size	oversample	fps	accuracy
MNIST Simple DNN System(Camera + OLED)	10,944	54	640x132	x1	1,000	0.913
MNIST CNN System(Camera + OLED)	13,193	45	640x132	x1	1,000	0.928
MNIST Simple DNN	1,460	0	28x28	x1	318,878	0.880
MNIST Simple DNN (oversampling)	1,460	0	28x28	x15	21,259	0.913
MNIST CNN Core	5,444	13	28x28	x1	318,878	0.928
MNIST CNN Core (oversampling)	5,444	13	28x28	x15	21,259	0.986
CIFAR-10 CNN Core	24,784	16	32x32	x1	244,141	0.348
CIFAR-10 CNN Core (oversampling)	24,784	16	32x32	x15	16,276	0.583

下記はカメラや OLED などの制御回路も含んだものもありますが、例えば MNIST の Simple DNN であればニューラルネット部分はわずか 1460 個の LUT のみで 88 %の認識が可能です。これは、今手に入る XILINX のもっとも小さな FPGA でも十分収まるサイズです。

これは 1024-360-60-10 の 4 層構造のネットワークであり、例えば 200MHz で動かした場合、4 サイクル (=20 ナノ秒) で認識が完了します。そのため極めてリアルタイム性の高い用途への応用も可能です。

もしカメラなどの入力に制約がなく、28x28 の画像を毎サイクル供給可能であれば、コア自体は 200Mfps で動作可能となります。これは 1 つの対象に対して条件を変えながら非常に多くの認識を行える帯域ですので、1 回の認識率は低くても、結果を二次加工することで実用的な認識率を目指すようなことも可能な帯域です。

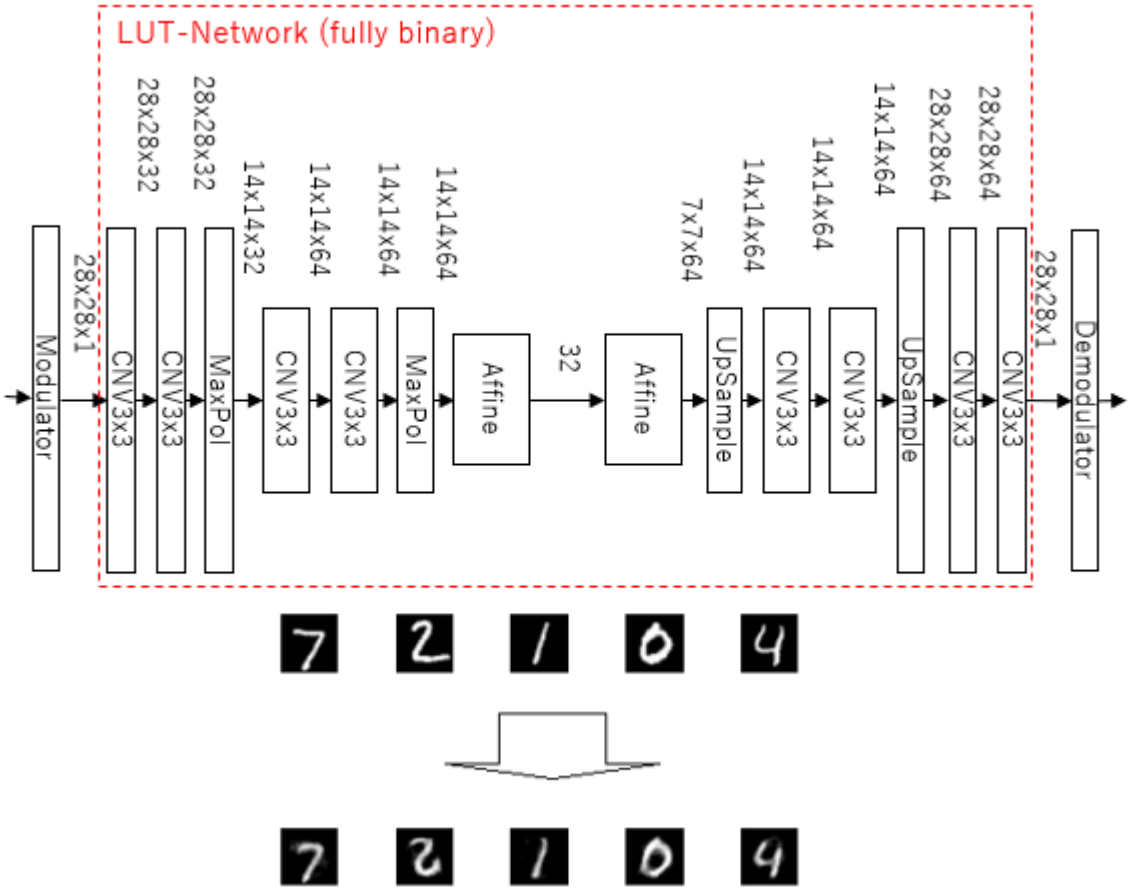
1.2.2 Autoencoder

通常のバイナリネットワークは出力もバイナリであるため、例えば Autoencoder のような多値出力が必要な用途には応用が難しいという課題があります。(入力に関しては最初の数層を多値で扱う手はあります)

BinaryBrain では、バイナリ変調を用いることで、入力から出力まで全層がバイナリである Fully binary neural network で多値データを扱う方法を提供しています。

1.2.2.1 MNIST

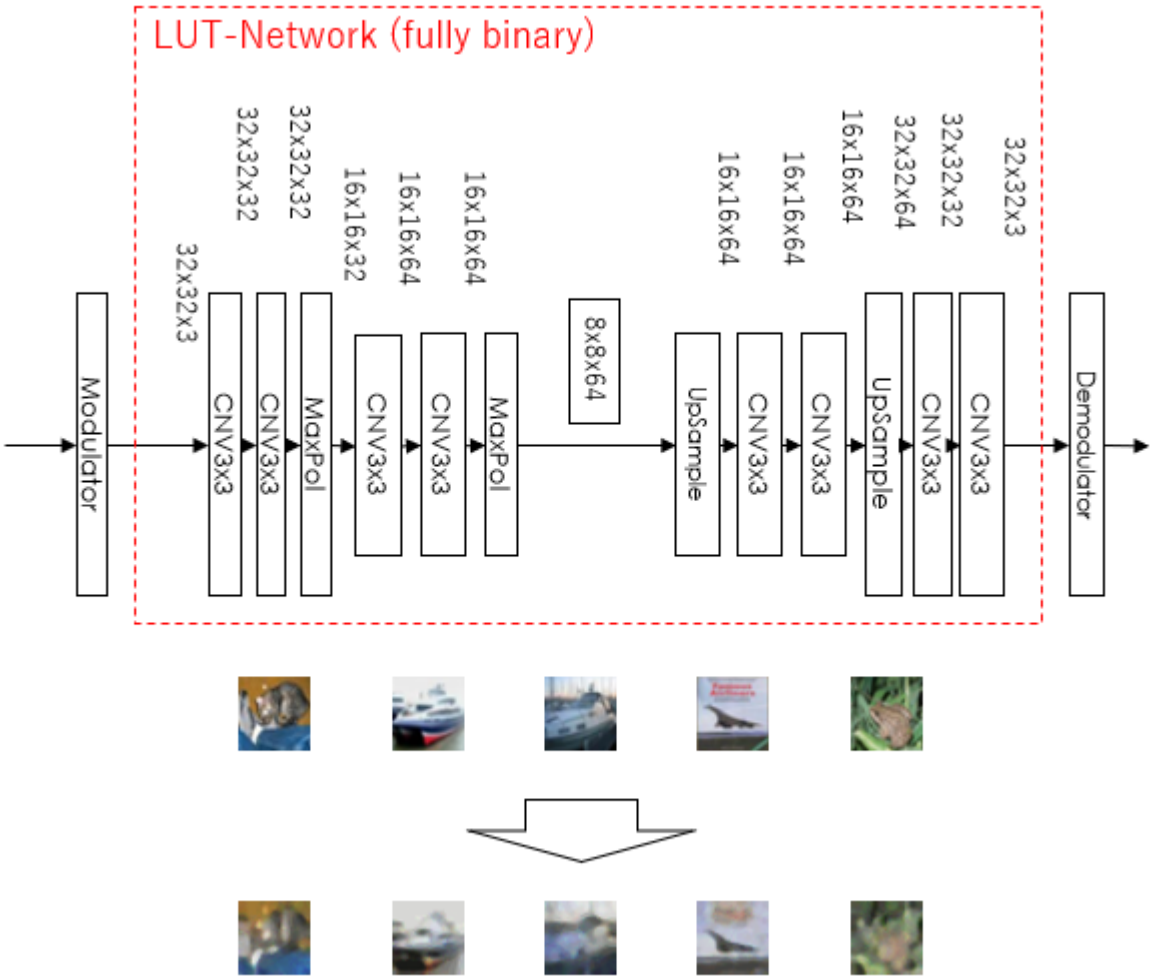
MNIST での Autoencoder の実験結果です。



MNIST 画像自体が 2 値に近いのですが、輪郭付近でやや滑らかさが出ています。

1.2.2.2 CIFAR-10

同様に CIFAR-10 のデータセットで扱ったものです。



ぼやけた感じは否めませんが、多値出力に対してある程度のことができているのは確認できます。

もともとが CIFAR-10 のデータセット自体が Autoencoder のような学習を目的としたデータセットではないので、多値の従来ネットワークでもかなりボケた画像しか作れない部分はあるので、まずは実験的な結果と言えます。

1.3 LUT-Network とは

1.3.1 概要

LUT-Network とは、当サイトの提唱するパーセプトロンモデルの代わりに LUT(ルックアップテーブル) のモデルを利用したディープニューラルネットワークのことです。重みの乗算の代わりにテーブル引きを行うことで、パーセプトロンでは学習することのできない XOR パターンのようなものも柔軟に学習することができます。乗算を用いない為、低スペックな計算環境でも高速に推論を行うことが可能です。

特にこれをバイナリ化した Binary LUT-Network は、FPGA の LUT に直接変換可能であるため、極めて高い演算

機高率を実現できます。

また以下のバイナリ化の欠点をバイナリ変調技術で克服する方法も提供しています

- ネットワーク部分は入力初段からフルバイナリネットワークを実現可能で高効率
- 出力を多値に戻せるため、回帰分析や AutoEncoder などのアプリケーションにも適用可能
- FPGA だと 1 レイヤーの計算が 1 サイクルで終わるのでナノ秒クラスで認識できる（高リアルタイム性）
- 超廉価＆低消費電力なワンコイン FPGA から適用が可能

高価な乗算機アレイが不要となるので、特に電力やリアルタイム性、コストなどが課題となるエッジコンピューティング分野にちょっとした認識を実現するなどに適したネットワークです。

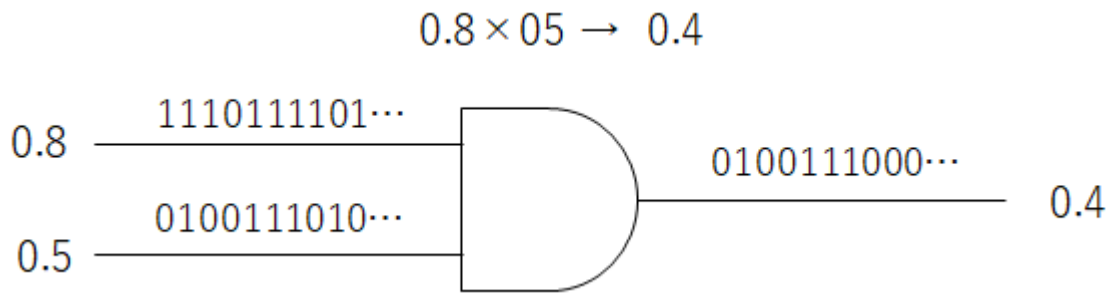
1.3.2 Differentiable-LUT モデル

LUT によるテーブル参照を誤差逆伝搬で学習させているというと、奇妙に感じられるかもしれません。一般にデジタル回路は AND や OR などの組み合わせ回路で実現されますので、そのままではこれを微分して逆伝搬することはできません。しかしながら機械学習で扱う値は多くの場合は尤度であって、一定の条件下でデジタル回路は Stoachstic 演算に置き換えても機能します。ここに着想を得て、FPGA の LUT 回路を微分して、テーブル内を学習可能な構成を得たうえで、さらになるべく広い条件で利用できるように工夫を要れたものが本モデルとなります。

1.3.2.1 Stochastic-LUT 演算

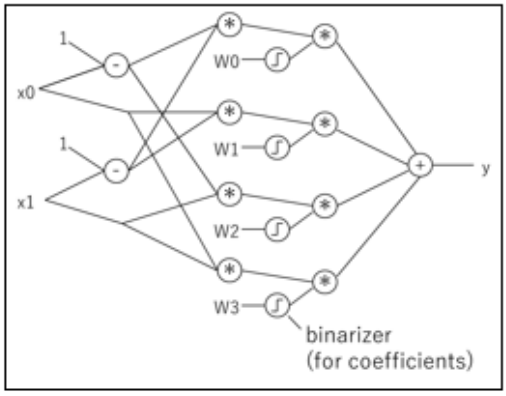
その計算モデルは、LUT のテーブル引きの回路演算を Stoachstic 演算に置き換えて実験しているときに発見されました。Stochastic-LUT モデルは Differentiable-LUT モデルの中にその一部として含まれています。

Stochastic 演算とは、確率的な 0/1 が入力される回路におけるデジタル演算において、その確率値に対して乗算などの演算子として機能する点に着目したものです。例えば AND ゲートは確率値に対しては乗算器として機能します。



後述しますが、BinaryBrain では任意のフルバイナリネットワークに対して、バイナリ変調を施したデータを入出力させながら学習させる機能があり、このバイナリ変調が Stochastic 演算を用いたモデルを有効に機能させるのに役立たせることができます。。

さて、早速ですが LUT も回路的には単なるマルチプレクサですので、一度デジタル回路として考えた後に、Stoachstic 演算に置き換えて考えることで下記のように微分可能な計算で表すことができます。



[n input LUT]
input : $x_0, x_1, \cdots x_n$
weight : $W_0, W_1, \cdots W_{2^n}$
output : y

$$\overline{x_n} = 1 - x_n$$

2input LUT

$$\begin{aligned} y = {} & W_0 \cdot \overline{x_1}\ \overline{x_0} \\ & + W_1 \cdot \overline{x_1}\ x_0 \\ & + W_2 \cdot x_1\ \overline{x_0} \\ & + W_3 \cdot x_1\ x_0 \end{aligned}$$

4 input LUT

$$\begin{aligned} y = {} & W_0 \cdot \overline{x_3}\ \overline{x_2}\ \overline{x_1}\ \overline{x_0} \\ & + W_1 \cdot \overline{x_3}\ \overline{x_2}\ \overline{x_1}\ x_0 \\ & + W_2 \cdot \overline{x_3}\ \overline{x_2}\ x_1\ \overline{x_0} \\ & + W_3 \cdot \overline{x_3}\ \overline{x_2} x_1\ x_0 \\ & + W_4 \cdot \overline{x_3}\ x_2\ \overline{x_1}\ \overline{x_0} \\ & + W_5 \cdot \overline{x_3}\ x_2\ \overline{x_1}\ x_0 \\ & + W_6 \cdot \overline{x_3}\ x_2\ x_1\ \overline{x_0} \\ & + W_7 \cdot \overline{x_3}\ x_2\ x_1\ x_0 \\ & + W_8 \cdot x_3\ \overline{x_2}\ \overline{x_1}\ \overline{x_0} \\ & + W_9 \cdot x_3\ \overline{x_2}\ \overline{x_1}\ x_0 \\ & + W_{10} \cdot x_3\ \overline{x_2}\ x_1\ \overline{x_0} \\ & + W_{11} \cdot x_3\ \overline{x_2}\ x_1\ x_0 \\ & + W_{12} \cdot x_3\ x_2\ \overline{x_1}\ \overline{x_0} \\ & + W_{13} \cdot x_3\ x_2\ \overline{x_1}\ x_0 \\ & + W_{14} \cdot x_3\ x_2\ x_1 \overline{x_0} \\ & + W_{15} \cdot x_3\ x_2\ x_1\ x_0 \end{aligned}$$

6 input LUT

$$\begin{aligned} y = {} & W_0 \cdot \overline{x_5}\ \overline{x_4}\ \overline{x_3}\ \overline{x_2}\ \overline{x_1}\ \overline{x_0} \\ & + W_1 \cdot \overline{x_5}\ \overline{x_4}\ \overline{x_3}\ \overline{x_2}\ \overline{x_1}\ x_0 \\ & + W_2 \cdot \overline{x_5}\ \overline{x_4}\ \overline{x_3}\ \overline{x_2}\ x_1\ \overline{x_0} \\ & + W_3 \cdot \overline{x_5}\ \overline{x_4}\ \overline{x_3}\ \overline{x_2} x_1\ x_0 \\ & + W_4 \cdot \overline{x_5}\ \overline{x_4}\ \overline{x_3}\ x_2\ \overline{x_1}\ \overline{x_0} \\ & + W_5 \cdot \overline{x_5}\ \overline{x_4}\ \overline{x_3}\ x_2\ \overline{x_1}\ x_0 \\ & \quad \cdot \\ & \quad \cdot \\ & \quad \cdot \\ & + W_{59} \cdot x_5\ x_4\ x_3\ \overline{x_2}\ x_1\ x_0 \\ & + W_{60} \cdot x_5\ x_4\ x_3\ x_2\ \overline{x_1}\ \overline{x_0} \\ & + W_{61} \cdot x_5\ x_4\ x_3\ x_2\ \overline{x_1}\ x_0 \\ & + W_{62} \cdot x_5\ x_4\ x_3\ x_2\ x_1 \overline{x_0} \\ & + W_{63} \cdot x_5\ x_4\ x_3\ x_2\ x_1\ x_0 \end{aligned}$$

W は各ルックアップテーブル内の値に対応し、テーブル内の値が 1 である確率を表します。x は入力値が 1 である確率値であり、y は出力が 1 となる確率値です。

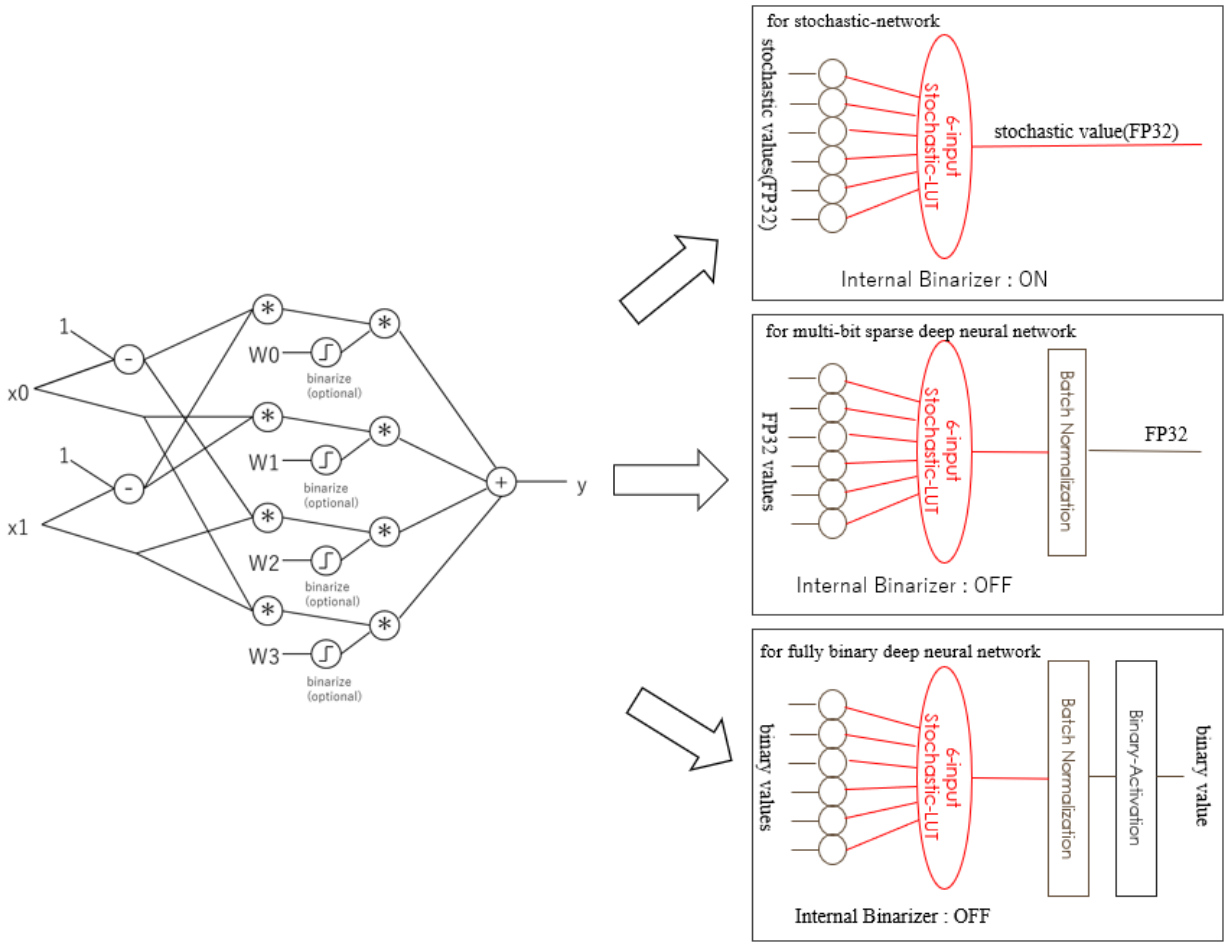
このモデルを使った学習は、入力同士に相関がなく、純粋に確率値として扱える範疇において、正しく機能し、出力値が一定の確率で 1 を出力するようにネットワーク全体を学習させることが可能です。

内部に備えた W の後にある Binaeizer を ON にして学習すれば、学習完了後にテーブル値はバイナリに置換することができます。

1.3.2.2 Differentiable-LUT モデルの全体像

Stochastic-LUT の計算モデルを活用し、Stochastic 性を持たないデータも視野に入れて広く学習可能にするための疎結合ルックアップテーブル方式のモデルとして、Differentiable-LUT モデルを提唱しています。

BinaryBrain の備える DifferentiableLUT クラスは下記の 3 つの使い方に対応しています。

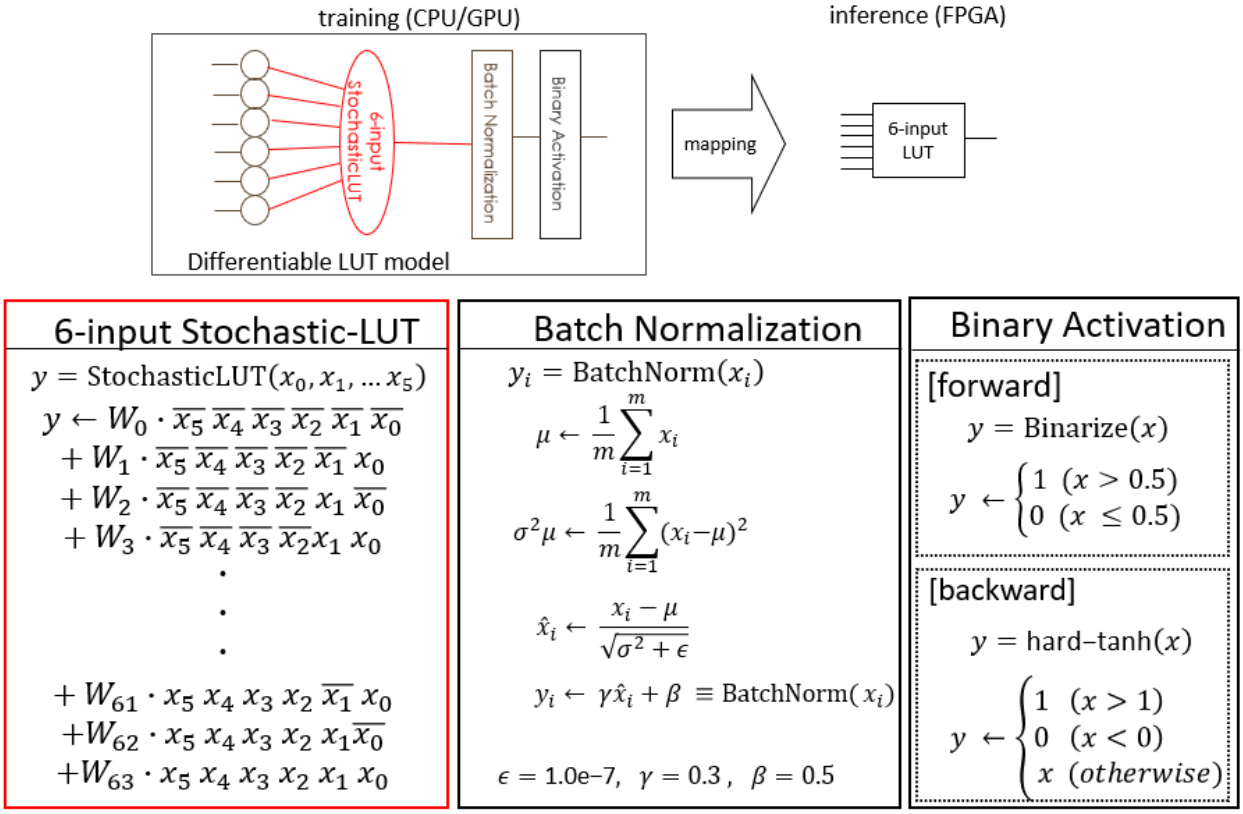


- Stochastic 値を扱うネットワークを学習可能
- 非バイナリ (FP32 など) の疎結合ネットワークで従来のパーセプトロンよりも高性能に機能
- バイナリ疎結合ネットワークで LUT に置換可能なモデルとして学習可能

1.3.2.3 Differentiable-LUT モデルによる FPGA 化

Differentiable-LUT を、Stochastic 演算用や、Fully-Binary 用に利用した場合には、FPGA に 1 個に割り当て可能なモデルとして学習させることができます。

特に Fully-Binary 用に利用しする場合が広く汎用的に応用可能であり、下記のようなモデルになります。



1.4 バイナリ変調

1.4.1 概要

本章ではバイナリ LUT-Network に限らず、広くバイナリネットワークに適用可能な技術として、バイナリ変調の適用について述べます。バイナリ変調とフルバイナリネットワークの組み合わせは、本サイトの提唱する技術の 1 つであり、入出力のに多値データが要求される場合にバイナリネットワークを適用するための手法です。

1.4.2 従来のバイナリネットワーク

従来のバイナリネットワークでは、多値画像の認識などを行うために、入力側のいくつかの層をバイナライズせずに多値入力とすることで多値データを扱っていました。この方法は一定の効果はあるものの、入力層では乗算器を必要とする為リソースが大きく増加する上に、出力はバイナリであり、クラスタ分類ぐらいにしか応用できないという課題がありました。

1.4.3 バイナリ変調

信号処理の世界にはバイナリ変調という技術があります。例えばデジタルオーディオなどの分野では 1bit ADC や D 級アンプの技術は非常に重要です。こでは信号をオーバーサンプリングにより、高い周波数の 1bit のデータに量子化することで、信号処理自体はバイナリで扱うにもかかわらず、入出力データには例えば 16bit 以上の高品質の信号を得る技術です。

もっとも簡単な方法はアナログ値を乱数閾値でバイナリ化することです。結果は元のアナログ値に応じた確率で 1 と 0 が生成されますので、扱いたい値がそのまま Stochastic 演算の対象となります。しかしながら確率的な振る舞いはデータ数が充分多い時に顕在化してきますので信号オーバーサンプリングは重要な技法となってきます。

BinaryBrain では同様の変調を元データに施してデータを水増しすることで、非常に小さな回路の認識率を上げたり、Autoencoder や回帰分析などの多値出力を必要とする分野への適用可能性を広げました。

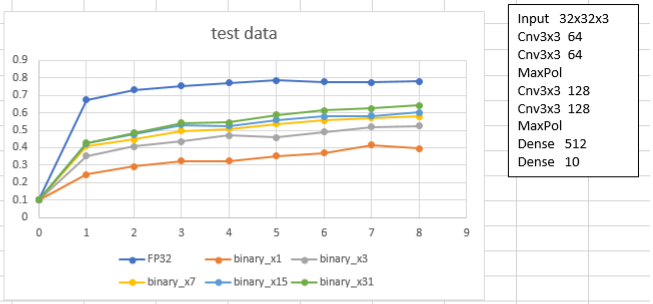
下記は、通常の Dense CNN の ReLU を Binarizer に置き換え、入力もバイナリ化してフルバイナリネットワーク化したものを用いて、バイナリ変調の効果を実験した結果です。

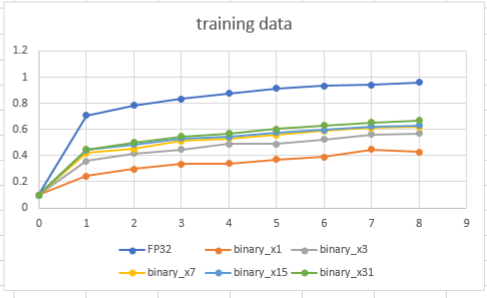
test data

epoch	FP32	binary_x1	binary_x3	binary_x7	binary_x15	binary_x31
0	0.099	0.1	0.1	0.1	0.1	0.1
1	0.6751	0.2453	0.3534	0.4105	0.4241	0.4284
2	0.7321	0.2917	0.4073	0.4464	0.4754	0.4842
3	0.7535	0.3237	0.4343	0.497	0.5299	0.5421
4	0.7723	0.3246	0.4689	0.5079	0.5254	0.5454
5	0.7875	0.3535	0.4581	0.5351	0.5593	0.5887
6	0.7786	0.3708	0.4898	0.5559	0.5781	0.6137
7	0.7757	0.4169	0.5187	0.5707	0.5835	0.6239
8	0.7812	0.3956	0.5242	0.5782	0.6032	0.6426

training data

epoch	FP32	binary_x1	binary_x3	binary_x7	binary_x15	binary_x31
0	0.0989	0.1	0.1	0.1	0.1	0.1
1	0.7078	0.244	0.3577	0.4213	0.4435	0.4471
2	0.7825	0.2956	0.4157	0.4525	0.4858	0.4987
3	0.8342	0.3357	0.4451	0.5105	0.5307	0.546
4	0.8763	0.3381	0.4896	0.5322	0.5418	0.5689
5	0.9135	0.3716	0.4878	0.5598	0.5765	0.6043
6	0.9321	0.3878	0.524	0.5877	0.5977	0.6302
7	0.9415	0.4461	0.5599	0.6129	0.6221	0.6528
8	0.9584	0.4282	0.5713	0.6189	0.6299	0.6696





binary_x1 が 1 倍のオーバーサンプル、すなわち何もせずに単純にフルバイナリ化した場合ですが、FP32 での結果に比べて大きく認識率が落ち込みます。そして、binary_x3、binary_x7, binary_x15, binary_x31 が、それぞれ 3

倍、7 倍、15 倍、31 倍のオーバーサンプリングでのバイナリ変調を行ったものですが、ある程度の回復を見せている事がうかがえます。

同じ回路に、より高いフレームレートで、変調したデータを通すだけなので、スループットは低下しますが、ネットワークを構成する回路自体のリソースは一切変化することなく、認識率だけが向上しているのが特徴です。

第 2 章

クイックスタート (C++)

まずはじめに付属の MNIST サンプルを動かすまでを紹介します。

AXV2 以降の命令が使える CPU と、Windows7 以降もしくは Linux の環境を想定しております。CUDA にも対応していまが、nvcc が利用可能な環境でビルドする必要があります。

CUDA については NVIDIA のページを参考に事前にインストールください。https://developer.nvidia.com/cuda-downloads

なお make 時に make WITH_CUDA=No と指定することで、GPU を使わない CPU 版もビルド可能です。

2.1 Windows

1. install VisualStudio 2019 + CUDA 11.3
2. git clone --recursive -b ver4_release https://github.com/ryuz/BinaryBrain.git
3. download MNIST from http://yann.lecun.com/exdb/mnist/
4. decompress MNIST for "samplescppmnist"
5. open VC++ solution "samplescppmnistsample_mnist.sln"
6. build "x64 Release"
7. run

2.2 Linux(Ubuntu 20.04)

2.2.1 1. install tools

```
% sudo apt update
% sudo apt upgrade
% sudo apt install git
% sudo apt install make
% sudo apt install g++
% wget https://developer.download.nvidia.com/compute/cuda/11.3.1/local_installers/cuda_
↪11.3.1_465.19.01_linux.run
% sudo sh cuda_11.3.1_465.19.01_linux.run
```

2.2.2 2. build and run

```
% git clone --recursive -b ver4_release  https://github.com/ryuz/BinaryBrain.git
% cd BinaryBrain/samples/cpp/mnist
% make
% make dl_data
% ./sample-mnist All
```

ここで単に

```
% ./sample-mnist
```

と打ち込むと、使い方が表示されます。

2.3 Google Colaboratory

nvcc が利用可能な Google Colaboratory でも動作可能なようです。以下あくまで参考ですが、ランタイムのタイプを GPU に設定した上で、下記のような操作で、ビルドして動作させることができます。

```
!git clone --recursive -b ver4_release  https://github.com/ryuz/BinaryBrain.git
%cd BinaryBrain/samples/cpp/mnist
!make all
!make run
```

第 3 章

クイックスタート (Python)

BinaryBrain は pybind11 を利用して Python からの呼び出しも可能にしています。python3 を前提としています。

3.1 pip でのインストール

下記のコマンドでインストール可能 (になる予定) です。

```
% pip3 install binarybrain
```

Google Colaboratory からも

```
!pip install binarybrain
```

とすることでインストール可能です。

BinaryBrain は python3 専用です。Python2 との共存環境の場合など必要に応じて pip3 を実行ください。そうでなければ pip に読み替えてください。インストール時にソースファイルがビルドされますので、コンパイラや CUDA などの環境は事前に整えておく必要があります。

(Windows 版はバイナリ wheel が提供されるかもしれません。作者環境は ver4.0.1 現在、Python 3.7.4(Windows10)、Python 3.6.9(Ubuntu 18) です)

Python 用のサンプルプログラムは下記などを参照ください。

https://github.com/ryuz/BinaryBrain/tree/ver4_release/samples/python

（ipynb 形式ですので、Jupyter Notebook、Jupyter Lab、VS code、PyCharm、GoogleColab など、読める環境を準備ください。）

3.2 setup.py でのインストール

pip でのインストールがうまくいかない場合や、github 上の最新版を試したい場合などは setup.py でのインストールも可能です。

3.2.1 事前準備

Python 版は各種データセットに PyTorch を利用しています。事前にインストールください。

またその他必要なパッケージを事前にインストールください。pybind11 などが必須です。

```
% pip3 install setuptools
% pip3 install pybind11
% pip3 install numpy
% pip3 install tqdm
```

Windows 環境の場合、nvcc のほかにも VisualStudio の 64bit 版がコマンドラインから利用できるようにしておく必要があります。例えば以下のように実行しておきます。 x64 の指定が重要です。

```
> "C:\Program Files (x86)\Microsoft Visual Studio\2017\Community\VC\Auxiliary\Build\
↪vcvarsall.bat" x64
```

3.2.2 インストール (pip を使う場合)

下記のコマンドでインストール可能です。

```
% pip install binarybrain
```

3.2.3 インストール (setup.py を使う場合)

下記のコマンドでインストール可能です。

```
% # install
% cd python
% python3 setup.py install
```

3.2.4 サンプルの実行

コマンドラインから以下のサンプルを試すことができます。

```
% cd samples/python/mnist

% # Simple DNN sample
% python3 MnistDifferentiableLutSimple.py

% # CNN sample
% python3 MnistDifferentiableLutCnn.py
```

その他のサンプルは ipynb 形式で samples/python フォルダの中にあるので Jupyter Notebook などで参照ください。

3.2.5 Google Colaboratory での setup.py

Google Colaboratory で利用する場合は、ランタイムのタイプを「GPU」にして、下記を実行した後にランタイムの再起動を行えば利用できるようになるはずです。

```
!pip install pybind11
!git clone -b ver4_release  https://github.com/ryuz/BinaryBrain.git
%cd BinaryBrain
!python3 setup.py install --user
```

第 4 章

クイックスタート (Verilog)

4.1 RTL Simulation の試し方

C++, Python ともに Verilog RTL のソースファイルの出力が可能です。出力した RTL の試し方は

https://github.com/ryuz/BinaryBrain/blob/ver4_release/samples/verilog/mnist/README.md

のなどをご参照ください。

第 5 章

C++ API

5.1 概要

本章では C++ の API について触れます。

現時点では細かなドキュメントが用意できておらず、ソースを読み人のために概要を掴む為の情報を記載します。

なお BinaryBrain のコードは namespace に bb という名称を持ちます。

5.2 モデルクラス

5.2.1 基本クラス

すべてのレイヤーは Model クラスからの派生で生成されます。

5.2.1.1 Model(抽象クラス)

抽象クラスは直接生成できませんが、各レイヤーの基礎となっており、操作を定義します。以下のようなメソッドを備えます。

SendCommand() 文
: 字列によって汎用的に各レイヤーの属性変更などを行えます。階層的にサブレイヤーに伝播させることを目的としておりますが、送信先クラス名を指定することで、特定のレイヤにのみコマンドを送ることも出来ます。現在の主な用途として "binary true" のようなコマンドで、バイナリ活性層を有効にしたり、"host_only" コマンドで、部分的に動作を CPU 版に切り替えたりできます。将来的には、部分的に学習時のパラメータ学習を固定したりなど、いろいろな設定を追加していくことを考えています。文字列なので、自作レイヤーに独自コマンドを追加することも簡単です。

GetClassName() ク
: ラス名を取得します。SendCommand() で、コマンド送付先をクラス名で指定することが出来ます。

SetName() クラス名とは別にインスタンス個別に自由に名前設定が出来ます。生成時に固有の名前をつけておけば、後から SendCommand() で、個別に属性変更コマンドが送れます。

GetParameters() 内部パラメータの参照を取得します。重み係数などが取得対象です。内部パラメータを持った自作レイヤーを作成する場合に実装が必要になります。

GetGradients() 内部パラメータの勾配への参照を取得します。Backward 時に値が計算され、主に Optimizer が利用します。内部パラメータを持った自作レイヤーを作成する場合に実装が必要になります。

SetInputShape() 入力のデータ形状を指定します。戻り値は出力のデータ形状となります。階層的にサブレイヤーに伝播させることを目的としており、各レイヤーを連結後に呼びさすことで内部パラメータのサイズが決定され初期化されます。自作レイヤーを作成する場合には必ず実装が必要になります。

Forward() 前方伝播を行います。階層的にサブレイヤーも実行することを想定しています。自作レイヤーを作成する場合には必ず実装が必要になります。

Backward() 誤差逆伝播を行います。階層的にサブレイヤーも実行することを想定しています。自作レイヤーを作成する場合には必ず実装が必要になります。

PrintInfo() レイヤーの情報を表示します。自作レイヤーを作成する場合に実装しておけば独自の情報を出力できます。

5.2.2 活性化層

5.2.2.1 Binarize クラス

バイナライズ層です。Forward では、0 を閾値に出力を 0 と 1 に二値化します。Backward では hard-tanh として動作します。バイナリネットワークの基礎となります。

5.2.2.2 ReLU クラス

普通の ReLU です。Binarize から派生しており、SendCommand() にて、"binary true" を送ることで Binarize 層として動作します。

5.2.2.3 Sigmoid クラス

普通の Sigmoid です。Binarize から派生しており、SendCommand() にて、"binary true" を送ることで Binarize 層として動作します。

5.2.3 演算層

5.2.3.1 StochasticLutN クラス

LUT-Network の LUT に相当する部分を Stochastic モデルに基づいて学習させるためのレイヤーです。Stochastic 計算に基づいて FPGA の LUT ロジックを微分可能記述に直して、テーブル内容の逆伝搬学習を可能としています。Stochastic バイナリデータが Stochastic 性を持っている対象への学習に限定されますが、DifferentiableLutN モデルよりも高速に学習させることが可能です。

このモデルは 1 層で XOR パターンを含めた LUT で表現可能な空間すべてを学習可能です。

5.2.3.2 DifferentiableLutN クラス

LUT-Network の LUT に相当する部分を独自のモデルで学習させるためのレイヤーです。StochasticLutN クラスにさらに BatchNormalization と hard-tanh によるバイナリ化を統合しています。Stochastic 性を持たない一般的なデータに対してバイナリネットワークとしての学習能力を持ちます。

このモデルも StochasticLutN 同様に 1 層で XOR パターンを含めた LUT で表現可能な空間すべてを学習可能です。

一方で 1 つのノードの接続数が FPGA 同様に 6 個などに限定されるため、疎結合網となり、ネットワークの構築には工夫が必要です。

5.2.3.3 MicroMlpAffine クラス

MicroMlp の構成要素で、入力数を 6 などに限定した疎結合、且つ、内部に隠れ層を備えた小さな MLP(Multi Layer Perceptron) の集合体です。入力数や隠れ層の数テンプレート引数で変更可能です。

5.2.3.4 MicroMlp クラス

LUT-Network の LUT に相当する部分をパーセプトロンを用いて学習させるレイヤーです。内部は MicroMlpAffine + BatchNormalization + 活性化層 の 3 層で構成されます。活性化層 は デフォルトは ReLU ですが、テンプレート引数で変更可能です。

5.2.3.5 DenseAffine クラス

いわゆる普通の浮動小数点による全結合のニューラルネットです。

5.2.3.6 BatchNormalization クラス

BatchNormalization 層です。活性化層でバイナリ化を行う前段ほぼ必須となってくる層です。

5.2.3.7 MaxPooling クラス

MaxPooling 層です。

5.2.3.8 LutLayer (抽象クラス)

LUT-Network を記述する基本モデルです。現在 ver2 の直接学習機能はまだ ver3 には未実装です。MicroMlp などで逆伝播で学習した内容をテーブル化して写し取ることを目的としています。テーブル化取り込みに ImportLayer() メソッドを備えます。

5.2.3.9 BinaryLutN クラス

各ノードの入力数を 1 つに固定した LUT モデルです。一般的な FPGA に適合します。入力数はテンプレート引数で指定でき、FPGA では 4 か 6 のものが一般的と思われます。入力数を固定することで演算を高速化できますが、ver3 への移植はまだ行えていません。

5.2.4 補助層

5.2.4.1 Sequential クラス

各種の層を直列に接続して 1 つの層として扱えるようにします。

5.2.4.2 Convolution2d クラス

Lowering を行い畳こみ演算を行います。

ConvolutionIm2Col + 引数で渡したモデル + ConvolutionCol2Im DenseAffine を渡すと、通常の CNN になり、MicroMlp を用いたサブネットワークを渡すことで、

LUT-Network での畳込みが可能です。

5.2.4.3 ConvolutionIm2 クラス

畳み込みの為の Lowering を行います。通常、 Convolution2d クラス の中で利用されます。Lowering されたデータに対して BatchNormalization するのも LUT-Network 学習時の特徴の一つかもしれません。

5.2.4.4 ConvolutionCol2Im クラス

畳み込みの為の Lowering の復元を行います。通常、 Convolution2d クラス の中で利用されます。

5.2.4.5 BinaryModulation クラス

内部で RealToBinary クラスと BinaryToReal クラスを組み合わせて、多値データをバイナリ化して学習するのに利用できます。

5.2.4.6 RealToBinary クラス

実数値をバイナライズします。その際に frame 方向に拡張して変調を掛ける (多重化) が可能です。現在、PWM 変調と、乱数での変調を実装しており、デフォルトで PWM 変調となります (将来 âŁ£ Σなどの誤差蓄積機能も検討中です)。変調を行うことで、入力値に対して確率的な 0/1 比率の値を生成できるため、出力も確率的なものとなります。

5.2.4.7 BinaryToReal クラス

多重化された確率的な 0 と 1 をカウンティングして実数値を生成します。RealToBinary 対応しますが、こちらは時間方向だけでなく、空間方向のカウントも可能です。オーバーサンプリングによる十分な多重化数が確保できれば、回路規模を増加させること無く回帰などの実数値へのフィッティング可能性が出てきます。

5.3 モデル以外のクラス

5.3.1 損失関数

5.3.1.1 LossSoftmaxCrossEntropy クラス

普通の Softmax-CrossEntropy クラスです。

5.3.1.2 "

平均二乗誤差を損失とするクラスです。

5.3.2 評価関数

5.3.2.1 MetricsCategoricalAccuracy クラス

Categorical Classification の精度を評価値とするクラスです。

5.3.2.2 MetricsMeanSquaredError クラス

MSE(平均二乗誤差) を評価値とするクラスです。

5.3.3 最適化 (Optimizer)

5.3.3.1 OptimizerSgd クラス

普通の SGD です。

5.3.3.2 OptimizerAdam クラス

普通の Adam です。

5.3.4 実行補助

5.3.4.1 Runner クラス

構築したモデルのフィッティングや評価などの実行を補助します。論より RUN。Runner のソースが各種の使い方で、参考になるはずです。

5.3.5 データ保持

5.3.5.1 Tensor クラス

多次元のデータを保持できるクラスで、演算も可能です。名前に反してまだ Tensor 演算は実装できていません。

5.3.5.2 Variables クラス

複数の Tensor を束ねる機能を持ったクラスです。形状が同じなら Variables 間での演算も可能です。主に Optimizer での利用を想定しています。

5.3.5.3 FrameBuffer クラス

１つの Tensor を 1 frame として、複数 frame を保持できるクラスです。ただし、内部では、NCHW や NHWC ではなく、CHWN 形式になるように並び替えてデータを保持しています。これは Lowering されて frame 数が十分増やされた疎行列に特化して性能を出すための配置で、BinaryBrain の特徴の一つです。一方で、一般的な算術ライブラリに適合しない (並び替えが必要) ので注意が必要です。

5.4 各種関数

5.4.1 FPGA へのエクスポート

5.4.1.1 ExportVerilog_LutLayers 関数

LutLayer を Verilog-RTL で出力します。

5.4.1.2 ExportVerilog_LutCnnLayersAxi4s 関数

畳み込み層を含む LutLayer を纏めて Verilog-RTL で出力します。MaxPooling などの入出力でデータが不連続になる層は最後に 1 つだけ指定することができます。

第 6 章

Python API

6.1 概要

Python 版モジュールは binarybarin パッケージを import することで利用可能です。

6.2 binarybarin パッケージ

binarybarin には以下のモジュールが含まれています。

6.2.1 基本クラス

6.2.1.1 Object クラス

class binarybrain.object.**Object**(*core_object=None*)

ベースクラス: object

各クラスの基底クラス

本クラスから派生する各種のクラスにはシリアライズの機能がサポートされる

dumps() → bytes

バイトデータにシリアライズ

モデルのデータをシリアライズして保存するためのバイト配列を生成

戻り値

Serialize data

戻り値の型

data (bytes)

loads(*data: bytes*) → bytes

バイトデータをロード

モデルのデータをシリアライズして復帰のバイト配列ロード

パラメータ

data (*bytes*) -- Serialize data

dump(*filename: str*)

ファイルに保存

モデルのデータをシリアライズしてファイルに保存

パラメータ

filename (*str*) -- ファイル名

load(*filename: str*)

ファイルからロード

ファイルからロード

パラメータ

filename (*str*) -- ファイル名

6.2.2 データ格納

6.2.2.1 DType クラス (Enum 定義)

class binarybrain.dtype.**DType**(*value*)

ベースクラス: IntEnum

データ型定義

BINARY = 2

BIT = 1

FP16 = 272

FP32 = 288

FP64 = 320

INT16 = 528

INT32 = 544

INT64 = 576

INT8 = 520

UINT16 = 784

UINT32 = 800

UINT64 = 832

UINT8 = 776

6.2.2.2 Tensor クラス

class binarybrain.tensor.**Tensor**(*shape: Optional[List[int]] = None*, ***, *dtype=DType.FP32*, *host_only=False*, *core_tensor=None*)

ベースクラス: *Object*

Tensor class

多次元データ構造。

パラメータ

- **shape** (*list[int]*) -- Shape of created array
- **dtype** (*int*) -- Data type
- **host_only** (*bool*) -- flag of host only

static from_numpy(*ndarray: ndarray*, *host_only=False*)

NumPy から生成

パラメータ

- **ndarray** (*ndarray*) -- array of NumPy
- **host_only** (*bool*) -- flag of host only

get_shape() → List[int]

データのシェイプ取得

戻り値

shape

get_type() → int

データ型取得

戻り値

data type.

numpy() → ndarray

NumPy の ndarray に変換

戻り値

ndarray (array)

6.2.2.3 FrameBuffer クラス

class binarybrain.frame_buffer.**FrameBuffer**(*frame_size: int = 0*, *shape: List[int] = []*, *dtype=DType.FP32*, *host_only: bool = False*, *core_buf=None*)

ベースクラス: *Object*

FrameBuffer class

BinaryBrain での学習データを格納する特別な型であるバッチと対応する 1 次元の frame 項と、各レイヤーの入出力ノードに対応する多次元の node 項を有している numpy の ndarray に変換する際は axis=0 が frame 項となり、以降が node 項となる

Tensor と異なり、frame 項に対して reshape を行うことはできず、node 項に対しても transpose することはできない。

node に関しては 2 次元以上の shape も持ちうるが、実際のレイヤー間接続に際しては、畳み込みなどの次元に意味を持つノード以外では、ノード数さえあっていれば接続できるものが殆どである (多くの処理系で必要とする flatten が省略できる)。

host_only フラグを指定すると device(GPU 側) が利用可能であっても host(CPU 側) のみにメモリを確保する

パラメータ

- **frame_size** (*int*) -- frame サイズ
- **shape** (*list[int]*) -- node シェイプ
- **dtype** (*int*) -- Data type

- **host_only** (*bool*) -- flag of host only

static from_numpy(*ndarray: ndarray*, *host_only=False*)

Create from NumPy

パラメータ

- **ndarray** (*np.ndarray*) -- array of NumPy
- **host_only** (*bool*) -- flag of host only

get_frame_size() → int

get size of frame.

戻り値

frame size.

get_frame_stride() → int

get stride of frame.

戻り値

frame stride.

get_node_shape() → List[int]

get shape of node.

戻り値

shape

get_node_size() → int

get size of node.

戻り値

node size.

get_type() → int

データ型取得

戻り値

dtype (DType)

numpy() → ndarray

Convert to NumPy

パラメータ

- **shape** (*list[int]*) -- Shape of created array

- **dtype** (*int*) -- Data type
- **host_only** (*bool*) -- flag of host only

6.2.2.4 Variables クラス

class binarybrain.variables.**Variables**

ベースクラス: object

Variables class

学習の為の Optimizer と実際の学習ターゲットの変数の橋渡しに利用されるクラス。内部的には各モデル内の重みや勾配を保有する Tensor をまとめて保持している。

append(*variables*)

変数を追加

パラメータ

variables (Variables) -- 追加する変数

6.2.3 基本モデル (Base models)

models モジュールには、ネットワークを構成するための各種演算モデルがあります。

6.2.3.1 Model クラス

class binarybrain.models.**Model**(*, *core_model=None*, *input_shape=None*, *name=None*)

ベースクラス: *Object*

Model class ネットワーク各層の演算モデルの基底クラスすべてのモデルはこのクラスを基本クラスに持つ

BinaryBrain では、モデルを実際にインスタンスとして生成して組み合わせることで学習ネットワークを構成する。特にネットワーク内のインスタンス化されたモデルをレイヤーという呼び方をする場合がある。

set_name(*name: str*)

インスタンス名の設定

生成したモデルには任意の名前を付けることが可能であり、表示や保存時のファイル名などに利用することが可能である

パラメータ

name (*str*) -- 新しいインスタンス名

get_name()

インスタンス名の取得

インスタンス名を取得する。名称が設定されていない場合はクラス名が返される

戻り値

name (str)

is_named()

インスタンス名の設定確認

インスタンス名が設定されているかどうか確認する

戻り値

named (bool)

get_model_name()

モデル名の取得

モデル名を取得する。

戻り値

model name (str)

get_info(*depth: int = 0*, *, *columns: int = 70*, *nest: int = 0*) → str

モデル情報取得

モデルの情報表示用の文字列を取得するそのまま表示やログに利用することを想定している

戻り値

info (str)

print_info(*depth: int = 0*)

モデル情報表示

モデルの情報を表示する

send_command(*command*, *send_to='all'*)

コマンドの送信

モデルごとにコマンド文字列で設定を行うコマンド文字列の定義はモデルごとに自由である
Sequential クラスなどは、保有する下位のモデルに再帰的にコマンドを伝搬させるので、複数の層

に一括した設定が可能である受取先は send_to で送り先はインスタンス名やクラス名で制限でき 'all' を指定するとフィルタリングされない

パラメータ

- **command** (*str*) -- コマンド文字列
- **send_to** (*str*) -- 送信先

set_input_shape(*input_shape*)

入力シェイプ設定

BinaryBarain ではモデル生成時に入力のシェイプを決定する必要はなくネットワーク構築後に、ネットワークを構成する各モデルの set_input_shape を順に呼び出して形状を伝搬させることで各モデルの形状の設定を簡易化できる

set_input_shape が呼ばれるとそれまでの各層で保有する情報は保証されない。ネットワーク構築後に一度だけ呼び出すことを想定している

パラメータ

input_shape (*List[int]*) -- 入力シェイプ

戻り値

出力シェイプ

戻り値の型

output_shape (List[int])

get_input_shape() → List[int]

入力シェイプ取得

戻り値

入力シェイプ

戻り値の型

input_shape (List[int])

get_output_shape() → List[int]

出力シェイプ取得

戻り値

出力シェイプ

戻り値の型

output_shape (List[int])

get_parameters()

パラメータ変数取得

学習対象とするパラメータ群を Variables として返す主に最適化 (Optimizer) に渡すことを目的としている

モデル側を自作する際には、このメソッドで戻す変数の中に含めなかったパラメータは学習対象から外すことができる為パラメータを凍結したい場合などに利用できる

戻り値

パラメータ変数

戻り値の型

parameters (*Variables*)

get_gradients()

勾配変数取得

get_parameters と対になる勾配変数を Variables として返す主に最適化 (Optimizer) に渡すことを目的としている

戻り値

勾配変数

戻り値の型

gradients (*Variables*)

forward(*x_buf*, *train=True*)

Forward

モデルは学習および推論の為に forward メソッドを持つ

train 変数を

パラメータ

- **x_buf** (FrameBuffer) -- 入力データ
- **train** (*bool*) -- True で学習、False で推論

戻り値

出力データ

戻り値の型

y_buf (*FrameBuffer*)

backward(*dy_buf*)

Backward

モデルは学習の為に backword メソッドを持つ必ず forwad と対で呼び出す必要があり、直前の forward 結果に対する勾配計算を行いながら逆伝搬する

BinaryBrain は自動微分の機能を備えないので、backword の実装は必須である

パラメータ

dy_buf (FrameBuffer) -- 入力データ

戻り値

出力データ

戻り値の型

dx_buf (*FrameBuffer*)

6.2.3.2 Sequential クラス

class binarybrain.models.**Sequential**(*model_list=None*, ***, *input_shape=None*, *name=None*)

ベースクラス: *Model*

Sequential class

複数レイヤーを直列に接続してグルーピングするクラス

リストの順番で set_input_shape, forward, backward などを実行するまた send_command の子レイヤーへのブロードキャストや、get_parameters, get_gradients の統合を行うことで複数のレイヤーを 1 つのレイヤーとして操作できる

パラメータ

model_list (*List[Model]*) -- モデルのリスト

append(*model*)

リストへのモデル追加

パラメータ

model (Model) -- リストに追加するモデル

get_model_list()

モデルリストの取得

戻り値

モデルのリスト

戻り値の型

model_list (List[*Model*])

set_model_list(*model_list*)

モデルリストの設定

パラメータ

model_list (*List[Model]*) -- モデルのリスト

6.2.3.3 Switcher クラス

class binarybrain.models.**Switcher**(*model_dict=None*, *init_model_name=None*, *, *input_shape=None*, *name=None*)

ベースクラス: *Model*

モデル切り替え用基底クラス

主に蒸留や転移学習などでレイヤー差し替えに用いる send_command から 'switch_model [name]' でも切り替え可能

パラメータ

- **(Dict{str** (*model_dict*) -- Model}): 切り替えるモデルの辞書
- **init_model_name** (*str*) -- 初期選択するモデルの名前

backward(*dy_buf*)

Backward

モデルは学習の為に backword メソッドを持つ必ず forwad と対で呼び出す必要があり、直前の forward 結果に対する勾配計算を行いながら逆伝搬する

BinaryBrain は自動微分の機能を備えないので、backword の実装は必須である

パラメータ

dy_buf (FrameBuffer) -- 入力データ

戻り値

出力データ

戻り値の型

dx_buf (*FrameBuffer*)

dumps()

バイトデータにシリアライズ

モデルのデータをシリアライズして保存するためのバイト配列を生成

戻り値

Serialize data

戻り値の型

data (bytes)

forward(*x_buf*, *train=True*)

Forward

モデルは学習および推論の為に forward メソッドを持つ

train 変数を

パラメータ

- **x_buf** (FrameBuffer) -- 入力データ
- **train** (*bool*) -- True で学習、False で推論

戻り値

出力データ

戻り値の型

y_buf (*FrameBuffer*)

get_gradients()

勾配変数取得

get_parameters と対になる勾配変数を Variables として返す主に最適化 (Optimizer) に渡すことを目的としている

戻り値

勾配変数

戻り値の型

gradients (*Variables*)

get_info(*depth: int = 0*, *, *columns: int = 70*, *nest: int = 0*) → str

モデル情報取得

モデルの情報表示用の文字列を取得するそのまま表示やログに利用することを想定している

戻り値

info (str)

get_input_shape() → List[int]

入力シェイプ取得

戻り値

入力シェイプ

戻り値の型

input_shape (List[int])

get_output_shape() → List[int]

出力シェイプ取得

戻り値

出力シェイプ

戻り値の型

output_shape (List[int])

get_parameters()

パラメータ変数取得

学習対象とするパラメータ群を Variables として返す主に最適化 (Optimizer) に渡すことを目的としている

モデル側を自作する際には、このメソッドで戻す変数の中に含めなかったパラメータは学習対象から外すことができる為パラメータを凍結したい場合などに利用できる

戻り値

パラメータ変数

戻り値の型

parameters (*Variables*)

loads(*data*)

バイトデータをロード

モデルのデータをシリアライズして復帰のバイト配列ロード

パラメータ

data (*bytes*) -- Serialize data

send_command(*command*, *send_to='all'*)

コマンドの送信

モデルごとにコマンド文字列で設定を行うコマンド文字列の定義はモデルごとに自由である Sequential クラスなどは、保有する下位のモデルに再帰的にコマンドを伝搬させるので、複数の層に一括した設定が可能である受取先は send_to で送り先はインスタンス名やクラス名で制限でき 'all' を指定するとフィルタリングされない

パラメータ

- **command** (*str*) -- コマンド文字列
- **send_to** (*str*) -- 送信先

set_input_shape(*input_shape: List[int]*)

入力シェイプ設定

BinaryBarain ではモデル生成時に入力のシェイプを決定する必要はなくネットワーク構築後に、ネットワークを構成する各モデルの set_input_shape を順に呼び出して形状を伝搬させることで各モデルの形状の設定を簡易化できる

set_input_shape が呼ばれるとそれまでの各層で保有する情報は保証されない。ネットワーク構築後に一度だけ呼び出すことを想定している

パラメータ

input_shape (*List[int]*) -- 入力シェイプ

戻り値

出力シェイプ

戻り値の型

output_shape (List[int])

switch_model(*model_name: str*)

モデルを切り替える

パラメータ

(Dict{str (*model_dict*) -- Model}): 切り替えるモデルの辞書

6.2.4 バイナリ変調モデル (Binary modulation)

models モジュールのうち、バイナリネットを構成する変調にかかわるモデルです。

6.2.4.1 RealToBinary class

class binarybrain.models.**RealToBinary**(*, *input_shape=None*, *frame_modulation_size=1*, *depth_modulation_size=1*, *value_generator=None*, *framewise=False*, *input_range_lo=0.0*, *input_range_hi=1.0*, *name=None*, *bin_dtype=DType.FP32*, *real_dtype=DType.FP32*, *core_model=None*)

ベースクラス: *Model*

RealToBinary class 実数値をバイナリ値に変換する。バイナリ変調機能も有しており、フレーム方向に変調した場合フレーム数 (=ミニバッチサイズ) が増える。またここでビットパッキングが可能であり、32 フレームの bit を int32 に詰め込みメモリ節約可能である

パラメータ

- **frame_modulation_size** (*int*) -- フレーム方向への変調数 (フレーム数が増える)
- **depth_modulation_size** (*int*) -- Depth 方向への変調数 (チャンネル数が増える)
- **framewise** (*bool*) -- True で変調閾値をフレーム単位とする (False でピクセル単位)
- **bin_dtype** (DType) -- 出力の型を bb.DType.FP32 もしくは bb.DType.BIT で指定可能

6.2.4.2 BinaryToReal class

class binarybrain.models.**BinaryToReal**(*, *frame_integration_size=1*, *depth_integration_size=1*, *output_shape=None*, *input_shape=None*, *name=None*, *bin_dtype=DType.FP32*, *real_dtype=DType.FP32*, *core_model=None*)

ベースクラス: *Model*

BinaryToReal class バイナリ値を実数値に戻す。その際にフレーム方向に変調されたデータを積算して元に戻すことが可能である

パラメータ

- **frame_integration_size** (*int*) -- フレーム方向の積算サイズ数 (フレーム変調の統合)

- **depth_integration_size** (*int*) -- チャンネル方向の積算サイズ (0 の時は output_shape 優先)
- **output_shape** (*List[int]*) -- 出力のシェイプ (指定が無ければ入力と同じ shape)
- **bin_dtype** (DType) -- 入力の型を bb.DType.FP32 もしくは bb.DType.BIT で指定可能

6.2.4.3 BitEncode class

class binarybrain.models.**BitEncode**(*bit_size=1*, ***, *output_shape=None*, *input_shape=None*, *name=None*, *bin_dtype=DType.FP32*, *real_dtype=DType.FP32*, *core_model=None*)

ベースクラス: *Model*

BitEncode class 実

数値をバイナリ表現として depth 方向に展開する

パラメータ

bit_size (*int*) -- エンコードする bit 数

6.2.4.4 Reduce class

class binarybrain.models.**Reduce**(*output_shape=None*, *integration_size=0*, ***, *input_shape=None*, *name=None*, *bin_dtype=DType.FP32*, *real_dtype=DType.FP32*, *core_model=None*)

ベースクラス: *Model*

Reduce class 多

重化されている出力を折り返して積和する

パラメータ

- **output_shape** (*[int]*) -- 出力のシェイプ
- **integration_size** (*int*) -- 積算するサイズ

6.2.5 演算モデル (Operation models)

models モジュールには、ネットワークを構成するための各種演算モデルがあります。

6.2.5.1 DifferentiableLut クラス

class binarybrain.models.**DifferentiableLut**(*output_shape=None*, *, *input_shape=None*, *connection='random'*, *binarize=True*, *batch_norm=True*, *momentum=0.0*, *gamma=0.3*, *beta=0.5*, *seed=1*, *name=None*, *N=6*, *bin_dtype=DType.FP32*, *real_dtype=DType.FP32*, *core_model=None*)

ベースクラス: SparseModel

DifferentiableLut class 微分可能 LUT モデル

内部計算的には StocasticLUT + BatchNormalization + Binarize(HardTanh) で構成される

FPGA 合成するためのルックアップテーブル型のモデルを学習することができる純粋な Stochastic 演算のみを行いたい場合は binarize と batch_norm の両方を False にすればよい。

パラメータ

- **output_shape** (*List[int]*) -- 出力のシェイプ
- **connection** (*str*) -- 結線ルールを 'random', 'serial', 'depthwise' から指定可能
- **batch_norm** (*bool*) -- BatchNormalization を有効にするか
- **binarize** (*bool*) -- 二値化出力を有効にするか
- **momentum** (*float*) -- BatchNormalization の momentum
- **gamma** (*float*) -- BatchNormalization の gamma
- **beta** (*float*) -- BatchNormalization の beta
- **N** (*int*) -- LUT の入力数
- **seed** (*int*) -- 変数初期値などの乱数シード
- **bin_dtype** (DType)) -- バイナリ出力の型を bb.DType.FP32 と bb.DType.BIT から指定 (bb.DType.BIT は binarize=True 時のみ)

W()

重み行列取得

コピーではなくスライス参照を得ており、本体の値を書き換え可能

戻り値 重み行列を指す Tensor

戻り値の型

W (*Tensor*)

dW()

重みの勾配行列取得

コピーではなくスライス参照を得ており、本体の値を書き換え可能

戻り値

重みの勾配を指す Tensor

戻り値の型

W (*Tensor*)

6.2.5.2 AverageLut クラス

class binarybrain.models.**AverageLut**(*output_shape=None*, *, *input_shape=None*, *name=None*, *N=6*, *connection='serial'*, *binarize=True*, *binarize_input=False*, *seed=1*, *bin_dtype=DType.FP32*, *real_dtype=DType.FP32*, *core_model=None*)

ベースクラス: SparseModel

AverageLut class

入力値の平均を出力する LUT 型のモデルバイナリの場合、bit をカウントして 1 の方が多ければ 1 を出力するテーブル固定の LUT と考える事ができる

パラメータ

- **output_shape** (*[int]*) -- 出力のシェイプ
- **connection** (*str*) -- 結線ルールを 'random', 'serial', 'depthwise' から指定可能
- **binarize** (*bool*) -- 二値化出力を有効にするか
- **binarize_input** (*bool*) -- 入力を二値化してから使うようにするか
- **N** (*int*) -- LUT の入力数
- **seed** (*int*) -- 変数初期値などの乱数シード
- **bin_dtype** (DType)) -- バイナリ出力の型を bb.DType.FP32 と bb.DType.BIT から指定 (bb.DType.BIT は binarize=True 時のみ)

6.2.5.3 BinaryLut クラス

class binarybrain.models.**BinaryLut**(*output_shape=None*, *, *input_shape=None*, *connection='random'*, *seed=1*, *name=None*, *N=6*, *fw_dtype=DType.FP32*, *bw_dtype=DType.FP32*, *core_model=None*)

ベースクラス: SparseModel

バイナリ LUT モデル

一般的な FPGA の LUT と同等の機能をするモデル。学習能力はなく、他のモデルで学習した結果をインポートしてモデル評価を行うためのもの

パラメータ

- **output_shape** (*List[int]*) -- 出力のシェイプ
- **connection** (*str*) -- 結線ルールを 'random', 'serial', 'depthwise' から指定可能 (未実装)
- **N** (*int*) -- LUT の入力数
- **seed** (*int*) -- 変数初期値などの乱数シード
- **fw_dtype** (DType)) -- forward の型を bb.DType.FP32 と bb.DType.BIT から指定

static from_sparse_model(*layer*, *, *fw_dtype=DType.FP32*)

他のモデルを元に生成

インポート元は入出力が同じ形状をしている必要があり、デジタル値の入力に対して出力を 0.5 を閾値として、バイナリテーブルを構成して取り込む

パラメータ

leyaer (Model) -- インポート元のモデル

import_layer(*leyaer*)

他のモデルからからインポート

インポート元は入出力が同じ形状をしている必要があり、デジタル値の入力に対して出力を 0.5 を閾値として、バイナリテーブルを構成して取り込む

パラメータ

leyaer (Model) -- インポート元のモデル

6.2.5.4 DenseAffine クラス

class binarybrain.models.**DenseAffine**(*output_shape=None*, *, *input_shape=None*, *initialize_std=0.01*, *initializer=''*, *seed=1*, *name=None*, *dtype=DType.FP32*, *core_model=None*)

ベースクラス: *Model*

DenseAffine class

普通の DenseAffine

パラメータ

- **output_shape** (*List[int]*) -- 出力のシェイプ
- **initialize_std** (*float*) -- 重み初期化乱数の標準偏差
- **initializer** (*str*) -- 変数の初期化アルゴリズム選択 (default/normal/uniform/He/Xavier)。
- **seed** (*int*) -- 変数初期値などの乱数シード

6.2.5.5 DenseAffineQuantize クラス

class binarybrain.models.**DenseAffineQuantize**(*output_shape=None*, *, *input_shape=None*, *quantize=True*, *weight_bits=8*, *output_bits=16*, *input_bits=0*, *weight_scale=0.00390625*, *output_scale=0.00390625*, *input_scale=0.00390625*, *initialize_std=0.01*, *initializer=''*, *seed=1*, *name=None*, *dtype=DType.FP32*, *core_model=None*)

ベースクラス: *Model*

DenseAffineQuantize class

DenseAffine に量子化のサポートを付けたもの

パラメータ

- **output_shape** (*List[int]*) -- 出力のシェイプ
- **initialize_std** (*float*) -- 重み初期化乱数の標準偏差
- **initializer** (*str*) -- 変数の初期化アルゴリズム選択 (default/normal/uniform/He/Xavier)。
- **quantize** (*bool*) -- 量子化の有効状態の初期値
- **weight_bits** (*int*) -- 重みの量子化 bit 数 (0 で量子化しない)
- **output_bits** (*int*) -- 入力の量子化 bit 数 (0 で量子化しない)

- **input_bits** (*int*) -- 出力の量子化 bit 数 (0 で量子化しない)
- **weight_scale** (*float*) -- 重みの量子化スケール (0 で bit 数に合わせる)
- **output_scale** (*float*) -- 入力の量子化スケール (0 で bit 数に合わせる)
- **input_scale** (*float*) -- 出力の量子化スケール (0 で bit 数に合わせる)
- **seed** (*int*) -- 変数初期値などの乱数シード

6.2.5.6 DepthwiseDenseAffine クラス

class binarybrain.models.**DepthwiseDenseAffine**(*output_shape=None*, ***, *input_shape=None*, *input_point_size=0*, *depth_size=0*, *initialize_std=0.01*, *initializer=''*, *seed=1*, *name=None*, *dtype=DType.FP32*, *core_model=None*)

ベースクラス: *Model*

DepthwiseDenseAffine class 普通の DepthwiseDenseAffine

パラメータ

- **output_shape** (*List[int]*) -- 出力のシェイプ
- **input_point_size** (*int*) -- 入力の point 数
- **depth_size** (*int*) -- depth サイズ
- **initialize_std** (*float*) -- 重み初期化乱数の標準偏差
- **initializer** (*str*) -- 変数の初期化アルゴリズム選択 (default/normal/uniform/He/Xavier)。
- **seed** (*int*) -- 変数初期値などの乱数シード

6.2.5.7 DepthwiseDenseAffineQuantize クラス

class binarybrain.models.**DepthwiseDenseAffineQuantize**(*output_shape=None*, ***, *input_shape=None*, *input_point_size=0*, *depth_size=0*, *quantize=True*, *weight_bits=8*, *output_bits=16*, *input_bits=0*, *weight_scale=0.00390625*, *output_scale=0.00390625*, *input_scale=0.00390625*, *initialize_std=0.01*, *initializer=''*, *seed=1*, *name=None*, *dtype=DType.FP32*, *core_model=None*)

ベースクラス: *Model*

DepthwiseDenseAffineQuantize class

DepthwiseDenseAffine に量子化のサポートを付けたもの

パラメータ

- **output_shape** (*List[int]*) -- 出力のシェイプ
- **input_point_size** (*int*) -- 入力の point 数
- **depth_size** (*int*) -- depth サイズ
- **weight_bits** (*int*) -- 重みの量子化 bit 数 (0 で量子化しない)
- **output_bits** (*int*) -- 入力の量子化 bit 数 (0 で量子化しない)
- **input_bits** (*int*) -- 出力の量子化 bit 数 (0 で量子化しない)
- **weight_scale** (*float*) -- 重みの量子化スケール (0 で bit 数に合わせる)
- **output_scale** (*float*) -- 入力の量子化スケール (0 で bit 数に合わせる)
- **input_scale** (*float*) -- 出力の量子化スケール (0 で bit 数に合わせる)
- **initialize_std** (*float*) -- 重み初期化乱数の標準偏差
- **initializer** (*str*) -- 変数の初期化アルゴリズム選択 (default/normal/uniform/He/Xavier)。
- **seed** (*int*) -- 変数初期値などの乱数シード

6.2.6 畳み込み/プーリング (Convolution and Pooling)

models モジュールの、畳み込みやプーリングなどのフィルタ演算を行うモデルです。

6.2.6.1 Convolution2d クラス

class binarybrain.models.**Convolution2d**(*sub_layer*, *filter_size=(1, 1)*, *stride=(1, 1)*, ***, *input_shape=None*, *padding='valid'*, *border_mode='reflect_101'*, *border_value=0.0*, *name=None*, *fw_dtype=DType.FP32*, *bw_dtype=DType.FP32*)

ベースクラス: *Sequential*

Convolution class

Lowering による畳み込み演算をパッキングするクラス

sub_layer で指定した演算レイヤーを畳み込み計算するためのモデル例えば sub_layer に DenseAffine レイヤーを指定すると一般的な CNN 用の畳み込み層となるが、BinaryBrain ではここに DifferentiableLut モデルを組み合わせて作った複合レイヤーを指定することで FPGA 化に適した畳み込み層を学習させることができる。

sub_layer で指定したサブレイヤーが im2col と col2im で挟み込まれ、一般に Lowering と呼ばれる方法で畳み込み演算が実行される

パラメータ

- **sub_layer** (Model) -- 畳み込みを行うサブレイヤー (このレイヤが im2col と col2im で挟み込まれる)
- **filter_size** (*(int, int)*) -- 2 次元のタプルでフィルタサイズを指定する
- **stride** (*(int, int)*) -- 2 次元のタプルでストライドサイズを指定する
- **batch_norm** (*bool*) -- BatchNormalization を有効にするか
- **padding** (*str*) -- パディングの方法を 'valid' と 'same' で指定する
- **border_mode** (*Border*) -- 'same' 時のボーダー処理を指定する
- **border_value** (*float*) -- 'same' 時のボーダー処理が CONSTANT の場合にパディング値を指定する
- **fw_dtype** (DType) -- forwar する型を bb.DType.FP32 と bb.DType.BIT から指定

backward(*dy_buf*)

Backward

モデルは学習の為に backword メソッドを持つ必ず forwad と対で呼び出す必要があり、直前の forward 結果に対する勾配計算を行いながら逆伝搬する

BinaryBrain は自動微分の機能を備えないので、backword の実装は必須である

パラメータ

dy_buf (FrameBuffer) -- 入力データ

戻り値

出力データ

戻り値の型

dx_buf (*FrameBuffer*)

dumps()

バイトデータにシリアライズ

モデルのデータをシリアライズして保存するためのバイト配列を生成

戻り値

Serialize data

戻り値の型

data (bytes)

forward(*x_buf*, *train=True*)

Forward

モデルは学習および推論の為に forward メソッドを持つ

train 変数を

パラメータ

- **x_buf** (FrameBuffer) -- 入力データ
- **train** (*bool*) -- True で学習、False で推論

戻り値

出力データ

戻り値の型

y_buf (*FrameBuffer*)

loads(*data*)

バイトデータをロード

モデルのデータをシリアライズして復帰のバイト配列ロード

パラメータ

data (*bytes*) -- Serialize data

send_command(*command*, *send_to='all'*)

コマンドの送信

モデルごとにコマンド文字列で設定を行うコマンド文字列の定義はモデルごとに自由である Sequential クラスなどは、保有する下位のモデルに再帰的にコマンドを伝搬させるので、複数の層に一括した設定が可能である受取先は send_to で送り先はインスタンス名やクラス名で制限でき 'all' を指定するとフィルタリングされない

パラメータ

- **command** (*str*) -- コマンド文字列
- **send_to** (*str*) -- 送信先

set_input_shape(*shape*)

入力シェイプ設定

BinaryBarain ではモデル生成時に入力のシェイプを決定する必要はなくネットワーク構築後に、ネットワークを構成する各モデルの set_input_shape を順に呼び出して形状を伝搬させることで各モデルの形状の設定を簡易化できる

set_input_shape が呼ばれるとそれまでの各層で保有する情報は保証されない。ネットワーク構築後に一度だけ呼び出すことを想定している

パラメータ

input_shape (*List[int]*) -- 入力シェイプ

戻り値

出力シェイプ

戻り値の型

output_shape (List[int])

6.2.6.2 MaxPooling クラス

class binarybrain.models.**MaxPooling**(*filter_size=(2, 2)*, ***, *input_shape=None*, *name=None*, *fw_dtype=DType.FP32*, *bw_dtype=DType.FP32*, *core_model=None*)

ベースクラス: *Model*

MaxPooling class

パラメータ

- **filter_size** (*(int, int)*) -- 2 次元のタプルでフィルタサイズを指定する
- **fw_dtype** (*DType*) -- forwar する型を bb.DType.FP32 と bb.DType.BIT から指定

6.2.6.3 StochasticMaxPooling クラス

class binarybrain.models.**StochasticMaxPooling**(*filter_size=(2, 2)*, ***, *input_shape=None*, *name=None*, *fw_dtype=DType.FP32*, *bw_dtype=DType.FP32*, *core_model=None*)

ベースクラス: *Model*

StochasticMaxPooling class

Stochastic 演算として OR 演算で Pooling 層を構成するモデル

パラメータ

- **filter_size** (*(int, int)*) -- 2 次元のタプルでフィルタサイズを指定する (現在 2x2 のみ)
- **fw_dtype** (DType)) -- forwar する型を bb.DType.FP32 と bb.DType.BIT から指定

6.2.6.4 UpSampling クラス

class binarybrain.models.**UpSampling**(*filter_size=(2, 2)*, ***, *fill=True*, *input_shape=None*, *name=None*, *fw_dtype=DType.FP32*, *bw_dtype=DType.FP32*, *core_model=None*)

ベースクラス: *Model*

UpSampling class

畳み込みの逆方向にアップサンプリングを行うモデル

パラメータ

- **filter_size** (*(int, int)*) -- 2 次元のタプルでフィルタサイズを指定する (現在 2x2 のみ)
- **fw_dtype** (DType)) -- forwar する型を bb.DType.FP32 と bb.DType.BIT から指定

6.2.7 活性化 (Activation)

models モジュールの 活性化層 (Activation 層 () を作るためのモデルです。

6.2.7.1 Binarize クラス

class binarybrain.models.**Binarize**(*, *input_shape=None*, *binary_th=0.0*, *binary_low=-1.0*, *binary_high=1.0*, *hardtanh_min=-1.0*, *hardtanh_max=1.0*, *name=None*, *bin_dtype=DType.FP32*, *real_dtype=DType.FP32*, *core_model=None*)

ベースクラス: *Model*

Binarize class

2 値化 (活性化層) backward は hard-tanh となる

パラメータ

bin_dtype (DType)) -- バイナリ型を bb.DType.FP32 と bb.DType.BIT から指定

6.2.7.2 Sigmoid クラス

class binarybrain.models.**Sigmoid**(*, *input_shape=None*, *name=None*, *bin_dtype=DType.FP32*, *real_dtype=DType.FP32*, *core_model=None*)

ベースクラス: *Model*

Sigmoid class

Sigmoid 活性化層 send_command で "binary true" とすることで、Binarize に切り替わる多値で学習を進めて、途中から Binarize に切り替える実験などが可能である

6.2.7.3 ReLU クラス

class binarybrain.models.**ReLU**(*, *input_shape=None*, *name=None*, *bin_dtype=DType.FP32*, *real_dtype=DType.FP32*, *core_model=None*)

ベースクラス: *Model*

ReLU class

ReLU 活性化層 send_command で "binary true" とすることで、Binarize に切り替わる多値で学習を進めて、途中から Binarize に切り替える実験などが可能である

6.2.7.4 HardTanh クラス

class binarybrain.models.**HardTanh**(*, *input_shape=None*, *name=None*, *bin_dtype=DType.FP32*, *real_dtype=DType.FP32*, *core_model=None*)

ベースクラス: *Model*

HardTanh class

HardTanh 活性化層 send_command で "binary true" とすることで、Binarize に切り替わる多値で学習を進めて、途中から Binarize に切り替える実験などが可能である

6.2.7.5 Softmax クラス

class binarybrain.models.**Softmax**(*, *input_shape=None*, *name=None*, *dtype=DType.FP32*, *core_model=None*)

ベースクラス: *Model*

Softmax class

Softmax 活性化層

6.2.8 補助モデル

models モジュールのその他のモデルです。

6.2.8.1 BatchNormalization クラス

class binarybrain.models.**BatchNormalization**(*, *input_shape=None*, *momentum=0.9*, *gamma=1.0*, *beta=0.0*, *fix_gamma=False*, *fix_beta=False*, *name=None*, *dtype=DType.FP32*, *core_model=None*)

ベースクラス: *Model*

BatchNormalization class

パラメータ

- **momentum** (*float*) -- 学習モーメント
- **gamma** (*float*) -- gamma 初期値
- **beta** (*float*) -- beta 初期値
- **fix_gamma** (*bool*) -- gamma を固定する (学習させない)
- **fix_beta** (*bool*) -- beta を固定する (学習させない)
- **bin_dtype** (*DType*)) -- バイナリ型を bb.DType.FP32 と bb.DType.BIT から指定

6.2.8.2 Dropout クラス

class binarybrain.models.**Dropout**(*, *rate=0.5*, *input_shape=None*, *seed=1*, *name=None*, *fw_dtype=DType.FP32*, *bw_dtype=DType.FP32*, *core_model=None*)

ベースクラス: *Model*

Dropout class

パラメータ

- **rate** (*float*) -- Drop 率
- **seed** (*int*) -- 乱数シード
- **fw_dtype** (DType) -- forward の型を bb.DType.FP32 と bb.DType.BIT から指定

6.2.8.3 Shuffle クラス

class binarybrain.models.**Shuffle**(*shuffle_unit*, *, *output_shape=None*, *input_shape=None*, *name=None*, *core_model=None*)

ベースクラス: *Model*

Shuffle class

所謂 ShuffleNet のようなシャッフルを行うモデル入力ノードが shuffle_unit 個のグループに分割されるようにシャッフルする

パラメータ

shuffle_unit (*int*) -- シャッフルする単位

6.2.9 最適化 (optimizer)

6.2.9.1 Optimizer クラス

class binarybrain.optimizer.**Optimizer**(*core_optimizer=None*)

ベースクラス: *Object*

Optimizer の基本クラス

set_learning_rate(*learning_rate*)

学習率設定

set_variables(*params*, *grads*)

変数設定

パラメータ

- **params** (Variables) -- 学習対象のパラメータ変数
- **grads** (Variables) -- params に対応する勾配変数

step()

パラメータ更新

set_variables で設定された勾配変数に基づいた学習を set_variables で設定されたパラメータ変数に適用する

update()

パラメータ更新＆勾配ゼロクリア

set_variables で設定された勾配変数に基づいた学習を set_variables で設定されたパラメータ変数に適用して、勾配をゼロクリアする

zero_grad()

勾配のゼロクリア

set_variables で設定された勾配変数をゼロクリアする

6.2.9.2 OptimizerSgd クラス

class binarybrain.optimizer.**OptimizerSgd**(*learning_rate=0.001*, *dtype=DType.FP32*)

ベースクラス: *Optimizer*

SGD 最適化クラス

パラメータ

learning_rate (*float*) -- 学習率

6.2.9.3 OptimizerAdaGrad クラス

class binarybrain.optimizer.**OptimizerAdaGrad**(*learning_rate=0.01*, *dtype=DType.FP32*)

ベースクラス: *Optimizer*

AdaGrad 最適化クラス

パラメータ

learning_rate (*float*) -- 学習率

6.2.9.4 OptimizerAdam クラス

class binarybrain.optimizer.**OptimizerAdam**(*learning_rate=0.001*, *beta1=0.9*, *beta2=0.999*, *dtype=DType.FP32*)

ベースクラス: *Optimizer*

Adam 最適化クラス

パラメータ

- **learning_rate** (*float*) -- 学習率
- **beta1** (*float*) -- beta1
- **beta2** (*float*) -- beta2

6.2.10 損失関数 (Loss functions)

6.2.10.1 LossFunction クラス

class binarybrain.losses.**LossFunction**(*core_loss=None*)

ベースクラス: *Object*

LossFunction class 損失関数の基底クラス

calculate(*y_buf*, *t_buf*, *mini_batch_size=None*)

損失の計算

mini_batch_size はメモリなどの都合でミニバッチをさらに分割する場合などに用いる。通常は None でよい。

パラメータ

- **y_buf** (FrameBuffer) -- forward 演算結果
- **t_buf** (FrameBuffer) -- 教師データ
- **mini_batch_size** (*int*) -- ミニバッチサイズ

戻り値 逆

伝搬させる誤差を返す

戻り値の型

dy_buf (*FrameBuffer*)

clear()

値のクリア

集計をクリアする。通常 epoch の単位でクリアして再集計を行う

get()

値の取得

戻り値 現

在までの損失値を返す

戻り値の型

loss(float)

6.2.10.2 LossMeanSquaredError クラス

class binarybrain.losses.**LossMeanSquaredError**(*reduction='mean', dtype=DType.FP32*)

ベースクラス: *LossFunction*

LossMeanSquaredError class

平均二乗誤差 (MSE) を計算して誤差として戻す

6.2.10.3 LossCrossEntropy クラス

class binarybrain.losses.**LossCrossEntropy**(*dtype=DType.FP32*)

ベースクラス: *LossFunction*

LossCrossEntropy class

交差エントロピー誤差を戻す

6.2.10.4 LossBinaryCrossEntropy クラス

class binarybrain.losses.**LossBinaryCrossEntropy**(*dtype=DType.FP32*)

ベースクラス: *LossFunction*

LossBinaryCrossEntropy class

2 値の交差エントロピー誤差を戻す

6.2.10.5 LossSoftmaxCrossEntropy クラス

class binarybrain.losses.**LossSoftmaxCrossEntropy**(*dtype=DType.FP32*)

ベースクラス: *LossFunction*

LossSoftmaxCrossEntropy class

Softmax(活性化) と CrossEntropy(損失関数) の複合両者を統合すると計算が簡略化される。

利用に際しては最終段に Softmax が挿入されたのと等価になるので注意すること。

6.2.10.6 LossSigmoidCrossEntropy クラス

class binarybrain.losses.**LossSigmoidCrossEntropy**(*dtype=DType.FP32*)

ベースクラス: *LossFunction*

LossSigmoidCrossEntropy class

Sigmoid(活性化) と BinaryCrossEntropy(損失関数) の複合両者を統合すると計算が簡略化される。

利用に際しては最終段に Sigmoid が挿入されたのと等価になるので注意すること。

6.2.11 評価関数 (Metrics functions)

6.2.11.1 Metrics クラス

class binarybrain.metrics.**Metrics**(*core_metrics=None*)

ベースクラス: *Object*

Metrics class 評価関数の基底クラス

calculate(*y_buf*, *t_buf*)

評価の計算

パラメータ

- **y_buf** (FrameBuffer) -- forward 演算結果
- **t_buf** (FrameBuffer) -- 教師データ

clear()

値のクリア

集計をクリアする。通常 epoch の単位でクリアして再集計を行う

get()

値の取得

戻り値

現在までの損失値を返す

戻り値の型

metrics(float)

get_metrics_string()

評価対象の文字列取得

評価関数ごとに評価値の単位が異なるため計算しているものの文字列を返す平均二乗誤差 (MSE) であったり、認識率 (accuracy) であったり get で得られる値を、表示やログで表す際に利用できる

パラメータ

metrics_string (*str*) -- 評価対象の文字列取得

6.2.11.2 MetricsMeanSquaredError クラス

class binarybrain.metrics.**MetricsMeanSquaredError**(*dtype=DType.FP32*)

ベースクラス: *Metrics*

MetricsMeanSquaredError class

平均二乗誤差の評価関数教師信号との平均二乗誤差を計算する

6.2.11.3 MetricsCategoricalAccuracy クラス

class binarybrain.metrics.**MetricsCategoricalAccuracy**(*dtype=DType.FP32*)

ベースクラス: *Metrics*

MetricsCategoricalAccuracy class

多クラス分類用の評価関数一致率を accuracy として計算する

6.2.11.4 MetricsBinaryCategoricalAccuracy クラス

class binarybrain.metrics.**MetricsBinaryCategoricalAccuracy**(*dtype=DType.FP32*)

ベースクラス: *Metrics*

MetricsBinaryCategoricalAccuracy class

2 クラス分類用の評価関数一致率を accuracy として計算する

6.2.12 保存／復帰 (Serialize)

6.2.12.1 storage モジュール

binarybrain.storage.**copy_network_files**(*data_path*, *dst_name*, *src_name*, *wildcard='*'*, *exist_ok=False*, *verbose=1*)

ネットワークファイルのコピー

binarybrain.storage.**get_latest_path**(*path: str*) → str

Get latest data path 最新のデータ保存パスを取得

パラメータ

path (*str*) -- 検索するパス

戻り値

見つかったパス. 見つからなければ None

binarybrain.storage.**get_load_networks_path**(*path: str*, *net*, *name=None*)

ネットワークをロードするパスを取得

binarybrain.storage.**is_date_string**(*text: str*)

Check if the string is a date データ保存パス用の日付文字列かどうか判定

パラメータ

text (*str*) -- 判定する文字列

戻り値

Boolean.

binarybrain.storage.**load_models**(*path: str*, *net*, ***, *read_layers: bool = False*, *file_format=None*, *verbose=True*)

load networks ネットを構成するモデルの保存

パラメータ

- **path** (*str*) -- 読み出すパス
- **net** (Model) -- 読み込むネット
- **read_layers** (*bool*) -- レイヤー別に読み込むか
- **verbose** (*bool*) -- 詳しく表示する

binarybrain.storage.**load_networks**(*path: str*, *net*, *name=None*, ***, *read_layers: bool = False*, *file_format=None*, *verbose=True*)

load network ネットを構成するモデルの読み込み

最新のデータを探して読み込み名前を指定した場合はそれを読み込み

パラメータ

- **path** (*str*) -- 読み込むパス
- **net** (Model) -- 読み込むネット
- **file_format** (*str*) -- 読み込む形式 (None がデフォルト)
- **verbose** (*bool*) -- 詳しく表示する

binarybrain.storage.**remove_backups**(*path: str*, *keeps: int = -1*)

Get latest data path 最新のデータ保存パスを取得

パラメータ

- **path** (*str*) -- 検索するパス
- **keeps** (*int*) -- 削除せずに残す数

binarybrain.storage.**save_models**(*path: str*, *net*, *, *write_layers=True*, *file_format=None*)

save networks ネットを構成するモデルの保存

パラメータ

- **path** (*str*) -- 保存するパス
- **net** (Model) -- 保存するネット
- **write_layers** (*bool*) -- レイヤー別にも出力するかどうか

binarybrain.storage.**save_networks**(*path: str*, *net*, *name=None*, *, *backups: int = 10*, *write_layers: bool = False*, *file_format=None*)

save networks ネットを構成するモデルの保存

指定したパスの下にさらに日付でディレクトリを作成して保存古いものから削除する機能あり名前を指定すれば日付ではなく名前で記録可能

パラメータ

- **path** (*str*) -- 保存するパス
- **net** (Model) -- 保存するネット
- **name** (*str*) -- 保存名 (指定しなければ日時で保存)
- **backups** (*int*) -- 残しておく古いデータ数 (-1 ですべて残す)
- **write_layers** (*bool*) -- レイヤー別に出力

戻り値 保

存名を返す

戻り値の型

name (str)

6.2.13 RTL(Verilog/HLS) 変換

学習が完了したネットは結果パラメータに基づいて、ユーザ側で自由に実装可能ですが、BinaryBrain でも若干のサポート関数を備えています。

binarybrain.verilog.**dump_verilog_lut_cnv_layers**(*f*, *module_name: str*, *net*, *device=''*)

dump verilog source of Convolutional LUT layers

畳み込み層を含むネットを AXI4 Stream Video 形式の Verilog ソースコードして出力する。縮小を伴う MaxPooling 層は最後に 1 個だけ挿入を許される

パラメータ

- **f** (*StreamIO*) -- 出力先ストリーム
- **module_name** (*str*) -- モジュール名
- **net** (Model) -- 変換するネット

binarybrain.verilog.**dump_verilog_lut_layers**(*f*, *module_name: str*, *net*, *device=''*)

dump verilog source of LUT layers 変換できないモデルは影響ない層とみなして無視するので注意

パラメータ

- **f** (*StreamIO*) -- 出力先ストリーム
- **module_name** (*str*) -- モジュール名
- **net** (Model) -- 変換するネット

binarybrain.verilog.**dump_verilog_readmemb_image_classification**(*f*, *loader*, *, *class_digits=8*)

verilog 用データダンプ verilog の $readmemb() での読み込み用データ作成

クラス ID + 画像データの形式で出力する

パラメータ

- **f** (*StreamIO*) -- 出力先
- **loader** (*Loader*) -- モジュール名
- **class_digits** (*int*) -- クラス分類の bit 数

binarybrain.verilog.**make_image_tile**(*rows*, *cols*, *img_gen*)

画像をタイル状に並べて大きな画像にする

学習用の c, h, w 順の画像データをタイル状に結合する

パラメータ

- **rows** (*int*) -- 縦の結合枚数
- **cols** (*int*) -- 横の結合枚数
- **gen** (*ndarray*) -- 画像を返すジェネレータ

戻り値

作成した画像

戻り値の型

img (ndarray)

binarybrain.verilog.**make_verilog_lut_layers**(*module_name: str*, *net*, *device=''*)

make verilog source of LUT layers 変換できないモデルは影響ない層とみなして無視するので注意

パラメータ

- **module_name** (*str*) -- モジュール名
- **net** (Model) -- 変換するネット

戻り値

Verilog source code (str)

binarybrain.verilog.**write_ppm**(*fname*, *img*)

ppm ファイルの出力

学習用の c, h, w 順の画像データを ppm 形式で保存する

パラメータ

- **fname** (*str*) -- 出力ファイル名
- **img** (*ndarray*) -- モジュール名

binarybrain.hls.**dump_hls_lut_layer**(*f*, *name*, *lut*)

dump HLS source of LUT layer

パラメータ

- **f** (*StreamIO*) -- 出力先ストリーム

- **name** (*str*) -- 関数名
- **lut** (Model) -- 変換するネット

binarybrain.hls.**make_hls_lut_layer**(*name*, *lut*)

make HLS source of LUT layer

パラメータ

- **name** (*str*) -- 関数名
- **lut** (Model) -- 変換するネット

戻り値

HLS source code (str)

6.2.14 システム／ GPU 関連 (System/GPU)

その他システム制御関連の API です

binarybrain.system.**get_cuda_driver_version**()

CUDA ドライババージョンの取得

戻り値

CUDA ドライババージョン

戻り値の型

driver_version (int)

binarybrain.system.**get_cuda_driver_version_string**()

CUDA ドライババージョン文字列の取得

戻り値

CUDA ドライババージョン文字列

戻り値の型

driver_version (str)

binarybrain.system.**get_device_count**()

利用可能なデバイス (GPU) の個数を確認

戻り値

利

用可能なデバイス (GPU) の個数を返す

戻り値の型

device_count (int)

binarybrain.system.**get_device_properties_string**(*device_id*)

現在のデバイス (GPU) の情報を入れた文字列を取得

パラメータ

device_id (*int*) -- 情報を取得するデバイス番号を指定

戻り値

現在のデバイス (GPU) の情報を入れた文字列を返す

戻り値の型

device_properties_string (str)

binarybrain.system.**get_version_string**()

バージョン文字列取得

戻り値

バージョン文字列

戻り値の型

version (str)

binarybrain.system.**is_device_available**()

デバイス (GPU) が有効かの確認

戻り値

デバイス (GPU) が利用可能なら True を返す

戻り値の型

device_available (bool)

binarybrain.system.**omp_set_num_threads**(*threads: int*)

omp_set_num_threads を呼び出すバックグランドで学習する場合など、Host 側の CPU をすべて使うと逆に性能が落ちる場合や、運用上不便なケースなどで個数制限できる

パラメータ

threads (*int*) -- OpenMP でのスレッド数

binarybrain.system.**set_device**(*device_id*)

利用するデバイス (GPU) を切り替え

パラメータ

device_id (*int*) -- 利用するデバイス番号を指定

binarybrain.system.**set_host_only**(*host_only: bool*)

ホスト (CPU) のみの指定

True を設定するとデバイス (GPU) を未使用としてホスト (CPU) のみを利用

パラメータ

host_only (*bool*) -- ホストのみの場合 True を指定

第 7 章

開発情報

7.1 github について

現在 version4 は下記の branch で管理しています

ver4_develop 開
: 発用ブランチです。ビルド不能な状態になることもあります。最新のコードにアクセスしたい場合はここをご覧ください。

ver4_release リ
: リース作成用ブランチです。

master リ
: リースブランチで確認したものを反映。

tag は リリースのタイミングでバージョン番号のタグを打つようにしております。また、開発都合で ver4_build0001 のような形式でリリースと無関係にビルドタグを打つ場合があります。

まだ、開発初期で仕様が安定していませんので、再現性の確保などが必要な際はタグを活用ください。

7.2 作者情報

渕上 竜司 (Ryuji Fuchikami)

- github : https://github.com/ryuz
- blog : http://ryuz.txt-nifty.com
- twitter : https://twitter.com/ryuz88
- facebook : https://www.facebook.com/ryuji.fuchikami
- web-site : http://ryuz.my.coocan.jp/

- e-mail : ryuji.fuchikami@nifty.com

7.3 参考にさせて頂いた情報

- バイナリニューラルネットとハードウェアの関係
  https://www.slideshare.net/kentotajiri/ss-77136469
- BinaryConnect: Training Deep Neural Networks with binary weights during propagations
  https://arxiv.org/pdf/1511.00363.pdf
- Binarized Neural Networks
  https://arxiv.org/abs/1602.02505
- Binarized Neural Networks: Training Deep Neural Networks with Weights and Activations Constrained to +1 or -1
  https://arxiv.org/abs/1602.02830
- XNOR-Net: ImageNet Classification Using Binary Convolutional Neural Networks
  https://arxiv.org/abs/1603.05279
- Xilinx UltraScale Architecture Configurable Logic Block User Guide
  https://japan.xilinx.com/support/documentation/user_guides/ug574-ultrascale-clb.pdf

7.4 参考にした書籍

- ゼロから作る Deep Learning ―Python で学ぶディープラーニングの理論と実装
  https://www.oreilly.co.jp/books/9784873117584/

第 8 章

Indices and tables

- genindex
- modindex
- search

Python モジュール索引

b

索引