

# BinaryConnect in PyTorch

Team 4b: The Umbrellass

Original Paper: <https://arxiv.org/abs/1511.00363>

GitHub: <https://github.com/CoolSunflower/BinaryConnect-PyTorch>

# Core Algorithm Pseudocode

---

**Algorithm 1** SGD training with BinaryConnect.  $C$  is the cost function for minibatch and the functions  $\text{binarize}(w)$  and  $\text{clip}(w)$  specify how to binarize and clip weights.  $L$  is the number of layers.

---

**Require:** a minibatch of (inputs, targets), previous parameters  $w_{t-1}$  (weights) and  $b_{t-1}$  (biases), and learning rate  $\eta$ .

**Ensure:** updated parameters  $w_t$  and  $b_t$ .

**1. Forward propagation:**

$w_b \leftarrow \text{binarize}(w_{t-1})$

For  $k = 1$  to  $L$ , compute  $a_k$  knowing  $a_{k-1}$ ,  $w_b$  and  $b_{t-1}$

**2. Backward propagation:**

Initialize output layer's activations gradient  $\frac{\partial C}{\partial a_L}$

For  $k = L$  to 2, compute  $\frac{\partial C}{\partial a_{k-1}}$  knowing  $\frac{\partial C}{\partial a_k}$  and  $w_b$

**3. Parameter update:**

Compute  $\frac{\partial C}{\partial w_b}$  and  $\frac{\partial C}{\partial b_{t-1}}$  knowing  $\frac{\partial C}{\partial a_k}$  and  $a_{k-1}$

$w_t \leftarrow \text{clip}(w_{t-1} - \eta \frac{\partial C}{\partial w_b})$

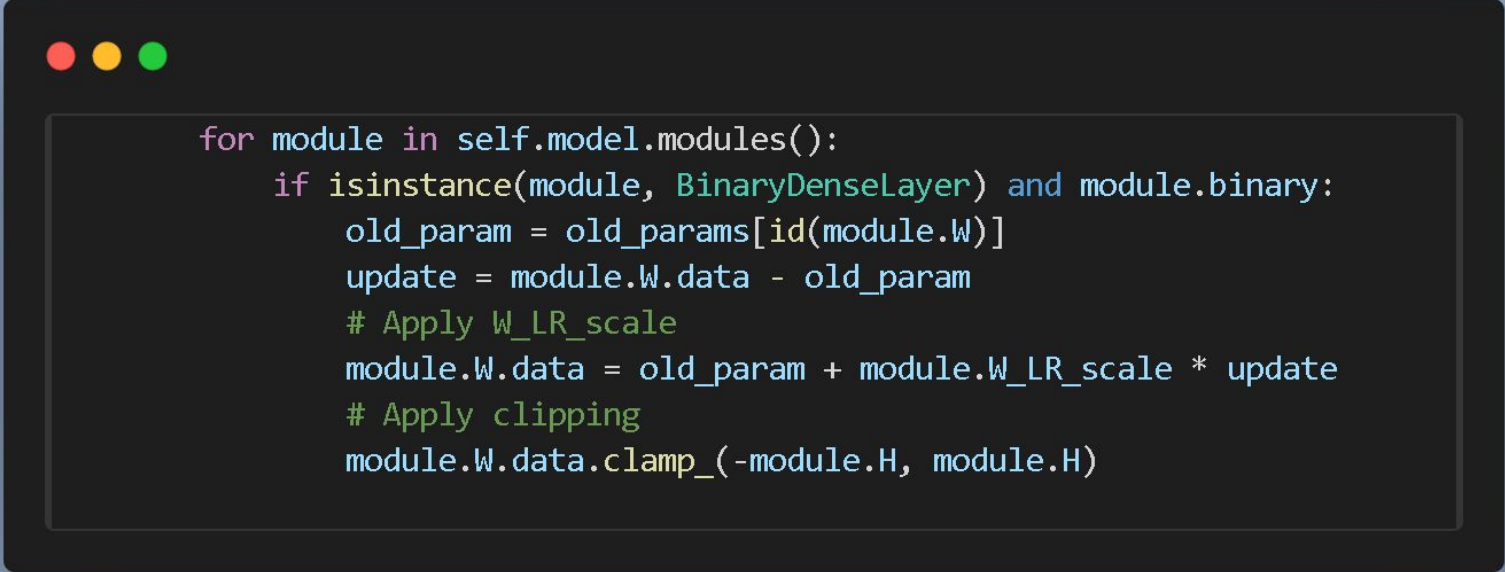
$b_t \leftarrow b_{t-1} - \eta \frac{\partial C}{\partial b_{t-1}}$

---

# Binarization

```
def hard_sigmoid(x):  
    return torch.clamp((x+1.)/2., 0, 1)  
  
def binarization(W, H, binary=True, deterministic=False, stochastic=False, srng=None):  
    if not binary or (deterministic and stochastic):  
        Wb = W  
    else:  
        # [-1,1] -> [0,1]  
        Wb = hard_sigmoid(W/H)  
  
        # Stochastic BinaryConnect  
        if stochastic:  
            Wb = torch.bernoulli(Wb)  
        # Deterministic BinaryConnect (round to nearest)  
        else:  
            Wb = torch.round(Wb)  
  
        # 0 or 1 -> -1 or 1  
        Wb = torch.where(Wb==1, H, -H)  
  
    return Wb
```

# Weight Update



```
for module in self.model.modules():  
    if isinstance(module, BinaryDenseLayer) and module.binary:  
        old_param = old_params[id(module.W)]  
        update = module.W.data - old_param  
        # Apply W_LR_scale  
        module.W.data = old_param + module.W_LR_scale * update  
        # Apply clipping  
        module.W.data.clamp_(-module.H, module.H)
```

# Test Time Inference

Use the real-valued weights  $w$ , i.e., the binarization only helps to achieve faster training but not faster test-time performance.

In the stochastic case, many different networks can be sampled by sampling a  $w_b$  for each weight according to Eq. 2. The ensemble output of these networks can then be obtained by averaging the outputs from individual networks.

$$w_b = \begin{cases} +1 & \text{with probability } p = \sigma(w), \\ -1 & \text{with probability } 1 - p. \end{cases}$$

We employ the first approach following the direction of the original work.

# Results

We trained our model with stochastic BinaryConnect as that's what gave the best results in the original paper.

Validation Error Rate	Ours (PyTorch)	Original (Theano)
CIFAR10	7.42%	-
MNIST	1.10%	-

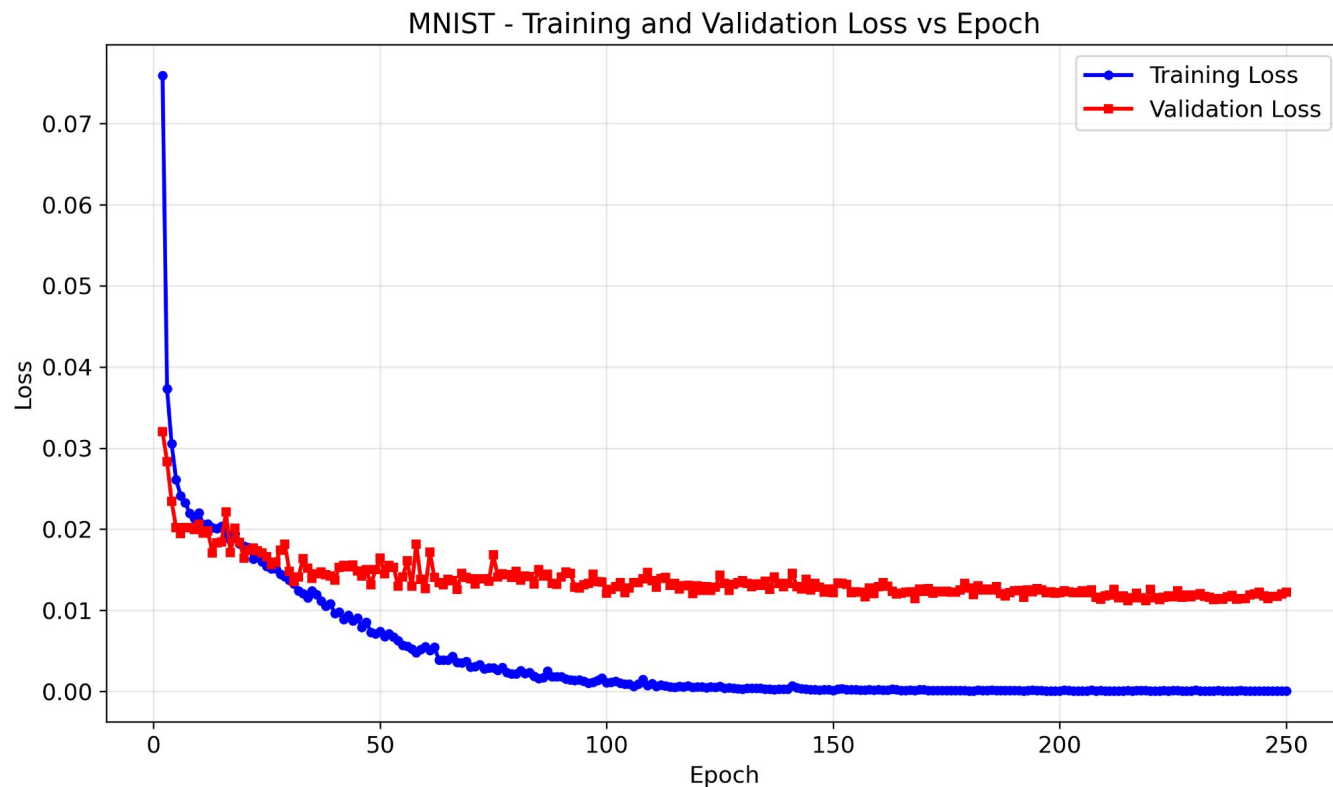
Test Error Rate	Ours (PyTorch)	Original (Theano)
CIFAR10	8.03%	8.27%
MNIST	1.09%	1.18%

# MNIST Experimentation

The MNIST model consists of 3 hidden layers with 1024 ReLU nodes each with BN layers in between two hidden layers. Training-Validation-Testing = 50000-10000-10000. 28x28 Grayscale.

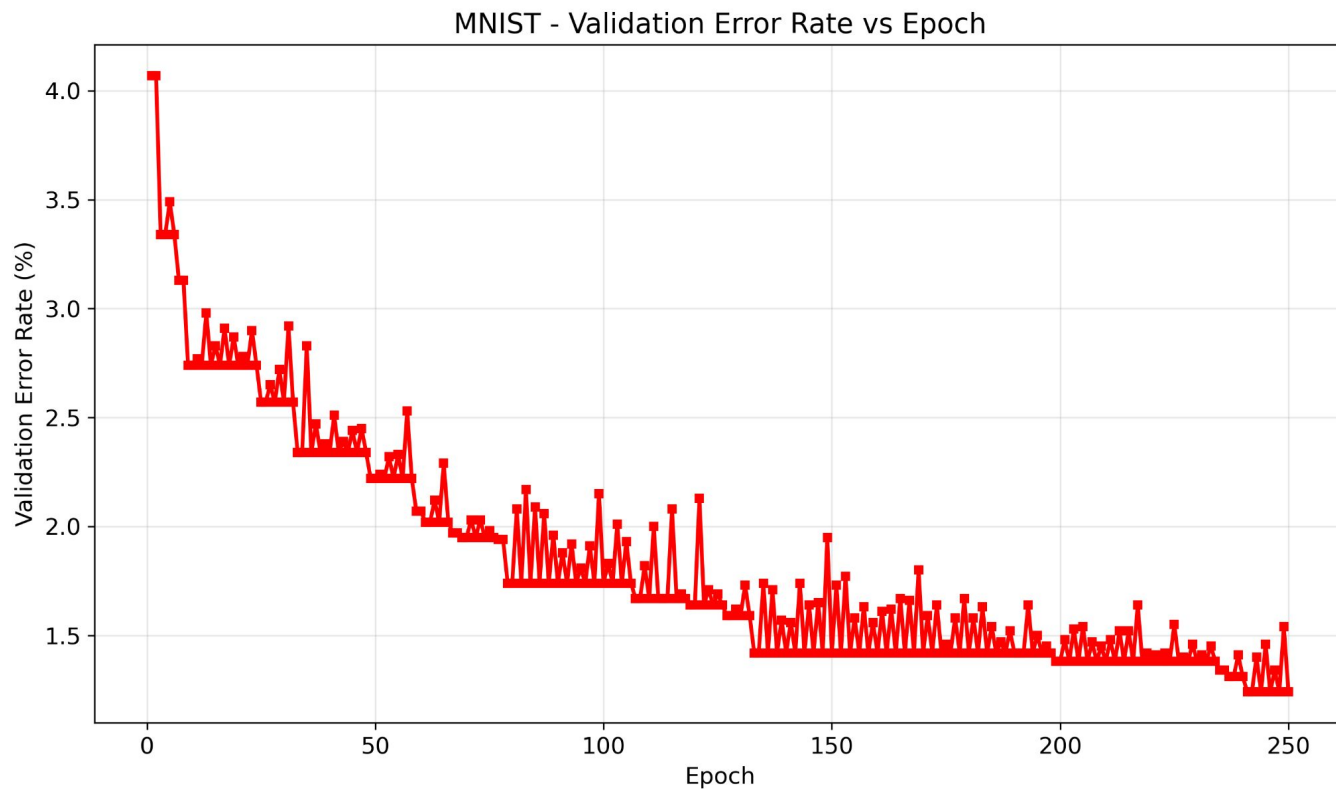
Loss → Squared Hinge Loss (without momentum). Exponentially decaying learning rate ( $0.01 \rightarrow 0.000003$  over 250 epochs).

# MNIST Experimentation





# MNIST Experimentation



# CIFAR10 Experimentation

Model architecture:

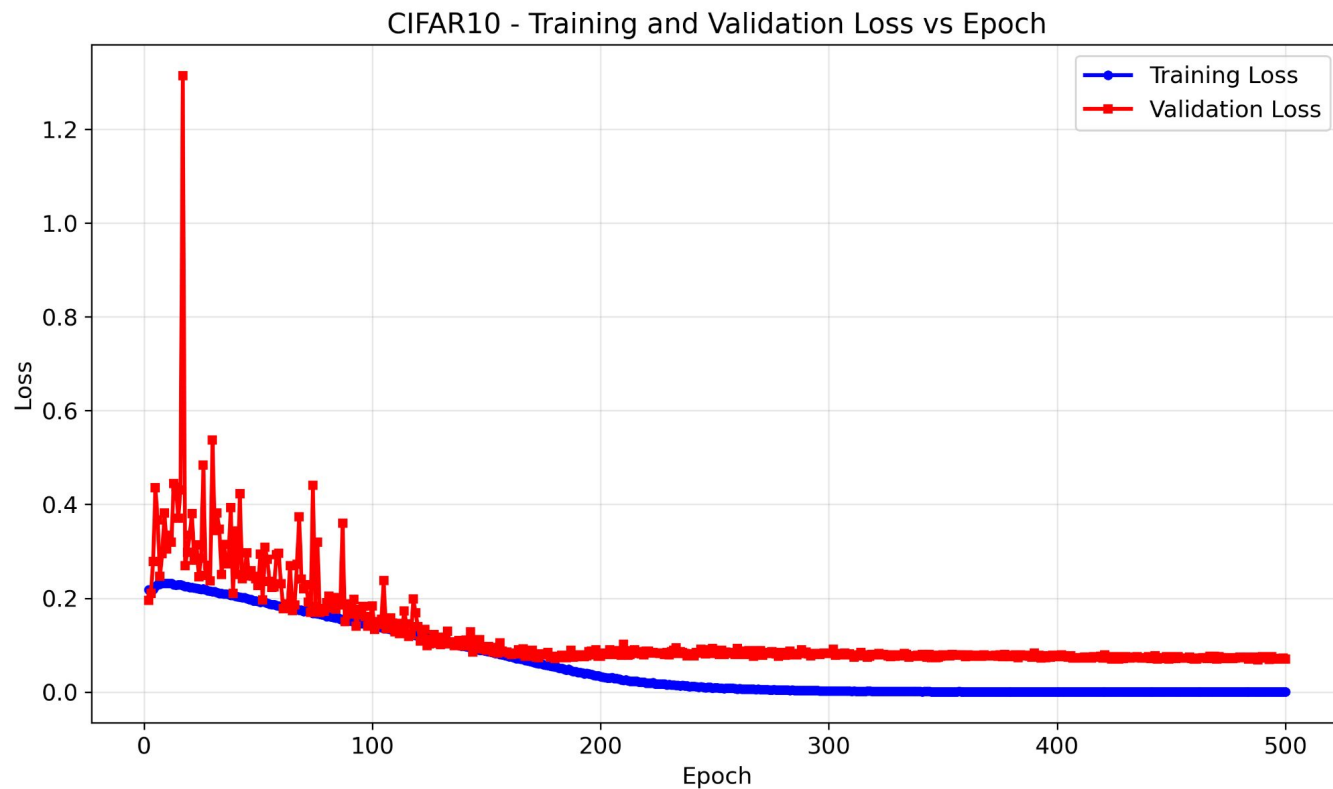
$(2 \times 128C3) - MP2 - (2 \times 256C3) - MP2 - (2 \times 512C3) - MP2 - (2 \times 1024FC) - 10SVM$

Training-Validation-Testing: 45000-5000-10000

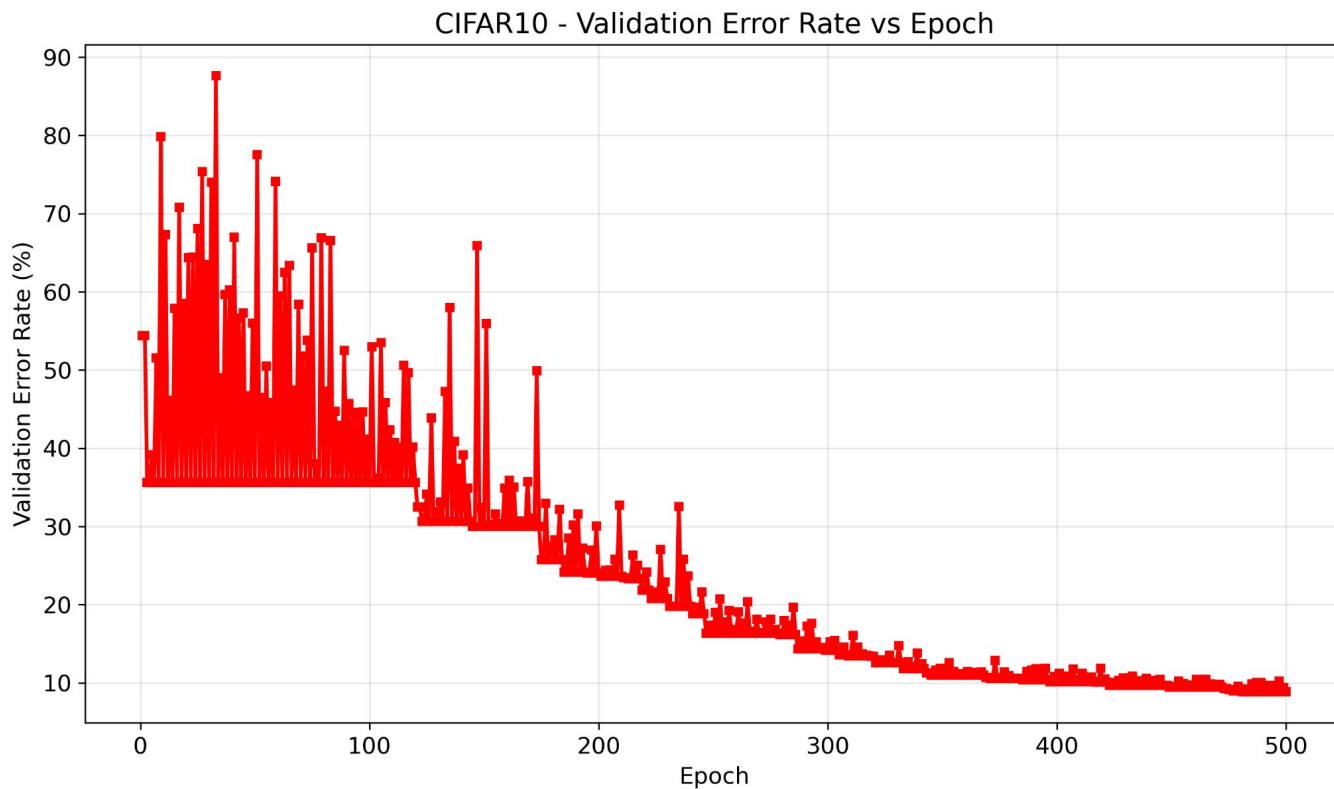
Squared Hinge Loss (with ADAM)

Exponentially decaying LR ( $0.03 \rightarrow 0.000002$  over 500 epochs)

# CIFAR10 Experimentation



# CIFAR10 Experimentation



# Asking the real question

**Does this modification of using binary weights actually make the training faster?**

**TL;DR: Probably Not.**

The main logic behind this model is that using binary weights will reduce the requirements of multiply-accumulate operations in the forward pass to just be accumulate operations. Thus decreasing the forward pass times.

This should help (not a lot, since majority time is anyway taken by backward pass [[Explanation](#)]). But this can only help if you explicitly code your forward pass method yourself as all modern DL frameworks rely on GPUs.

# Asking the real question

**Does this modification of using binary weights actually make the training faster?**

**TL;DR: Probably Not.**

GPUs are great at multiply-accumulate (MAC) operations, but the benefit they offer is in the multiply part parallelism and not the accumulate part.

So unless you can code a faster way to do the accumulate operation, the benefits of this model are more or less rendered moot by the incredible MAC operations speed.

The original Theano codebase actually does not incorporate any new accumulate operation code and instead relies on the linear matrix multiplication provided by Theano. So we do the same in our implementation.

# Asking the real question

To reinforce this point further that the new DL frameworks have many more optimisations already built into them. We also trained a separate DL model (with exact same architecture (except the binarization part), and actually had an incredible decrease in the running time.

MNIST w/ Stoch. BinaryConnect (PyTorch)	MNIST Standard (PyTorch)
9.58s/epoch (Total = 39.9 minutes)	4.93s/epoch (Total = 20.5 minutes)

NOTE: This experimentation regarding time was not part of the original paper, which just states that the model should potentially offer a 3x boost in training time. We are not denying that, just saying, like they did, that it would require some specialized hardware.

Thank you!