BinaryBrain Documentation

リリース *3.13.0*

Ryuji Fuchikami

2020 年 12 月 31 日

Contents:

第 1 章

はじめに

1.1 概要

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 Sparse-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

1.2 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。

1.2.1 Windows

1. VisualStudio 2017 と CUDA 10.1　をインストールします
2. git clone –recursive -b ver3_release https://github.com/ryuz/BinaryBrain.git
3. http://yann.lecun.com/exdb/mnist/ から MNIST データをダウンロードします
4. ダウンロードした　MNSIT　データを "samplesmnist" に展開します
5. VisuaStudio にて "samplesmnistsample_mnist.sln" を開きます
6. "x64 Release"　のターゲットにてビルドします
7. run

1.2.2 Linux(Ubuntu 18.04.1)

1.2.2.1 1. install tools

```
% sudo apt update
% sudo apt upgrade
% sudo apt install git
% sudo apt install make
% sudo apt install g++
% # sudo apt install nvidia-cuda-toolkit
% wget http://developer.download.nvidia.com/compute/cuda/10.1/Prod/local_installers/
↪cuda_10.1.243_418.87.00_linux.run
% sudo sh cuda_10.1.243_418.87.00_linux.run
```

1.2.2.2 2. build and run

```
% git clone --recursive -b ver3_release  https://github.com/ryuz/BinaryBrain.git
% cd BinaryBrain/samples/mnist
% make
% make dl_data
% ./sample-mnist All
```

1.2.3 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 ver3_release  https://github.com/ryuz/BinaryBrain.git
%cd BinaryBrain/samples/mnist
!make all
!make run
```

1.3 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。

1.3.1 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です。

```
% pip3 install binarybrain==3.13.*
```

Windows など環境によっては pip3 が存在せず、pip のみ場合は pip3 を pip に置き換えて実行ください。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)

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/master/python/samples

1.3.2 setup.py でのインストール

1.3.2.1 事前準備

必要なパッケージを事前にインストールください

```
% 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
```

1.3.2.2 インストール

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
```

1.4 github について

現在 version3 は下記の branch で管理しています

ver3_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。

ver3_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 開発都合で ver3_build0001 のような形式で定期的に打っており、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。(以前はリリースごとに ver3_release1 のような形で打つようにしていました)。

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。

1.5 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 などで出力して、FPGA 化することを目的に作成しています。

もちろん BinaryBrain 自体は学習によってネットワークのパラメータも求めるまでが主体ですので、その結果を使って C 言語を出力するルーチンをユーザー側で開発することも自由です。

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、import するだけで利用可能です。

使い方は sample などを参考にしてください。

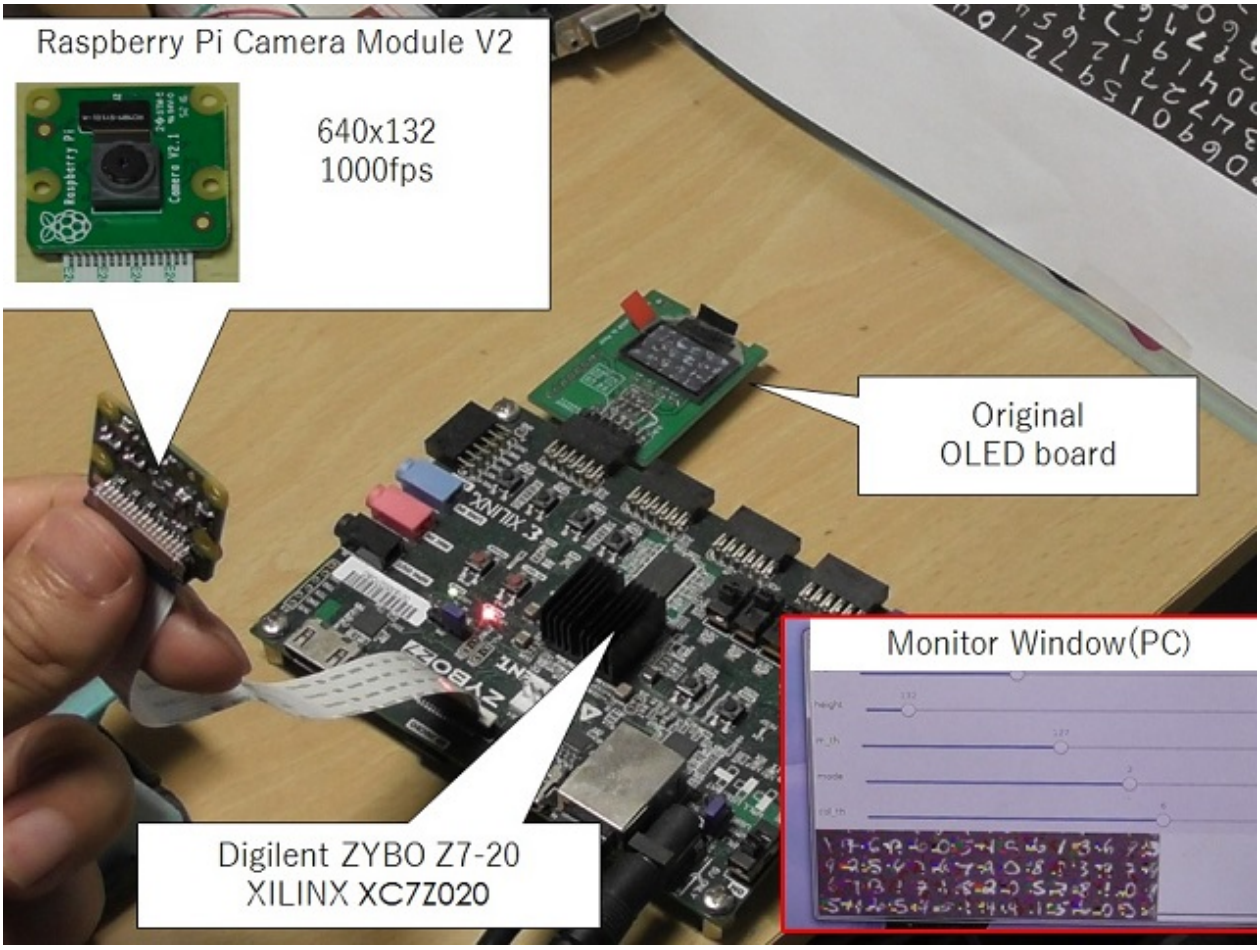
第 2 章

事例紹介

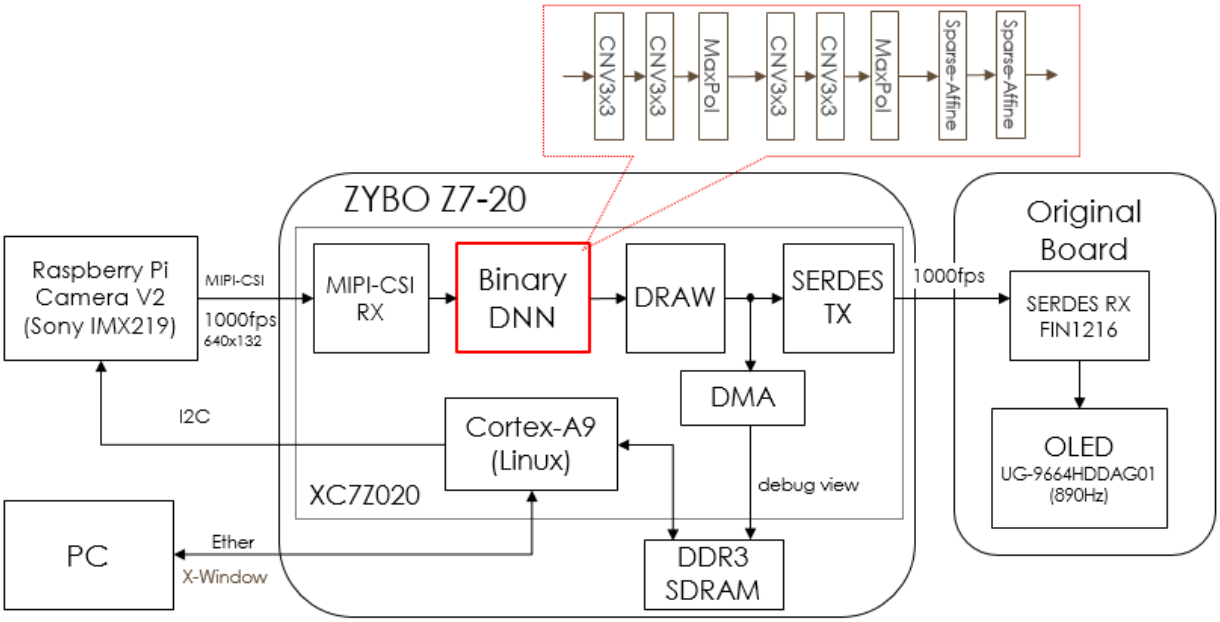
2.1 リアルタイム認識

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。



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。

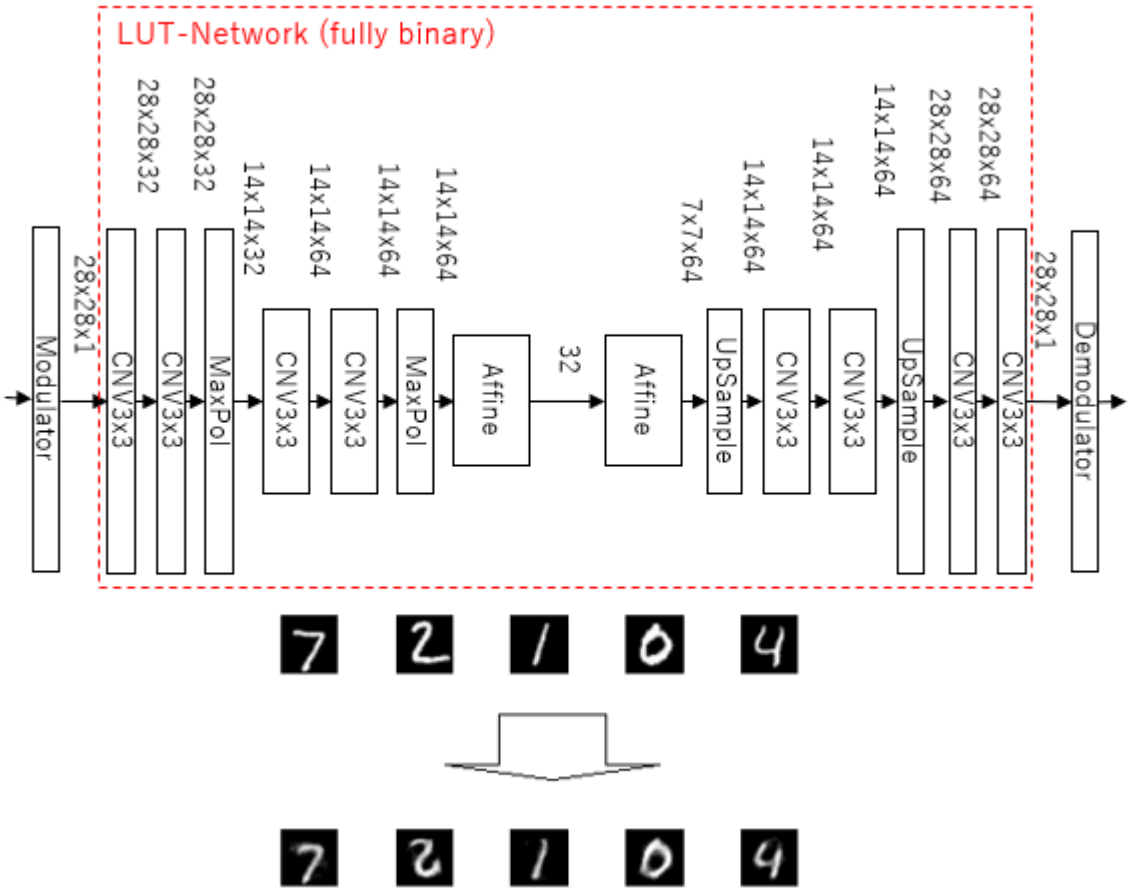
2.2 Autoencoder

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。

2.2.1 MNIST

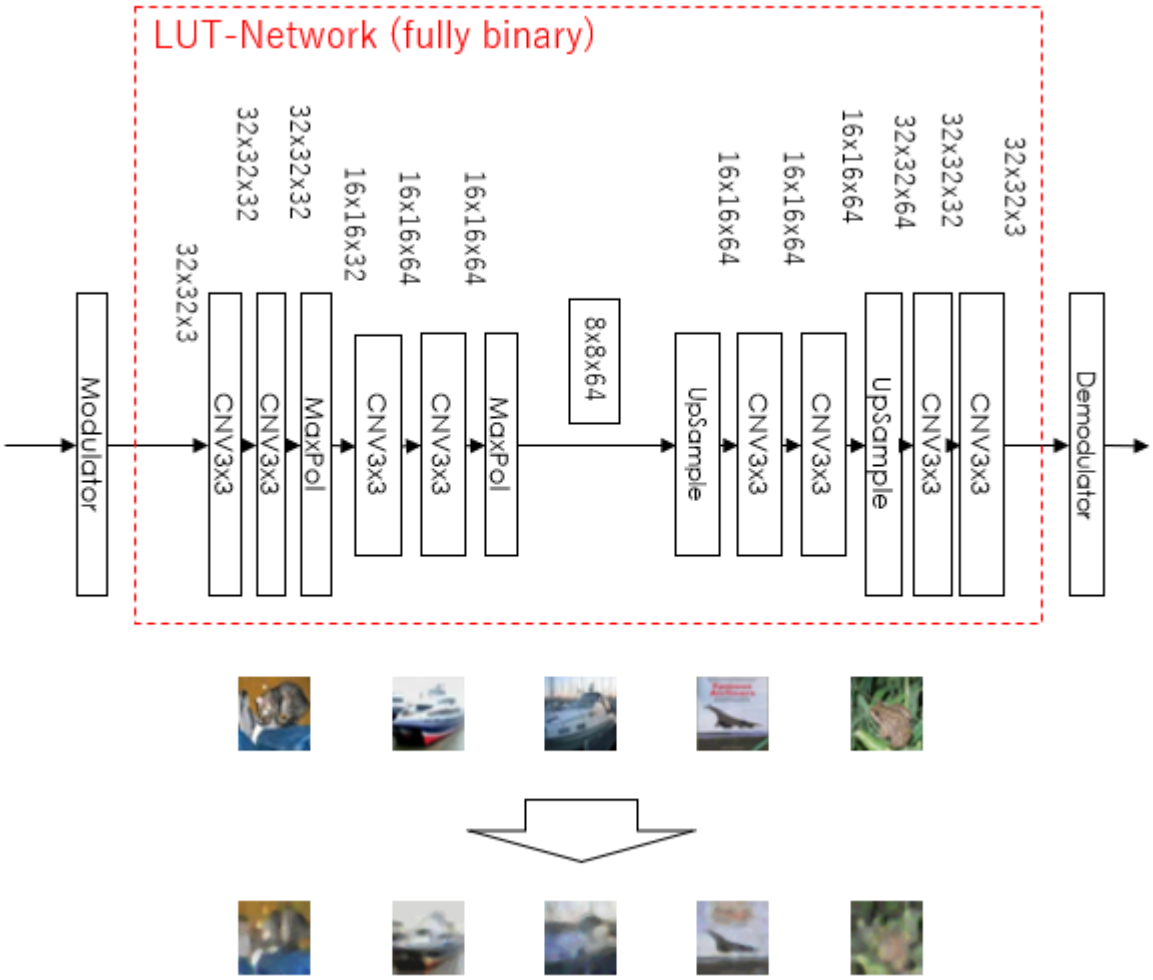
MNIST での Autoencoder の実験結果です。



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。

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。

第 3 章

RTL の試し方

3.1 sample の動かし方

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/tree/master/samples/mnist/verilog

の readme.txt を参照ください。

第 4 章

C++ API

4.1 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。

4.2 モデルクラス

4.2.1 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。

4.2.1.1 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。

4.2.2 活性化層

4.2.2.1 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。

4.2.2.2 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。

4.2.2.3 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。

4.2.3 演算層

4.2.3.1 SparseLutN 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。パーセプトロンと異なる独自のモデルを用いており、単体で XOR パターンを含めた LUT で表現可能な空間すべてを効率的に学習可能です。

4.2.3.2 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 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、SparseLut もでるよりも高速に学習させることが可能です。

4.2.3.3 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。

4.2.3.4 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。

4.2.3.5 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。

4.2.3.6 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。

4.2.3.7 MaxPooling クラス

MaxPooling 層です。

4.2.3.8 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。

4.2.3.9 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。

4.2.4 補助層

4.2.4.1 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。

4.2.4.2 LoweringConvolution 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。

4.2.4.3 ConvolutionIm2 クラス

畳み込みの為の Lowering を行います。通常、LoweringConvolution 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。

4.2.4.4 ConvolutionCol2Im クラス

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

4.2.4.5 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。

4.2.4.6 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。

4.2.4.7 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。

4.3 モデル以外のクラス

4.3.1 損失関数

4.3.1.1 LossSoftmaxCrossEntropy クラス

普通の Softmax-CrossEntropy クラスです。

4.3.1.2 "

平均二乗誤差を損失とするクラスです。

4.3.2 評価関数

4.3.2.1 MetricsCategoricalAccuracy クラス

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

4.3.2.2 MetricsMeanSquaredError クラス

MSE(平均二乗誤差) を評価値とするクラスです。

4.3.3 最適化 (Optimizer)

4.3.3.1 OptimizerSgd クラス

普通の SGD です。

4.3.3.2 OptimizerAdam クラス

普通の Adam です。

4.3.4 実行補助

4.3.4.1 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。

4.3.5 データ保持

4.3.5.1 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。

4.3.5.2 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。

4.3.5.3 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。

4.4 各種関数

4.4.1 FPGA へのエクスポート

4.4.1.1 ExportVerilog_LutLayers 関数

LutLayer を Verilog-RTL で出力します。

4.4.1.2 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 5 章

Python API

5.1 binarybrain module

第 6 章

LUT-Network とは

6.1 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（超リアルタイム）
- 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。

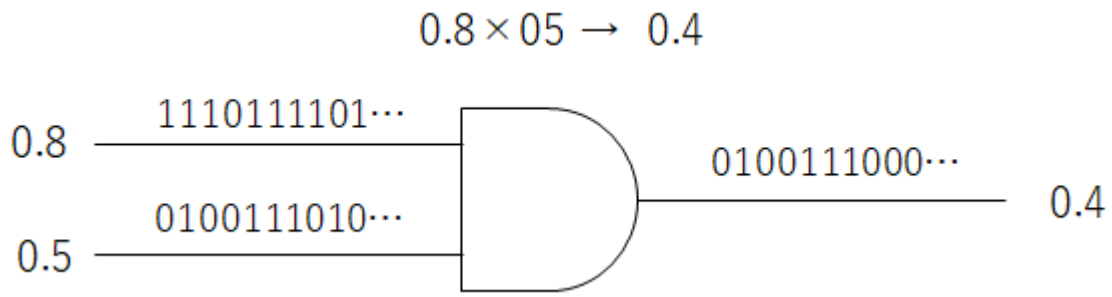
6.2 Sparse-LUT モデル

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。驚くことに LUT によるテーブル参照を微分可能なモデルで表現し、これを実現しています。

6.2.1 Stochastic-LUT モデル

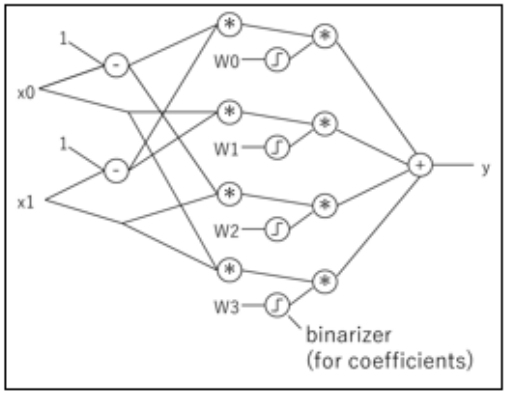
その計算モデルは、LUT のテーブル引きの回路演算を Stoachstic 演算に置き換えて実験しているときに発見されました。Stochastic-LUT モデルは Sparse-LUT モデルの中にその一部として含まれています。

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

$$y = W_0 \cdot \overline{x_1}\ \overline{x_0} + W_1 \cdot \overline{x_1}\ x_0 + W_2 \cdot x_1\ \overline{x_0} + W_3 \cdot x_1\ x_0$$

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

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

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。

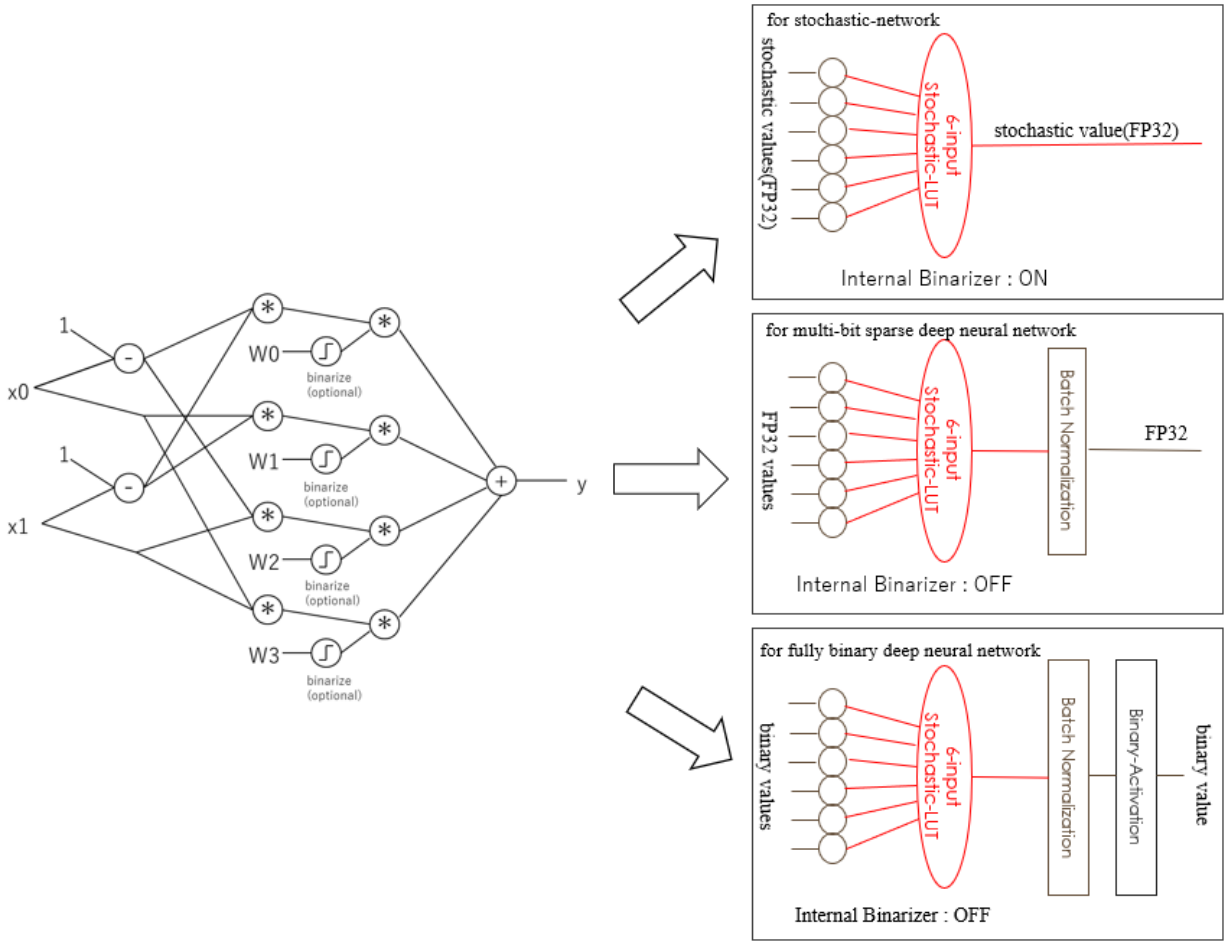
このモデルを使った学習は、入力同士に相関がなく、純粋に確率値として扱える範疇において、正しく機能し、出力値が一定の確率で 1 を出力するようにネットワーク全体を学習させることが可能です。

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。

6.2.2 Sparse-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、Sparse-LUT モデルを提唱しています。

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

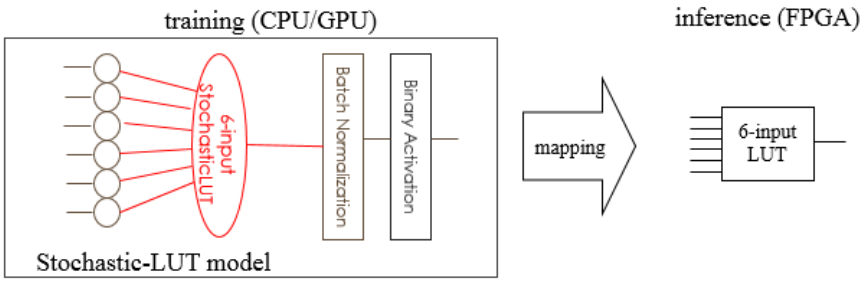


- Stochastic 値を扱うネットワークを学習可能
- 非バイナリ (FP32 など) の疎結合ネットワークで従来のパーセプトロンよりも高性能に機能
- バイナリ疎結合ネットワークで LUT に置換可能なモデルとして学習可能

6.2.3 Sparse-LUT モデルによる FPGA 化

Sparse-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。



6-input Stochastic-LUT

$$y = \mathrm{StochasticLUT}(x_0, x_1, \ldots x_5)$$

$$\begin{aligned} y \leftarrow\ & W_0 \cdot \overline{x_5}\ \overline{x_4}\ \overline{x_3}\ \overline{x_2}\ \overline{x_1}\ \overline{x_0} \\ & + W_1 \cdot \overline{x_5}\ \overline{x_4}\ \overline{x_3}\ \overline{x_2}\ \overline{x_1}\ x_0 \\ & + W_2 \cdot \overline{x_5}\ \overline{x_4}\ \overline{x_3}\ \overline{x_2}\ x_1\ \overline{x_0} \\ & + W_3 \cdot \overline{x_5}\ \overline{x_4}\ \overline{x_3}\ \overline{x_2} x_1\ x_0 \\ & \vdots \\ & + W_{61} \cdot x_5\ x_4\ x_3\ x_2\ \overline{x_1}\ x_0 \\ & + W_{62} \cdot x_5\ x_4\ x_3\ x_2\ x_1 \overline{x_0} \\ & + W_{63} \cdot x_5\ x_4\ x_3\ x_2\ x_1\ x_0 \end{aligned}$$

Batch Normalization

$$y_i = \mathrm{BatchNorm}(x_i)$$

$$\mu \leftarrow \frac{1}{m}\sum_{i=1}^{m} x_i$$

$$\sigma^2 \mu \leftarrow \frac{1}{m}\sum_{i=1}^{m} (x_i - \mu)^2$$

$$\hat{x}_i \leftarrow \frac{x_i - \mu}{\sqrt{\sigma^2 + \epsilon}}$$

$$y_i \leftarrow \gamma \hat{x}_i + \beta \equiv \mathrm{BatchNorm}(x_i)$$

$\epsilon = 1.0\mathrm{e}{-7},\ \gamma = 0.3,\ \beta = 0.5$

Binary Activation

[forward]

$$y = \mathrm{Binarize}(x)$$

$$y \leftarrow \begin{cases} 1 & (x > 0.5) \\ 0 & (x \le 0.5) \end{cases}$$

[backward]

$$y = \mathrm{hard{-}tanh}(x)$$

$$y \leftarrow \begin{cases} 1 & (x > 1) \\ 0 & (x < 0) \\ x & (otherwise) \end{cases}$$

第 7 章

バイナリ変調

7.1 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。

7.2 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。

7.3 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。

BinaryBrain では全く同じことをフルバイナリのニューラルネットに行うことで、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。

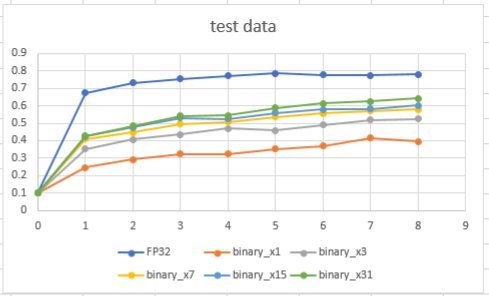
下記は、通常の Dense CNN の ReLU を Binarizer に置き換え、入力もバイナリ化してフルバイナリネットワーク化したものを用いて、バイナリ変調の効果を実験した結果です。

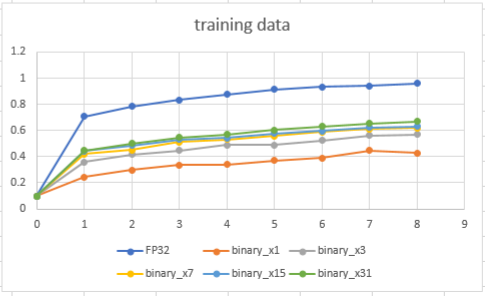
test data

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





Input 32x32x3
Cnv3x3 64
Cnv3x3 64
MaxPol
Cnv3x3 128
Cnv3x3 128
MaxPol
Dense 512
Dense 10

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 8 章

Indices and tables

- genindex
- modindex
- search