Chapter 1

Binary Neural Networks (BNNs)

1.1 Overview

Binary (or Binarized) Neural Networks have emerged as compact, yet powerful, neural network architectures for embedded deep learning. At their core, BNNs operate with binary weights and activations rather than full precision values as in typical deep neural networks. This fundamental shift not only simplifies computations but also introduces a range of advantages that address challenges in modern AI systems.

The goal of this chapter is to introduce Binary Neural Networks and provide an understanding as to how these types of networks function. It will begin by motivating the development of BNNs with a discussion of the benefits these networks provide in regards to efficiency and memory consumption. Next, the basic operations and concepts behind BNNs success will be explained to give readers an idea of how binary values are created and used within the network. Following this, the chapter will move onto the two main stages involved in model inference and training, forward and backward propagation, and discuss how BNNs address the challenges that arise as a result of using only binary values.

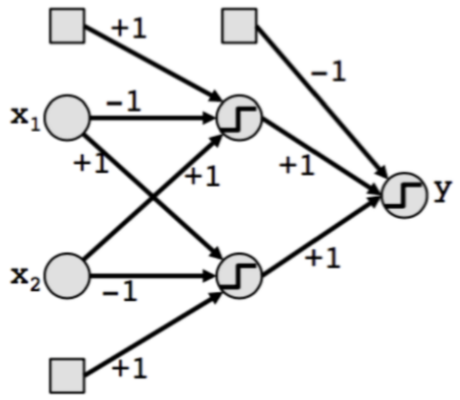


Figure 1.1: High-level depiction of BNN [12]

1.2 Motivation

The development of BNNs has largely been driven by the need for more efficient architectures, particularly in resource-constrained environments. Because traditional deep neural networks use floating-point precision, the significant computational resources (i.e. many GPUs) and memory capacities required for training and storing the model are often on a scale that is infeasible for small devices. BNNs, however, are able to drastically reduce the computational complexity and memory footprint by employing binary values and bitwise operations, making them particularly well-suited for deployment in edge devices, IoT applications, and other scenarios where resources are limited.

The operational mechanisms of BNNs involve specialized binary operations including XNOR and population count (pop count) which facilitate efficient forward and backward propagation, the details of which will be discussed in the sections that follow. By swapping normal matrix multiplication with these bitwise operations, BNNs act as a binary circuit and consequently, are particularly hardware-friendly and ideal for deployment in FPGAs and other hardware devices.

Compare, for instance, a 32 bit floating point value which is used in most deep neural networks to a single bit used in BNNs. The single bit representation uses 1/32 of the space required by the 32 bit representation making it 32 times more memory efficient. Further, bitwise operations can typically be executed in a single clock cycle in hardware as opposed to the tens of cycles often required for computation with floating point numbers, offering significant speedup. Such improved efficiency translates to accelerated inference times and enables BNNs to be used for applications requiring real-time decision-making.

1.3 Binary Values

The binary values in BNNs can be either one of the following combinations:

- 0 and 1
- -1 and 1

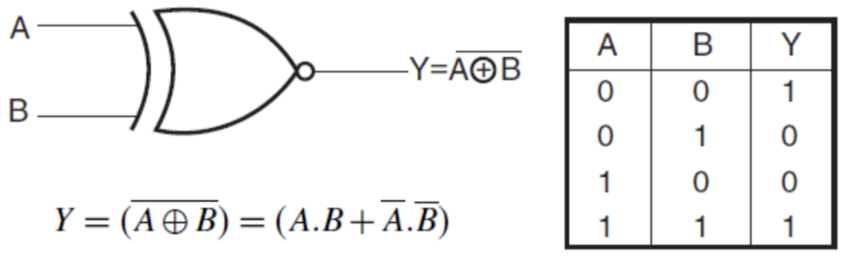
However, -1 and 1 is much more common in practice.

Can you guess why this may be the case? Think about how logical operation, specifically XNOR, could relate to multiplication. The answer will become clear in the sections that follow.

1.4 Binary Operations

1.4.1 XNOR

The XNOR (exclusive NOR) gate is a digital logic gate that acts as an XOR (exclusive or) operation followed by an inverter. As seen in the table below its output is "true" if the inputs are the same and "false" otherwise. In the context of BNNs, the XNOR operation is applied element-wise between binary inputs and binary weights. The resulting binary output indicates whether the input and weight have the same or different sign.



$$Y = (\overline{A \oplus B}) = (A.B + \overline{A}.\overline{B})$$

A	B	Y
0	0	1
0	1	0
1	0	0
1	1	1

Figure 1.2: XNOR logic gate and truth table.

It was mentioned previously that using -1 and 1 as the binary values in a BNN was preferred over 0 and 1. The reason for this lies in the XNOR operation. Consider two binary inputs $A \in \{0, 1\}$ and $B \in \{0, 1\}$ and perform the XNOR operation on these values. Then, map 0 to -1 and 1 to +1 so that $A \in \{-1, +1\}$ and $B \in \{-1, +1\}$ and perform multiplication on these values.

XNOR	**Multiplication**
$\overline{0 \bigoplus 0} = 1 \rightarrow +1$	$-1 * -1 = +1$
$\overline{0 \bigoplus 1} = 0 \rightarrow -1$	$-1 * +1 = -1$
$\overline{1 \bigoplus 0} = 0 \rightarrow -1$	$+1 * -1 = -1$
$\overline{1 \bigoplus 1} = 1 \rightarrow +1$	$+1 * +1 = +1$

Notice that the output for these two operations is the same. Thus, when using binary values of -1 and +1 the XNOR operation is simply multiplication, demonstrating how we can replace costly arithmetic operations with bitwise operations in BNNs.

1.4.2 Population Count

The population count (popcount) operation counts the number of bits set to 1 in a binary vector. In BNNs, pop count operations help determine binary activations during the forward pass by quantifying the number of positive weights in a given layer.

1.4.3 XNOR and Popcount for Matrix Multiplication

In combination, XNOR and popcount operations can replace normal MAC (multiply-accumulate) operations in BNNs. Below is the general formula for multiplying two binary vectors A and B using XNOR and popcount.

$$A \cdot B = sum - (N - sum) = 2 * sum - N \tag{1.1}$$

where,

$$sum = popcount(\overline{A \oplus B}) \tag{1.2}$$

Here, sum is the number of +1's in the dot product result, N is the total number of bits of the XNOR result, and $N - sum$ is the number of -1's in the dot product result.

Example

Let's take a look at an example to convince you of the equivalence of matrix multiplication and the XNOR popcount operations.

Consider two input vectors $[10, -10, -5, 9, -8, 2, 3, 1, -11]$ and $[12, -18, -13, -13, -14, -15, 11, 12, 13]$. We first binarize these using the sign function (defined in the next section) to convert our values to -1 or +1 and obtain $[1, -1, -1, 1, -1, 1, 1, 1, -1]$ and $[1, -1, -1, -1, -1, -1, 1, 1, 1]$. Then we perform both matrix multiplication (dot product) and XNOR with popcount and compare the results. Note that in order to use XNOR the values were first quantized to 0s and 1's. As depicted in the graphic on the next page, it is clear that both operations result in the same final output value of 3.

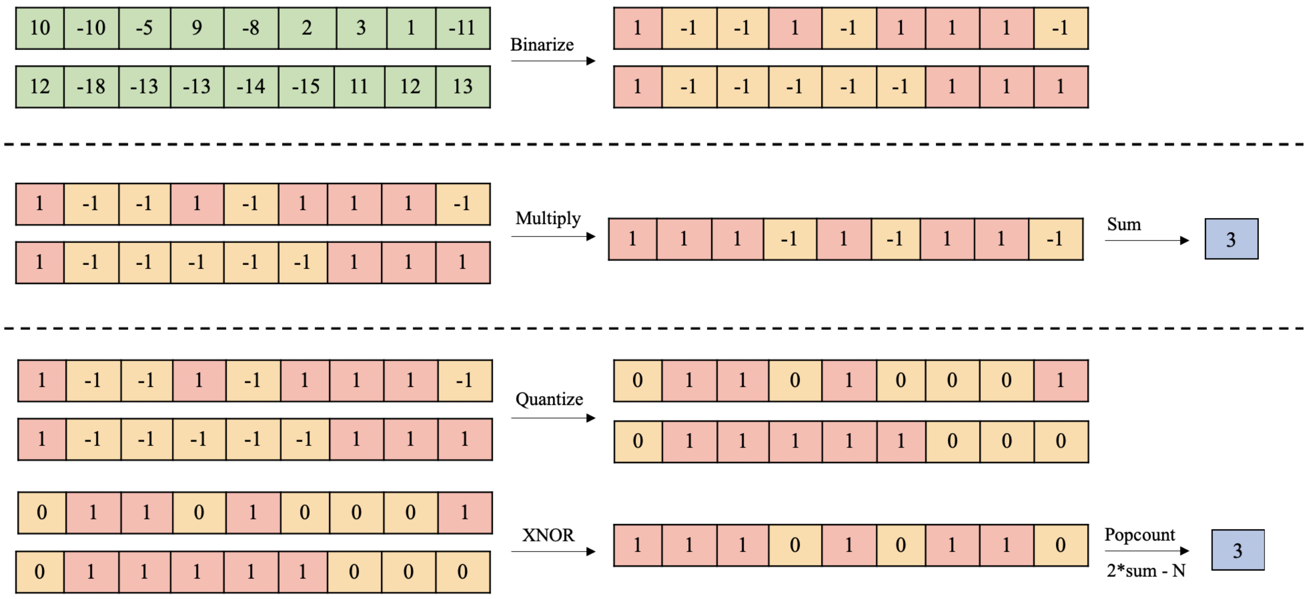


Figure 1.3: Example of normal MAC vs XNOR and popcount operations.

1.5 Binarization of the Network

In order to achieve the benefits of using binary values in-place of floating point values discussed above, we need a way to "binarize" our network. There are two binarization functions that can be used to convert the neural network's floating point values into binary values of -1 or 1.

The first is the deterministic signum or sign function which, as the name suggests, maps real-values to -1 or 1 based on their sign. The sign function is defined as:

$$x^b = Sign(x) = \begin{cases} +1 & \text{if } x \geq 0 \\ -1 & \text{otherwise} \end{cases} \tag{1.3}$$

where x^b is the binarized weight or activation and x is the original floating point value.

The second binarization function is stochastic and is defined as:

$$x^b = Sign(x) = \begin{cases} +1 & \text{with probability} p = \sigma(x) \\ -1 & \text{with probability } 1 - p \end{cases} \tag{1.4}$$

where σ is:

$$\sigma(x) = max(0, min(1, \frac{x+1}{2}) \tag{1.5}$$

Although the stochastic binarization can be esspecially effective in dealing with variability in data as compared to the deterministic case, it is harder to implement as it requires the hardware to generate random bits when quantizing. As a result, the deterministic sign function is more often used in practice and is what should used in your BNN project.

What classification tasks would perform better with stochastic binarization? Could we achieve the same performance with deterministic binarization if we add more layers?

1.6 Forward Propagation

Recall that in deep learning, forward propagation refers to model inference. That is, given an input, use the model's learned parameters to predict the output. For BNNs, the forward pass is similar to that of any typical deep neural networks architecture with the exception that all weights and activations are binarized to either -1 or +1.

1.6.1 General BNN Forward Propagation Psuedo Code

The pseudo code for forward propagation of a BNN is shown below. Note that the superscript b on a variable indicates that that variable is binarized.

For layers 1 through L, the weights are first binarized (and quantized) using the sign function. This step is only necessary during training, however, since once the model is fully trained only the binary weights will be stored. Then, our binary activations from the previous layer and binary weights are multiplied using the XNOR and popcount operations resulting in a non-binary sum. Following this, batch normalization is applied, if necessary, with parameters θ_k. If it is not the last layer of the network, then activations are once again binarized and the process is repeated.

```
for $k = 1$ to $L$ do
    $W_k^b \leftarrow \text{Binarize}(W_k)$
    $s_k \leftarrow a_{k-1}^b W_k^b$
    $a_k \leftarrow \text{BatchNorm}(s_k, \theta_k)$
    if $k < L$ then
        $a_k^b \leftarrow \text{Binarize}(a_k)$
    end if
end for
```

Figure 1.4: Pseudo code for BNN forward propagation

1.6.2 BNN Forward Propagation Implementation using XNOR and Popcount

Below you will find a sample python implementation of the forward pass function provided as part of the BNN class project. This network takes a 28x28 MNIST image as input and classifies the digit depicted in the images from 0-9. Since the model was already trained, stored weights ("fc1w_qntz", "fc2w_qntz", "fc3w_qntz") are binary. There are only three layers to this BNN, with each implementing the process described in the pseudo code above: (1) Inputs to each layer (i.e. activations from the previous layer) are binarized and quantized.(2) These inputs are then multiplied with the binary weights using XNOR + popcount.(3) The output is fed to the next layer or used to make the final prediction. For the project, you will be tasked will implementing the feed_forward_quantized function in HLS.

```
def feed_forward_quantized(self, input):
    #param input: MNIST sample input

    # layer 1
    X0_input = self.quantize(self.sign(self.adj(input)))
    layer1_output = self.matmul_xnor(X0_input, self.fc1w_qntz.T)
    layer1_activations = (layer1_output * 2 - 784)

    # layer 2
    layer2_input = self.sign(layer1_activations)
    layer2_quantized = self.quantize(layer2_input)
    layer2_output = self.matmul_xnor(layer2_quantized, self.fc2w_qntz.T)
    layer2_activations = (layer2_output * 2 - 128)

    # layer 3
    layer3_input = self.sign(layer2_activations)
    layer3_quantized = self.quantize(layer3_input)
    layer3_output = self.matmul_xnor(layer3_quantized, self.fc3w_qntz.T)

    final_output = (layer3_output * 2 - 64)
    A = np.array([final_output], np.int32)

    return A
```

1.6.3 BNN Forward Propagation Implementation Using MAC

For comparison, we have included a BNN forward propagation implementation using multiply and accumulate (MAC) rather than the XNOR and popcount introduced above. We hope it is now clear that this implementation is significantly more costly to implement in hardware.

```
def feed_forward(self, input):
     """This BNN using normal MAC.
    """
    # layer 1
    X0_q = self.sign(self.adj(input))
    X1 = np.matmul(X0_q, self.fc1w_q.T)

    # layer 2
    X1_q = self.sign(X1)
    X2 = np.matmul(X1_q, self.fc2w_q.T)

    # layer 3
    X2_q = self.sign(X2)
    X3 = np.matmul(X2_q, self.fc3w_q.T)

    return X3
```

1.6.4 BNN Forward Propagation Implementation in C++, Quantized

Below we have included the feed_forward_quantized function in C++, for use in HLS.

```
#include <iostream>
#include <cmath>

int XNOR(int a, int b) {
    return (a == b) ? 1 : 0;
}

void matmul_xnor(int* A, int* B, int* res, int rowsA, int rowsB, int colsB) {
    for (int x = 0; x < colsB; ++x) {
        int cnt = 0;
        for (int y = 0; y < rowsA; ++y) {
            cnt += XNOR(A[y], B[y * colsB + x]);
        }
        res[x] = cnt;
    }
}

int quantize(int x) {
    return (x == 1) ? 0 : 1;
}

int sign(int x) {
    return (x > 0) ? 1 : -1;
}

void feed_forward_quantized(int* X, int* w1, int* w2,
                            int X_size,
                            int rowsW1, int colsW1,
                            int rowsW2, int colsW2,
                            int* layer1_activations, int* layer2_activations) {

    // ********** Layer 1 **********
    // Quantize inputs
    int X0_input[X_size];
    for (int i = 0; i < X_size; ++i) {
        X0_input[i] = quantize(sign(X[i]));
    }
    // Perform matrix multiplication with W1 using XNOR
    matmul_xnor(X0_input, w1, layer1_activations, X_size, rowsW1, colsW1);
    for (int i = 0; i < colsW1; ++i) {
        layer1_activations[i] = (layer1_activations[i] * 2 - X_size);
    }

    // ********** Layer 2 **********
    // Quantize layer 1 activations
    int layer2_quantized[colsW1];
    for (int i = 0; i < colsW1; ++i) {
        layer2_quantized[i] = quantize(sign(layer1_activations[i]));
    }
    // Perform matrix multiplication with W2 using XNOR
    matmul_xnor(layer2_quantized, w2, layer2_activations, colsW1, rowsW2, colsW2);
    for (int i = 0; i < colsW2; ++i) {
        layer2_activations[i] = (layer2_activations[i] * 2 - colsW1);
    }

    // final output is layer2_activations

}

int main() {
    // Example usage:
    int input[] = { 1, 1 };
    int W1[] = {1, 1, 1, 1};
    int W2[] = {1, 0, 1, 1, 1, 0};
    int X_size = 2;
    int rowsW1 = 2;
    int colsW1 = 2;
    int rowsW2 = 2;
    int colsW2 = 3;
    int layer1_activations[colsW1];
    int layer2_activations[colsW2];


    feed_forward_quantized(input, W1, W2, X_size, rowsW1, colsW1, rowsW2, colsW2,
                            layer1_activations, layer2_activations);

    return 0;
}
```

1.7 Back Propagation

Backpropagation (short for "backwards propagation of errors") is a fundamental algorithm used to train supervised learning algorithms - BNNs being one of them. At the end of the forward pass (forward propagation), the network computes the loss (error) by comparing its outputs to the ground truth values using a loss function such as mean squared error for regression tasks or cross-entropy for classification tasks. The algorithm then calculates the gradient of the loss function with respect to each weight in the network by applying the chain rule.

Can you think of any issue with taking the gradient here? Remember in forward propagation, we used Sign(x) to binarize weights. What does the derivative look like for this function?

1.7.1 Gradient Decent Update Rule

Backpropagation gets its name from the fact that this chain rule process starts at the output layer and works backward through the network. The weights are then updated to minimize the loss (error) function, shown below with W the weight to update, J the loss function, and α the learning rate. This update process is called gradient decent.

This process of forward and backward propagation is repeated for many iterations, updating the weights of the network each time to minimize the loss function. Below is a graphical depiction of BNN training.

Remember that weights are binarized in forward propagation. Below is a graph of the Sign(x) function used for deterministic binarization. Note that the derivative of the Sign function is zero almost everywhere, which makes it seem incompatible with the backward propagation method described above, where all gradients would equal zero. So how do BNNs get around this issue?

$$W_{new} = W_{old} - \alpha \underbrace{\frac{dJ}{dW}}_{\text{gradient}}$$

Figure 1.5: Gradient decent update rule.

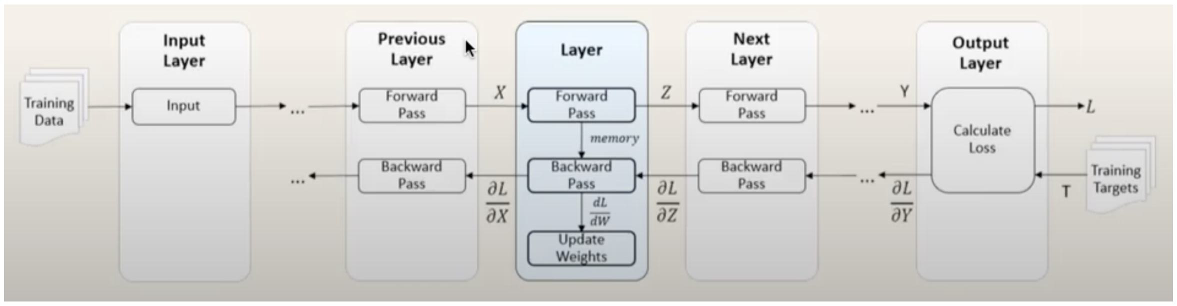


Figure 1.6: Overview of BNN training, with an emphasis on forward and backward propagation.

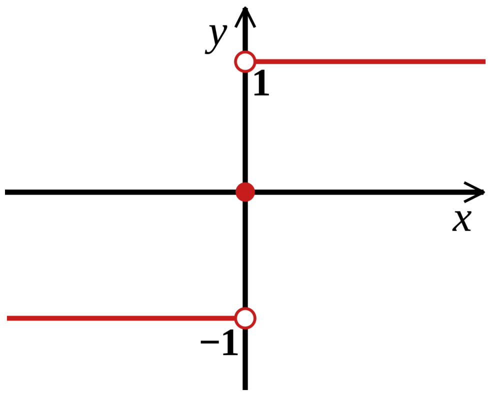
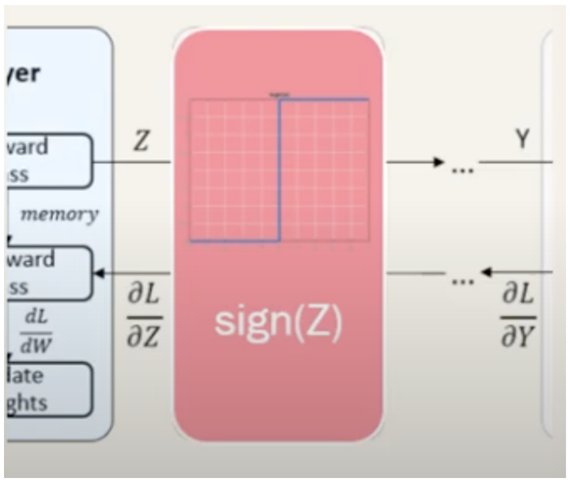


Figure 1.7: Plot of Sign(x) for deterministic binarization. Note that the derivative of Sign(x) with respect to x is zero for nearly all x.

1.7.2 Straight-Through Estimator

Rather than use the analytical gradient of the loss with respect to the weights, we will calculate an estimate. In the context of backpropagation in BNNs, we will call this a straight through estimator (STE). In STEs, you set the incoming gradients to a threshold function equal to its outgoing gradients, disregarding the gradient of the threshold function itself. The figure below shows one layer of a BNN with Sign(x) as the activation function. Note that the top row is depicting a forward pass where Z are the incoming weights and Y are the outgoing weights (Y = Sign(Z)). The bottom row is depicting a backward pass on weights with respect to a loss function, L. On the backward pass, let $g_Y = \frac{\partial L}{\partial Y}$ be the incoming derivative and let $g_Z = \frac{\partial L}{\partial Z}$ be the analytical derivative (which would be 0 based on the analysis above). The STE computes g_Z to be $g_Y \times \mathbb{1}_{|Z| \leq 1}$ where $\times$ is element-wise multiplication with the indicator function. This is a difficult concept to grasp, but informally, this STE allows the binarization function to act as the identity function in the region around the current point (equivalent to using hard tanh function).



yer
vard
ss
memory
ward
ss
$\frac{dL}{dW}$
late
ghts
Z
$\frac{\partial L}{\partial Z}$
sign(Z)
Y
$\frac{\partial L}{\partial Y}$

Figure 1.8: Zoomed-in look at one layer of BNN during training, showing use of STE during back-propagation.

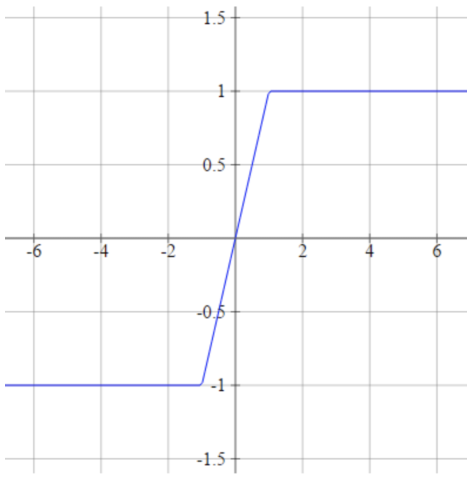


Figure 1.9: STE estimation of Sign function, as hard tanh function.

1.7.3 Backpropagation Pseudo Code

The pseudo code for backward propagation of a BNN is shown below. We have also defined some important terms. In short, for each layer (except the last), we apply straight through estimation. We then perform a backwards pass for BatchNorm and calculate the estimated gradients of the weights with respect to the loss, and apply activation (the transpose step). In the second for loop, the weights are updated via gradient decent, and the gradients are passed left through the network.

- C - loss function
- L - number of layers
- g - gradient (partial derivative)
- s_k - activations before BatchNorm
- a_k - activations after BatchNorm
- W - weight matrix
- superscript 'b' refers to a 'binarized' form of the variable
- θ - BatchNorm parameters
- η - learning rate

```
Compute $g_{a_L} = \frac{\partial C}{\partial a_L}$ knowing $a_L$ and $a^*$
for $k = L$ to $1$ do
    if $k < L$ then
        $g_{a_k} \leftarrow g_{a_k^b} \circ 1_{|a_k| \leq 1}$
    end if
    $(g_{s_k}, g_{\theta_k}) \leftarrow \text{BackBatchNorm}(g_{a_k}, s_k, \theta_k)$
    $g_{a_{k-1}^b} \leftarrow g_{s_k} W_k^b$
    $g_{W_k^b} \leftarrow g_{s_k}^\top a_{k-1}^b$
end for
for $k = 1$ to $L$ do
    $\theta_k^{t+1} \leftarrow \text{Update}(\theta_k, \eta, g_{\theta_k})$
    $W_k^{t+1} \leftarrow \text{Clip}(\text{Update}(W_k, \gamma_k \eta, g_{W_k^b}), -1, 1)$
    $\eta^{t+1} \leftarrow \lambda \eta$
end for
```

Figure 1.10: Pseudo code for BNN backward propagation

1.8 Inference in BNNs

This section serves as a review of how a BNN performs inference. At this stage, the BNN has been trained using multiple iterations of forward and backward propagation. Referencing (the first figure), an input to a BNN is a list (x vector). Let's say our goal is to predict the y. The network, having taken in a vector x, will perform the following operations to predict y (referred to as inference in neural networks). During the forward pass, the input data is processed through the network's layers, using the trained binary weights and activations. In BNNs, the standard dot product in the neurons is replaced by XNOR and popcount operations. These operations are much more efficient than floating-point multiplications and additions. At each layer, after the XNOR and bitcount operations, an activation function is applied (Sign). BNNs often use batch normalization to stabilize and speed up training. During inference, the learned parameters from batch normalization are used to normalize the activations. If the network includes pooling layers, convolutional layers, or other types of layers, these operate in a similar manner to their counterparts in traditional neural networks, but they are adapted to work with binary values. Finally, at the output layer, we receive our value for y (regression) or pass the final vector through typically a soft max (classification).

1.8.1 Restatement of Advantage of BNNs, Limitations

The key advantage of BNNs is that they require significantly less computational power and memory than traditional neural networks, making them well-suited for resource-constrained environments like mobile devices or embedded systems. However, the trade-off is that the binarization of weights and activations can lead to a loss in model accuracy, especially for complex tasks. In simple tasks such as number recognition, the BNN performs well, although requiring more layers than a full-precision NN. For complex tasks such as high-resolution image classificaiton and natural language processing, the lower expressiveness of the BNN may struggle to recognize complex patterns.

Bibliography

[1] Binary neural networks. https://docs.google.com/presentation/d/10lVe51Nh7w_qmhlYheaYP1b8q6i0sutZ/edit#slide=id.p1. Lecture, n.d.

[2] Ee545 (week 10) “binary neural networks” (part i). https://www.youtube.com/watch?v=5K6ko3H_ePg&list=PLC89UNusI0eSBZhwHlGauwNqVQWTquWqp&index=23&t=338s. YouTube video, 2020.

[3] Ee545 (week 10) “binary neural networks” (part ii). https://www.youtube.com/watch?v=LcECmeFqbrI&list=PLC89UNusI0eSBZhwHlGauwNqVQWTquWqp&index=24. YouTube video, 2020.

[4] Ee545 (week 10) “binary neural networks” (part iii). https://www.youtube.com/watch?v=RC_8bPv0uhM&list=PLC89UNusI0eSBZhwHlGauwNqVQWTquWqp&index=25. YouTube video, 2020.

[5] Ee545 (week 10) “binary neural networks” (part iv). https://www.youtube.com/watch?v=P6sD8lI61Uk&list=PLC89UNusI0eSBZhwHlGauwNqVQWTquWqp&index=26. YouTube video, 2020.

[6] Ee545 (week 10) “binary neural networks” (part v). https://www.youtube.com/watch?v=rHa0-mG5SuM&list=PLC89UNusI0eSBZhwHlGauwNqVQWTquWqp&index=27. YouTube video, 2020.

[7] Ee545 (week 10) “binary neural networks” (part vi). https://www.youtube.com/watch?v=z8CclU5wKuY&list=PLC89UNusI0eSBZhwHlGauwNqVQWTquWqp&index=28. YouTube video, 2020.

[8] Neural network. https://docs.google.com/presentation/d/1oC1Z_LzzlGMdDCFpC9rp6U2udwSpCaj-/edit#slide=id.p1. Lecture, n.d.

[9] M. Courbariaux, I. Hubara, D. Soudry, R. El-Yaniv, and Y. Bengio. Binarized neural networks: Training deep neural networks with weights and activations constrained to +1 or -1. arXiv preprint arXiv:1602.02830, 2016.

[10] F. Lin. Xnor neural networks on fpga. http://cs231n.stanford.edu/reports/2017/pdfs/118.pdf. CS231n: Deep Learning for Computer Vision report, n.d.

[11] K. G. A. Ludvigsen. The carbon footprint of gpt-4. https://towardsdatascience.com/the-carbon-footprint-of-gpt-4-d6c676eb21ae. Medium article, July 18, 2023.

[12] Natsu. Paper explanation: Binarized neural networks: Training neural networks with weights and activations constrained to +1 or 1. https://mohitjain.me/2018/07/14/bnn/. Blog post by Mohit Jain, August 16, 2018.

[13] V. K. Ojha. Binary neural networks: A game changer in machine learning. https://medium.com/geekculture/binary-neural-networks-a-game-changer-in-machine-learning-6ae0013d3dcb. Medium blog post, February 19, 2023.

[14] C. Yuan and S. S. Agaian. A comprehensive review of binary neural network. Artificial Intelligence Review, 2023. Pages 1-65.