Akida CNN2SNN Toolkit

©brainchip*

Version 1.1 March 2019

USA:
BrainChip Inc.
65 Enterprise
Aliso Viejo, CA 92656
and
1801 Wedemeyer Street, Suite 304
San Francisco, CA 94129
United States
Tel: +1 (949) 330-6750

Australia:
BrainChip Holdings Ltd.
c/o Boardroom Pty Limited
Level 12 225 George Street
Sydney, NSW 2000
Tel: +61 2 8016 2841
Fax: +61 2 9279 0664

Europe, Middle East, and Africa:
BrainChip SAS
20 Avenue Prat Gimont
31130 Balma
France
Tel: +33 5 6100 9145

Contents

Overview and Product Description

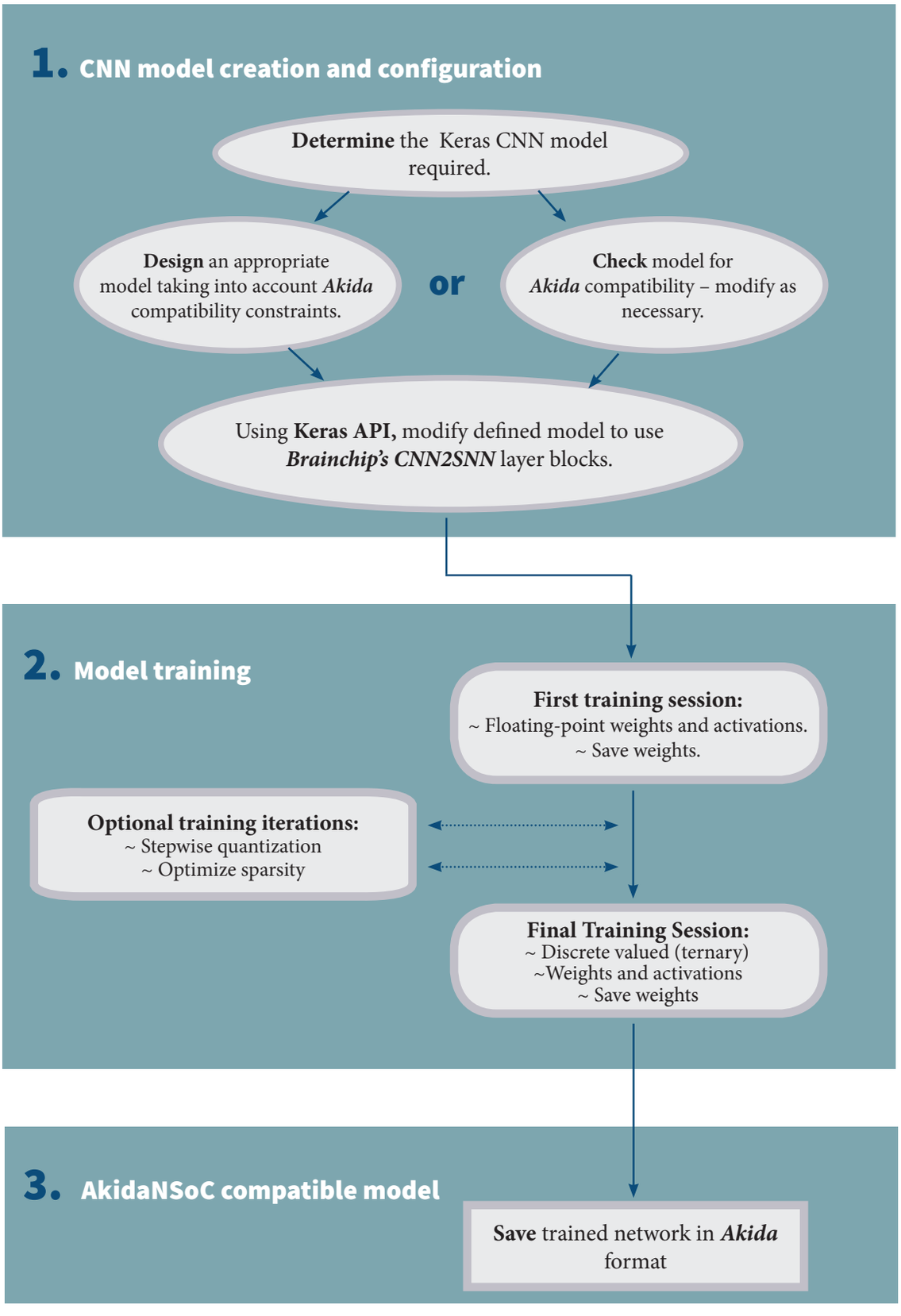
The **Brainchip*** CNN2SNN Toolkit provides a means to convert Artificial Neural Networks that were trained using Deep Learning methods to a low-latency and low-power Spiking Neural Network (SNN) for use with the Akida Neuromorphic System on a Chip (NSoC). This document is a detailed guide to that process. Working examples are provided in the Akida Development Environment (ADE) as Jupyter Notebooks.

The Akida NSoC provides Spiking Neural Networks (SNNs)* in which communications between neurons take the form of "spikes" or impulses that are generated when a neuron exceeds a threshold level of activation. Neurons that do not cross the threshold generate no output and contribute no further computational cost downstream. This feature is key to the efficiency of the Akida NSoC. The Akida NSoC further extends this efficiency by operating with low bitwidth "synapses" or weights of connections between neurons. Despite the apparent fundamental differences between SNNs and CNNs, the underlying mathematical operations performed by each may be rendered identical. Consequently, the trained parameters of a CNN can be converted to be compatible with those of the Akida NSoC, given only a small number of constraints. **
By careful attention to specifics in the architecture and training of the CNN, an overly complex conversion step from CNN to SNN can be avoided. The CNN2SNN Toolkit comprises of a set of functions designed for the popular Keras/tensorflow framework, making it easy to train a SNN compatible network.

* *Please refer to the Akida Execution Engine documentation pdf for details.*

** *Typically, for the Akida NSoC – ternary connection weights and binary activations*

CNN to SNN Conversion Workflow



Workflow for generating a deep-learning trained network for implementation in the Akida NSoC

Constraints for Akida NSoC Compatibility

When designing a model from scratch, or converting an existing model to obtain a final model compatible with the Akida NSoC, consider compatibility at these two distinct levels:

- Weights must be ternary (-1/0/+1) and activations must be binary (0/+1). This is the definitive step for preparing a CNN as a SNN. **Brainchip*** provides easy-to-use custom functions for this purpose. The use of **tf.Keras layers** is described in the next chapter.
- Modification or substitution of a layer will be required when a method that is used in the CNN is not compatible with the Akida architecture. For instance, this can be the case for residual connections.

The Overall Model

Serial, feedforward arrangement only

- No parallel layers
- No “residual” connections

Supported Layer Types:

Core Neural Layers

- Dense
- Conv2D

Specialized Layers

- DepthwiseConv2D
- Depthwise_conv_block

Support Layers

- BatchNormalization
- MaxPooling2D
- Dropout/Noise Layers

The Layer Blocks Concept

In Keras, when adding a core layer type (`Dense` or `Conv2D`) to a model, an activation function is typically included:

```
x = Dense(64, activation='relu')(x)
```

or the equivalent, explicitly adding the activation function separately:

```
x = Dense(64)(x)
x = Activation('relu'))(x)
```

It is very common for other functions to be included in this arrangement, e.g., a normalization of values before applying the activation function:

```
x = Dense(64)(x)
x = BatchNormalization()(x)
x = Activation('relu')(x)
```

This particular arrangement of layers is the key building block of Akida-compatible CNNs and is reflected in the way custom quantization operations are made available in the CNN2SNN toolkit, i.e., the following code snippet sets up the same trio of layers as those above. The BatchNormalization function is required because of the way Brainchip's activation function works; this will be detailed in the Technical Appendix at a later stage.

```
x = dense_block(x, 64)
```

Note that this is not simply convenient modularization of the code. Every `Dense` or `Conv` layer in an Akida-compatible CNN must be followed by `BatchNormalization` and `Activation` layers, and this arrangement is enforced by use of the provided `quantization_blocks:`

quantization_block =
Conv/Dense + BatchNorm + Activation

Exception: The final output layer does not need an Activation function, use: `block activ_quantization=None`
Exception: The DepthwiseConv block is a special case described under the heading 'DepthwiseConv' in this document.

The Layer Blocks Concept (continued)

The specific characteristics of the `dense_block, conv_block` and `depthwise_conv_block` are covered under their specific headings below.

Note these points concerning layer types and their implementation:

- The option of including MaxPooling2D layers is directly built into the provided `conv_block` and `depthwise_conv_block` modules.
- Note that only a pooling size of 2x2 is allowed. To use this option, simply set the 'pooling' argument as required, i.e., `pooling=(2, 2)`
- The Akida NSoC does not support pooling operations with a stride different from the pooling size. GlobalMaxPooling2D and GlobalAveragePooling2D layers are also permitted and should be added as normal in the Keras model definition – that is, added as standard Keras layers, and not integrated into the provided `conv_block` or `depthwise_conv_blocks.`

Training-only Layers (Dropout/Noise, etc.)

The Akida NSoC is used in CNN conversion for inference only. Training is done within the Keras environment and training-only layers may be added at will, such as Dropout or Noise layers. These are handled fully by Keras during the training and do not need to be modified to be Akida-compatible for inference. They should be added as standard Keras layers, and are not integrated into the provided `conv_block` or `depthwise_conv_blocks`.

BatchNormalization

As noted above, every major processing layer (Dense/Conv2D/ Depthwise block) must be accompanied by a BatchNormalization layer as part of the quantization process. This configuration is built into the provided block definitions.

This description applies to the Keras model definition only. As regards the implementation within the Akida NSoC: it may be helpful to understand that the associated scaling operations (multiplication and shift) are never performed within the Akida NSoC during inference. The computational cost is reduced by wrapping the batch normalization function and quantized activation function into the spike generating threshold of the Akida SNN.

That process is completely transparent to the User. It does, however, have an important consequence for the output of the final layer of the network; see "Final Layers" below.

First Layers

In the CNN2SNN framework, the first layer must be a convolutional layer. This is translated to a specialized Akida-compatible layer that can handle 8-bit input values with 1 (grayscale) or 3 (RGB) channels, as opposed to the 'spike' event inputs required by other Akida-compatible layer types. Note that the Akida NSoC does support Dense (or 'Fully Connected') type entry layers, but only for spike-event type inputs.

The padding in this first convolution layer can be of type 'valid' (no padding) or 'same' (zero-padding, such that the output is the same size as the input).

Depending on the rescaling of the inputs used for training, a 'same' type convolution here can lead to a slight performance cost; using a 'valid' type convolution, if possible, is recommended. See the "cost to performance" section below.

Input Rescaling: The first Akida-compatible convolutional layer receives 8-bit input values. If the data is already in 8-bit format it can be sent to Akida NSoC inputs without rescaling.

First Layers (continued)

For the Akida NSoC to compensate for the rescaled values that were used during training, it is necessary to provide the coefficients used for rescaling. The neuron firing thresholds in the first layer will be set to be appropriate for the input data. This applies to the common case where input data are natively 8-bit. Where that is not the case, the process is more complex, and we recommend applying rescaling in two steps:

1. Taking the data to an 8-bit range suitable for the Akida NSoC. Apply this step both for training and for inference
2. Rescaling the 8-bit values to some unit or zero centered range suitable for CNN training, as above. This step should only be applied for the CNN training. Also, remember to provide those rescaling coefficients when converting the trained network to an Akida-compatible format.

Final Layers

As is typical for CNNs, the final layer of a model should not include the standard activation nonlinearity. Rather, the Batch Normalized outputs of the final layer should be passed to the appropriate layer according to the loss function used for training the network, e.g., a softmax activation layer for a `categorical_crossentropy` loss function.
This is the only step where the Akida-compatible implementation of the trained network diverges (mathematically) from the Keras model definition.
The main Akida hardware implements no floating-point multiplication operations and does not include the final BatchNormalization or any subsequent (e.g., softmax) transformations.

Final Layers (continued)

The values returned by the Akida NSoC are the activation levels in the final processing (Conv/Dense) layer. In most cases, taking the maximum among these values is sufficient to obtain the correct response from the network.
However, if there is a difference in performance between the Keras and the Akida compatible implementations of the model, it is likely be at this step. In which case, the Keras level performance can be recovered by applying the final transformations on activation levels returned from the Akida NSoC

The Layer Block Modules

The layer block modules provided are: `conv_block, depthwise_conv_block` and `dense_block.`
Most of the parameters for these blocks are identical to those used when defining a model using standard Keras layers. However, each of the provided blocks includes a pair of parameters that are essential to preparing a network for the Akida NSoC. They are: `weight_quantization` and `activ_quantization`.

These are the parameters that enable you to implement and control **Brainchip's** quantization methods that quantize the relevant aspects of the model (weights and activations) down to the low bitwidth values required for implementation in the Akida NSoC.

`weight_quantization` controls the weights and can take one of two values:

1. **'float':** Weights behave exactly as in a standard Keras implementation, i.e., they are floating-point values.
2. **'ternary':** Weights are discretized to ternary values (-1/0/+1). This is the final configuration for all blocks prior to conversion to the Akida NSoC.

The Layer Block Modules (continued)

The weights in the Keras model are always stored as floating-point values and the back-propagated adjustments during training are applied to those values. The quantizing modifier is only applied during the forward pass (during inference) and those modified values are transferred to the Akida NSoC once training is completed.

`activ_quantization` controls the activation function applied and can take one of three values:

1. **'relu6':** Applies the well-known 'relu6' type activation function.
2. **'binary':** Applies a custom activation function, such that the outputs take binary values (0/+1). This is the final configuration for all blocks prior to conversion to the Akida NSoC (with the exception of the final block as below).
3. **None:** No activation layer will be added. This should be used for the final block in your model. The final layer will be a BatchNormalization layer, with output suitable for certain loss functions, e.g., `squared_hinge` loss function. Alternatively, you can add a different activation function directly after the final block, typically a `soft_max` layer; suitable for a `categorical_crossentropy` loss function, for example.

The restriction to using the 'relu6' type standard activation layer is required because, after rigorous testing of that model configuration, it is known that this provides a good starting point when applying the binary activation function. If a different activation function is found to be a better starting point, add that option by editing the layer block code. However, there is then no guarantee that replacement with the binary activation function will afterwards yield efficient training; testing will be required.

The rationale for providing these quantization parameters is that we recommend preparing your model for the Akida NSoC in two (or more) training episodes.

In the first episode, train with the standard Keras versions of the weights and activations.

The Layer Block Modules (continued)

In which case, your model definition would look like:

```
x = conv_block(x, filters=32,
          weight_quantization='float',
          activ_quantization='relu6')
```

Once it is established that the overall model configuration prior to quantization yields a satisfactory performance on the task, proceed with quantization suitable for the Akida NSoC, changing only the quantization parameters:

```
x = conv_block(x, filters=32,
     weight_quantization='ternary',
     activ_quantization='binary')
```

We recommend saving the weights of your trained non-quantized model and using those to initialize the quantized version; typically leading to both faster convergence and better final performance values. See the provided tutorials for examples.
Also, it is possible to proceed with quantization in a series of smaller steps. For example, it may be beneficial to train, first with all standard/floating-point values, then retrain with binary activations, and then, finally, with ternary weights. It is possible to subdivide even further – going step by step through the individual blocks and even modifying the targeted network sparsity* to achieve an optimal trade-off between accuracy and speed.
See the chapter on Optimizing Quantization for the Akida NSoC for full details.

**Sparsity refers to the fraction of both weights and activations with value zero.*

The Layer Block Modules (continued)

conv_block

```
def conv_block(inputs, filters,
     block_id=0,
     kernel=(3, 3),
     padding='valid',
     strides=(1, 1),
     weight_quantization='ternary',
     activ_quantization='binary',
     inittype='glorot',
     scale=1,
     addmaxpool=False,
     pool_size=(1, 1),
     alpha=1)
```

See the Python code docstrings for a full description of the parameters.

Akida-specific Layer Constraints and considerations:

- Stride 1 only: To adapt an existing network with a higher convolution stride, we suggest substituting a convolution with stride 1 followed by a pooling step of the appropriate size and stride.
- After final training, all blocks must use: `weight quantization='ternary', activ quantization='binary'`

dense_block

```
dense_block(inputs, units,
     block_id=0,
     weight_quantization='ternary',
     activ_quantization='binary',
     inittype='glorot')
```

See the Python code docstrings for a full description of the parameters.

The Layer Block Modules (continued)

dense_block (continued)

Akida-specific Layer Constraints and considerations:

- After final training, all blocks must use `weight_quantization='ternary', activ_quantization='binary'`
- Exception: The final layer block should not include a quantizing activation function, i.e., use `weight_quantization='ternary', activ_quantization=None.`
- This block can optionally be followed by an activation function appropriate for your loss function, e.g., a soft max layer. See the Final Layers section above.

depthwise_conv_block

```
depthwise_conv_block(inputs, pointwise_conv_filters,
     alpha=1,
     depth_multiplier=1,
     strides=(1, 1),
     block_id=1,
     weight_quantization='ternary',
     activ_quantization='binary',
     inittype='glorot',
     scale=1,
     addmaxpool=False,
     pool_size=(1, 1))
```

See the Python code docstrings for a full description of the parameters.

The Layer Block Modules (continued)

depthwise_conv_block (continued)

The `depthwise_conv_block` is an implementation of a depthwise separable convolution. Specifically, it comprises a depthwise convolutional layer (`DepthwiseConv2D`) followed by a pointwise convolutional layer (`Conv2D with kernel size 1x1`). The published literature (Sheng, et al., 2018) and our own experiments suggest that applying a low bitwidth nonlinearity here is harmful to performance, and so the default **Brainchip*** depthwise `conv_block` includes no nonlinearity between the depthwise and pointwise convolutional layers; this configuration is reflected in the Akida NSoC. It is, however, necessary to quantize the output of the block as a whole, i.e., the pointwise convolutional layer is followed by batch normalization and activation layers, and the activation function should be set to binary for the final training run.

Akida-specific Layer Constraints and Considerations:
After final training, all blocks must use:
`weight quantization='ternary', activ_quantization='binary'`

Quantization Walkthrough

The tutorial examples provide the best introduction to the practice of quantization (Jupyter Notebooks). For reference, the following provides a short description of the appropriate steps:

MNIST Example.

1. Starting Model

The following is a sequential model as defined within the tensorflow Keras framework, appropriate for the MNIST dataset:

```
img_input = Input(shape=(28, 28, 1))
x = Conv2D(filters=32,
     kernel_size=(5, 5),
     padding='valid',
     use_bias=False,
     data_format='channels_last')(img_input)
x = MaxPooling2D(pool_size=(2, 2), strides=(2, 2),
padding='valid')(x)
x = BatchNormalization()(x)
x = ReLU(6.)(x)

x = Conv2D(filters=64,
     kernel_size=(5, 5),
     padding='valid',
     use_bias=False)(x)
x = MaxPooling2D(pool_size=(2, 2), strides=(2, 2),
padding='valid')(x)
x = BatchNormalization()(x)
x = ReLU(6.)(x)

x = Flatten()(x)
x = Dense(512,
     use_bias=False)(x)
x = BatchNormalization()(x)
x = ReLU(6.)(x)
x = Dense(10,
     use_bias=False)(x)
x = BatchNormalization()(x)

model = Model(img_input, x, name='mnistnet')
```

MNIST Example. (continued)

2. Akida-compatibility model check and changes

Before proceeding, make sure that the model includes no layers or operations that are incompatible with the Akida NSoC, e.g., parallel layers or residual connections.

3. Redefine model using Brainchip Layer Blocks

To apply the **Brainchip*** quantization functions, re-define the model using the provided layer blocks.
The following definition is identical to the model above:

```
from cnn2snn.quantization_blocks import conv_block,
dense_block

# Define the model.
# The commented code shows the sets of layers in the
original definition
# that are being replaced by the provided conv_block and
dense_blocks here

img_input = Input(shape=(28, 28, 1))

x = conv_block(img_input, filters=32, block_id=0,
     kernel=(5, 5),
     padding='valid',
     weight_quantization='float',
     activ_quantization='relu6',
     addmaxpool=True)
x = conv_block(x, filters=64, block_id=1,
     kernel=(5, 5),
     padding='valid',
     weight_quantization='float',
     activ_quantization='relu6',
     addmaxpool=True)

x = Flatten()(x)

x = dense_block(x, units=512, block_id=2,
     weight_quantization='float',
     activ_quantization='relu6')

x = dense_block(x, units=10, block_id=3,
     weight_quantization='float',
     activ_quantization=None)

model = Model(img_input, x, name='mnistnet')

opt = Adam(lr=lr_start)
model.compile(loss='squared_hinge', optimizer=opt,
metrics=['acc'])
model.summary()
```

MNIST Example. (continued)

3. Redefine model using Brainchip Layer Blocks (continued)

So far, the model is still defined with the original (floating-point) weight and activation values. It is important now to run training with this model and to save the trained weights.

```
[run training …]
model.save_weights('float_pretrain_mnist.hdf5')
```

This serves two purposes:

1. It verifies that the (non-quantized) model you have defined can deliver satisfactory performance on the target task. The subsequent processes of low-bitwidth quantization will almost always lead to a slight drop in performance. Ensure that your starting point is satisfactory before investing time in the next steps.
2. It pre-trains the weights as floating-point values; using those to initialize the quantized version of the network generally leads to a slightly higher final performance level and a much faster convergence.

4. Re-train the model, applying quantization

To apply the critical quantization operations, use the same model definition as above, but replace:

```
weight_quantization='float'
```

with

```
weight_quantization='ternary'
```

and:

```
activ_quantization='relu6'
```

with

```
activ_quantization='binary'
```

Remember to leave the `activ_quantization` in the final layer as `None`

After compiling the model, initialize the weights with the pre-trained floating-point values:

```
model.load_weights(' float_pretrain_mnist.hdf5')
```

MNIST Example. (continued)

5. Convert the trained network for the Akida NSoC

The final step is a one-line command that converts the trained model into a saved Akida-compatible model definition.
Note that it is necessary to provide the coefficients used for rescaling the inputs prior to training within Keras. The `inputRescaling` variable passed, should be a dictionary with two values, `a` and `b`, used to prepare the training data such that: $data_{train} = (data_{raw} - b)/a$.

This example, provided in the tutorial, should clarify this step:

```
from cnn2snn.Keras2akida import saveAkidaNetwork
saveAkidaNetwork(model, inputRescaling, 'ak_mnist/
ak.yml')
```

References

Courbariaux, M., Bengio, Y., and David, J.-P. (2015). BinaryConnect: Training Deep Neural Networks with binary weights during propagations. ArXiv151100363 Cs.

Deng, L., Jiao, P., Pei, J., Wu, Z., and Li, G. (2017). Gated XNOR Networks: Deep Neural Networks with Ternary Weights and Activations under a Unified Discretization Framework. ArXiv170509283 Cs Stat.

Zhou, S., Wu, Y., Ni, Z., Zhou, X., Wen, H., and Zou, Y. (2016). DoReFa-Net: Training Low Bitwidth Convolutional Neural Networks with Low Bitwidth Gradients. ArXiv160606160 Cs.