Investigation and Analysis on Quantized Neural Networks

Section Number: DSC 180A - A12
Group members: Aileena Wen, Bonnie Zhong,
Hanzhang Zhao, Lukas Williams, Xun Gong
Supervisor: Rayan Saab

March 2022

Contents

Abstract

Our problem of interest for this project is to investigate ways to reduce the size of machine learning models while maintaining the same functionalities and performances as before. Our initial step is to investigate the methods introduced in the paper "Quantized Neural Networks: Training Neural Networks with Low Precision Weights and Activations." We will reproduce the methods and experiments described in the paper with more modernized libraries, Tensorflow and PyTorch, instead of Theano, which the paper initially used. After we have a better understanding of neural networks and the current methods, we will shift our focus on assessing the effect of changing the number of bits has on the performance of our neural network model. Additionally, we will explore if the use of other functions(other than the HardTanh function) during backpropagation affects BNNs' accuracy.

1 Introduction and Motivation

Machine learning models are powerful but large in size. Due to their size, machine learning models are unable to function on smaller devices such as phones and sensors. This dramatically reduces the practical application of such a powerful tool. There exists a gap between the needs of the market and our current technology. With our project, we hope to provide some insights on how to bridge this gap by exploring possible ways to reduce the size of machine learning models(particularly Neural Networks) while minimizing the change in performance. We are interested in this topic because we believe that utilizing NNs on smaller devices will revolutionize the industry, and we would like to be part of this world-changing process while diving deep into studying some of the cutting-edge models and optimization methods.

We based our initial investigation on the paper "Quantized Neural Networks: Training Neural Networks with Low Precision Weights and Activations." We studied the methods introduced in the paper and reproduced the BNN described in the paper using two more modernized libraries, Tensorflow and PyTorch, instead of Theano, which the paper originally used. The accuracy of our model is close to the accuracy of the paper's model.

Furthermore, we explore how using different numbers of bits affects the accuracy of our models, especially for RNN and LSTM, since the paper mentioned how RNN and LSTM did not work well for 2 bits and needed 4 bits instead. Furthermore, we will also investigate how using functions other than the HardTanh function during backpropagation will change the model's accuracy.

2 Theory

2.1 Introduction to Neural Networks

Neural networks are a class of machine learning algorithms modeled loosely after the human brain. It is used to model complex datasets with multiple hidden layers and non-linear activation functions.

2.1.1 General Structure of Neural Networks

An artificial neural network consists of node layers containing an input layer, hidden layer(s), and an output layer. Each node (neuron) has an associated weight and activation function. It takes an input and outputs an activation, which is passed onto the next layer. Neural networks rely on training data to improve their performance over time. There are generally two ways of training a neural network:

- feedforward: flowing in one direction only, from input to output.
- backpropagation: move in the opposite direction, from output to input. Most NNs are feedforward, but we can also train NN through a combination of both feedforward and backpropagation.

2.1.2 Feedforward Neural Networks

A feedforward neural network is an artificial neural network where the edges between nodes do not form a cycle. In this type of NN, information moves only in the forward direction from the input nodes, through any hidden/dense nodes, to the output nodes.

Let us consider an example of a single neuron with 3 inputs, we have:

$$y_1 = \text{Activation}((\theta_1 x_1 + \theta_2 x_2 + \theta_3 x_3) + b) \tag{1}$$

where

- y_1 is a neuron.
- x_1, x_2, x_3 are neuron's inputs.
- $\theta_1, \theta_2, \theta_3$ are weights for each neuron's input (x1, x2, x3 accordingly).
- b is the bias term, which is similar to the role of a constant in a linear function, whereby the line is effectively transposed by the constant value.
- Activation: some activation function (nonlinear), to ensure nonlinearity of the NN.

*Notice that the output of the neuron, i.e. y_1, is the activation function of the weighted sum of the neuron's input (x_1, x_2, x_3).

Now apply the same logic to 2 neurons with 3 inputs, we have:

$$y_1 = \text{Activation}((\theta_{11} x_1 + \theta_{12} x_2 + \theta_{13} x_3) + b) \tag{2}$$

$$y_2 = \text{Activation}((\theta_{21} x_1 + \theta_{22} x_2 + \theta_{23} x_3) + b) \tag{3}$$

In this case, every neuron in the second layer (y_1, y_2) is connected to every neuron (x_1, x_2, x_3) in the first layer. This type of neuron is called a fully connected network. For our convention, the first index of w indicates the output neuron while the second index indicates the input neuron, so if we express θ as a vector, we get:

$$\begin{pmatrix} \theta_{11} & \theta_{12} & \theta_{13} \\ \theta_{21} & \theta_{22} & \theta_{23} \end{pmatrix}$$

Expressing x as a vector $\vec{x}$, we get:

$$\begin{pmatrix} x_1 \\ x_2 \\ x_3 \end{pmatrix}$$

Expressing y as a vector $\vec{y}$, we get:

$$\begin{pmatrix} y_1 \\ y_2 \end{pmatrix}$$

Thus for any n layer of output, we have:

$$y_{n-1} = \text{Activation}(\theta_{n-1}x_{n-1} + b) \tag{4}$$

x_{n-1} is the input neuron from the $(n-1)^{th}$ layer, y_n the output layer for the nth layer. A neural network propagates information through the above equation and the NN can have as many layers as needed. A more general formula for a NN with multiple hidden layers and multiple nodes would be:

$$a_n^L = [\sigma(\sum_m \theta_{nm}^L[\cdots[\sigma(\sum_j \theta_{kj}^2[\sigma(\sum_i \theta_{ji}^1 x_i + b_j^1)] + b_k^2)]\cdots]_m + b_n^L)]_n \tag{5}$$

where we have L layers with n nodes and L-1 layers with m nodes.

If we are not training a NN, then all steps are completed and the final Y neuron should give us the output. However, if we are training a NN, then we need to proceed onto backpropagation to adjust the weight for our NN to be more accurate.

2.1.3 Backpropagation

Backpropagation is an algorithm that updates the weights between each epoch, or training cycle. To know how to update the weights, we first need to define the Cost function, which represents the sum of error (difference between the actual label and predicted y):

$$J(\theta) = \frac{1}{m}\sum_{i=1}^{m} \text{Cost}(h_{\theta(x^{(i)})}, y^{(i)}) \tag{6}$$

$h_\theta(x) = g(\theta^T x)$, where g is some activation function.

Our goal is to optimize the Cost function. This means we need to minimize $J(\theta)$ using the optimal set of weights θ. To do so, we need to use backpropagation to compute the partial derivative of $J(\theta)$ with respect to each θ parameter, which is then used in the Gradient Descent algorithm (more in section 2.3) for calculating the θ that minimizes $J(\theta)$.

It is important to note that because we need to perform gradient descent, popular activation functions such as the Sigmoid function cause trouble during backpropagation. This is because it is non-convex, meaning that it has multiple local minimums, which makes it difficult to find the global minimum. We usually use some other convex function in replacement of the Sigmoid function during backward propagation. The paper uses the HardTanh function to replace the hard sigmoid function. We will be exploring if replacing the HardTanh with something else will make a difference in the performance of BNN or not.

2.2 Binarized Neural Network

This section will explain the binarization methods described in the paper.

2.2.1 Deterministic v.s. Stochastic Binarization

The paper suggests constraining both the weights and activations to either +1 or -1 when training a BNN. The reason is that +1 and -1 are very advantageous from a hardware perspective in which run-time on GPU can be significantly reduced. This is because +1 and -1 can be stored in fewer bits than a typical 32-bit float. Using a technique known as "SWAR," it is possible to group smaller bits into a single 32-bit register on the GPU, reducing the number of clock cycles necessary to perform a computation. The paper utilized two different binarization functions to transform the real-valued variable into binarized weights. The first function is deterministic:

$$x^b = \text{sign}(x) = \begin{cases} +1 & \text{if } x \geq 0 \\ \text{-1} & \text{otherwise} \end{cases} \tag{7}$$

x^b is the outputting binarized variable weights θ or activation g, while the input x is the real-valued variable. We take the sign of the real-valued variable as the new weight/activation. The deterministic function is easy to implement and works well in practice. The other binarization function is stochastic:

$$x^b = \text{sign}(x - z) = \begin{cases} +1 & \text{with probability } p = \sigma(x) \\ \text{-1} & \text{with probability } 1 - p \end{cases} \tag{8}$$

where z is a uniform random variable existing in $[-1, 1]$. $\sigma(x)$ is the "hard sigmoid" function where:

$$\sigma(x) = \text{clip}(\frac{x+1}{2}, 0, 1) = \max(0, \min(1, \frac{x+1}{2})) \tag{9}$$

According to the paper, the stochastic function is more appealing theoretically. However, it requires the hardware to generate random bits when quantizing, which is more difficult to implement than a deterministic function. The paper mostly uses the deterministic function, with the expectation of activations at train time in some of their experiments.

2.2.2 Pseudo-Gradient used in BNN Backpropagation

In a standard neural network, we calculate the gradient of the cost function when we do backpropagation. However, this would be meaningless in binarized neural networks if the derivatives of the binarization function have 0 gradients on almost all points. Hence, we will use pseudo-gradient as a substitute to make backpropagation work.

One approach is to use a straight-through estimator (STE) when calculating derivatives in backpropagation steps. We specify its derivatives as follows:

$$\frac{\partial q(x)}{\partial x} = \begin{cases} 1 & \text{if } |x| \leq \text{clip\_value} \\ \text{-1} & \text{if } |x| > \text{clip\_value} \end{cases} \tag{10}$$

This preserves the value of gradient when x is between $\pm$ clip value and makes it 0 when $|x|$ is too large.

2.2.3 Multi-bit Quantization Function

Below is the function we use to create an evenly spaced quantization alphabet:

$$q(x, a, n) = \frac{\text{round}(\frac{(x+a)(n-1)}{2a})(2a-2)}{n-1} \tag{11}$$

Input for the function:

- x = a tensor of weights
- a = Maximum value
- n = Number of quantization elements
    - For 1 bit: 2 elements $-1, +1$

Output for the function:

- A tensor of quantized weights

2.3 Gradient Descent v.s. Stochastic Gradient Descent

As the algorithm sweeps through the training set, it performs the above update for each training example. Several passes can be made over the training set until the algorithm converges. If this is done, the data can be shuffled for each pass to prevent cycles. Typical implementations may use an adaptive learning rate so that the algorithm converges.

In pseudocode, stochastic gradient descent can be presented as follows:

- Choose an initial vector of parameters w and learning rate η
- Repeat until an approximate minimum is obtained:
    - Randomly shuffle examples in the training set

```
◦ for $i = 1, 2, \dots, n$:
    - $w \coloneqq w - \eta \Delta Q_i(w)$
      where Q is the loss function
```

SGD suffices for us to have the least square problem/method solved or justified, which ultimately serves us as a tool to solve locally weighted regressions, especially for non-smooth function/fundamental domains. Additionally, since it is an iterative method, we can boost this method to arbitrarily higher dimensions when dealing with multiple weights acting on parameters.

2.4 Introduction to Language Model

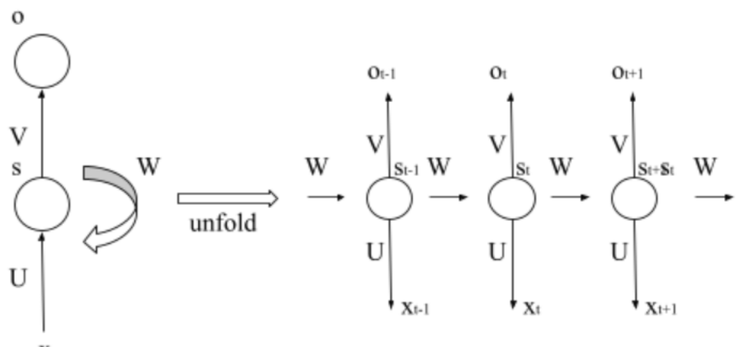
A language model is a probability distribution over a sequence of words that, for a given sequence of length m, can produce a probability $\mathbb{P}(w_1, w_2, \dots, w_m)$ for the entire sequence. In fact, it is to find a way to find a probability distribution, which can represent the probability of occurrence of any sentence or sequence. For example, for a sentence composed of T words in order, $\mathbb{P}(s)$ actually solves the joint probability of strings. Using the Bayesian formula, the Chain Rules are as follows:

$$\begin{aligned}\mathbb{P}(s) &= \mathbb{P}(w_1, w_2, \dots, w_T) \\ &= \mathbb{P}(w_1)\mathbb{P}(w_2|w_1)\mathbb{P}(w_3|w_1w_2)\dots\mathbb{P}(w_T|w_1w_2...w_{T-1})\end{aligned} \tag{12}$$

From the above equation, we can see that a statistical language model can be expressed as finding the conditional probability of the occurrence of the next word given the previous word. When we find $\mathbb{P}(s)$, we have actually established a model, where $\mathbb{P}(*)$ is the parameters of the model. If these parameters have been solved, it is easy to get the probability of the string.

2.5 RNN: Recurrent Neural Networks

The hierarchical structure of RNN mainly consists of the input layer, hidden layer, and output layer. The original neural network can only take a certain current input, and the historical input/output has no effect on the current result. Unlike the original neural network, the RNN will use the feature vector in the middle of the historical input as a hidden state, and feed it into the network together with the input at the current moment. This mimics fitting the probability distribution on the time series feature, which can be directly implemented for a time series model. The disadvantage is that the historical feature does not have any weight, so as the recurrence continues, the historical feature will be closer to the current feature to the domain, which cannot establish a relatively long-term connection. We will not dive deep into RNN but focus on LSTM instead since we chose to focus on LSTM due to time limitations.



Hidden Layer Stretch out view

As shown in the figure above is the hierarchical expansion diagram of Hidden Layer. $t-1$, t, $t+1$ represent the time series. X represents the input sample. S_t represents the memory of the sample at time t,

$$S_t = f(W \cdot S_{t-1} + U \cdot X_t) \tag{13}$$

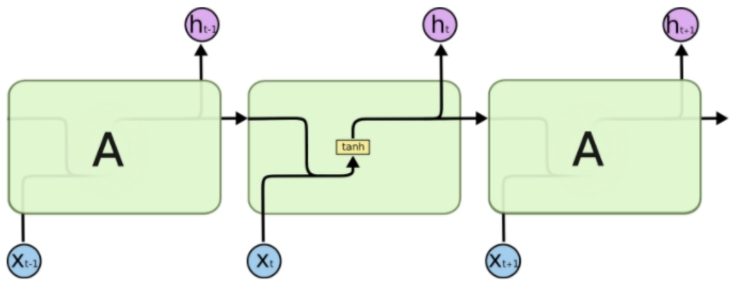
where

- W = weight of the input.
- U = weight of the input sample at this moment.
- V = weight of the output sample.

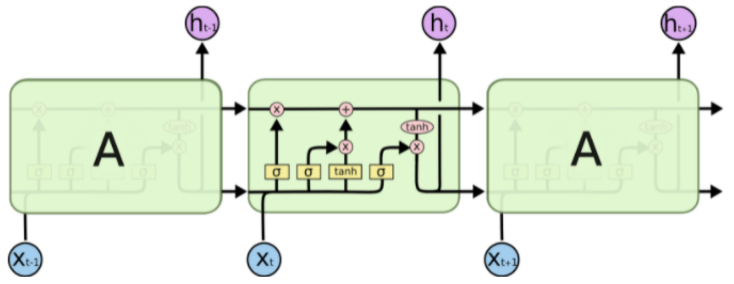
2.6 Introduction to LSTM Network

During backpropagation (feed backpropagation is mentioned in the Q1 paper), RNNs face the problem of vanishing gradients. The vanishing gradient problem happens when the gradient decreases over time as the gradient propagates, and if the gradient value becomes very small, machine learning cannot continue. Therefore, in RNNs, learning stops at layers with more minor gradient updates - these are usually earlier layers. Since these layers do not learn, the RNN can forget what it saw in longer sequences, thus having a short-term memory.

LSTM (Long Short Term Memory Networks) is a special kind of RNN that can deal with the problem of long-term dependencies.



The repeating module in a standard RNN contains a single layer



The repairing module in a LSTM contains 4 interacting layers
(Link [11])

In order to achieve long-term memory, LSTM utilizes three gates to construct the features in time series. Three gates include the input gate, the forgot gate, and the output gate. The input gate layer uses the sigmoid function to determine which feature should be extracted and then uses the tanh function to quantize the weights. The forget gate layer uses a sigmoid to decide what features should be kept or not. After the concatenation of the current input and the output of the previous moment passes through the sigmoid function, the closer it is to 0, the more it is forgotten, and the closer it is to 1, the less it is forgotten. Finally, the model comes to the output layer. The result in the forgotten layer combines with the result of the input gate layer. This combination goes through the tanh function and multiplies with the input that runs after the sigmoid, and then we have the output of the current timestamp.

To quantize the LSTM model, besides quantization of backpropagation, we should also quantize the special activation functions (sigmoid and tanh). We have used the "hard sigmoid" as the activation function for the gate, as mentioned in the paper. The equation is shown as below:

$$\sigma(x) = \mathrm{clip}(\frac{x+1}{2}, 0, 1) = \max(0, \min(1, \frac{x+1}{2})) \tag{14}$$

The hard sigmoid compresses values between 0 and 1. This setting helps to update or forget information because any number multiplied by 0 will get 0, and this part of the information will be eliminated. Likewise, any number multiplied by 1 gets itself, and this part of the information is perfectly preserved. In this way, the network can understand which data needs to be forgotten and which data needs to be saved.

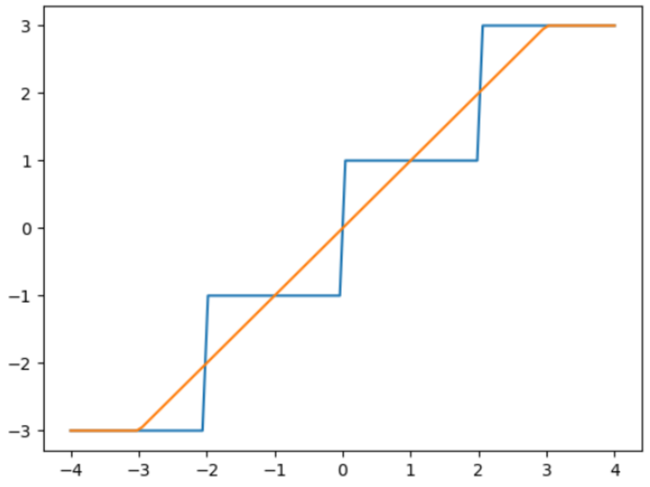
2.7 Activation Function

2.7.1 Hard Tanh

The hard tanh activation function is shown below in equation 15, where a is the hard tanh function and x are the weights.

$$a(x) = \begin{cases} \text{-1} & \text{if } x < -1 \\ \text{x} & \text{if } -1 \leq x \leq 1 \\ 1 & \text{if } x > 1 \end{cases} \tag{15}$$

It is important to note that the choice of 1 is arbitrary and can be changed based on the data being used. This function is useful for gradient descent because the derivative is easy to calculate, being 1 in the range $[-1, 1]$ and 0 elsewhere. To help illustrate, a graph with this function overlaid on a graph of quantized weights can be seen below:



2.7.2 Modified Tanh (Theory inspired by [15])

In addition to the hard tanh, we also investigate the effects of using a modified tanh activation function. The tanh function is shown below in equation (16), where f_1 is the tanh function and $x$are the weights.

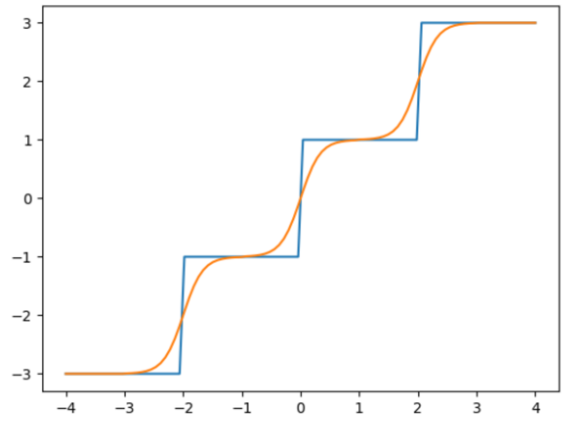
$$f_1(x) = \frac{e^x - e^{-x}}{e^x + e^{-x}} \tag{16}$$

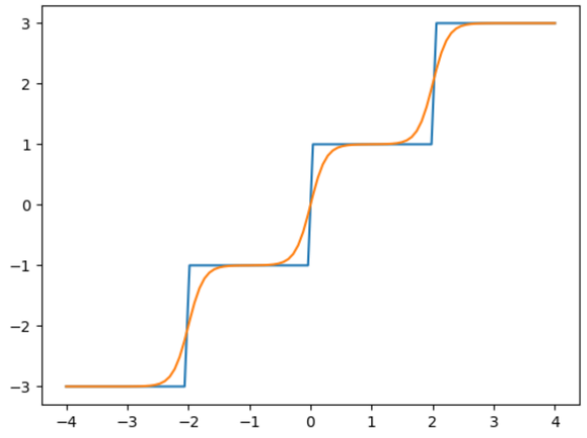
To apply this function to multi-bit models, this function must be modified to match the step-wise nature of quantized weights. To achieve this, we define f_2 and f_3 below in equations (17) and (18) respectively.

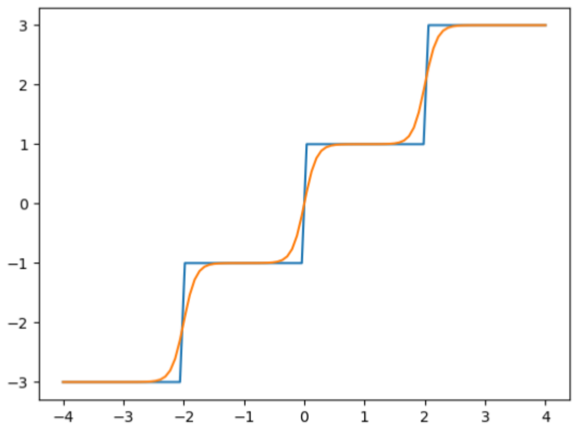
$$f_2(x) = f_1(4x) \tag{17}$$

$$f_3(x) = f_2(x - 2 \cdot \text{round}(\frac{x}{2})) + 2 \cdot \text{round}(\frac{x}{2}) \tag{18}$$

This function is useful as it more closely approximates the quantized weights; however, it is more computationally expensive to calculate the derivative. The choice of 4 in equation (17) affects the curve of the function, and can be changed. We investigate the effects of using 3, 4, and 5 for this value. To help illustrate, these functions are graphed below, overlaid on a graph of quantized weights in figures below respectively







3 Datasets

In order to implement these methods, we use four datasets for our analysis: MNIST, SVHN, CIFAR-10, IMDB_reviews. These datasets are commonly used for machine learning and are publicly available.

3.1 MNIST

The MNIST dataset contains grayscale images of handwritten numbers, ranging from 0 to 9. They are represented as $28\times$ 28 matrices where each value in the matrix is a pixel. Since the images are grayscale, the values in the matrix are 1 if they are black (i.e. there is writing on that pixel) and 0 if they are white (i.e. there is no writing on that pixel). There are 60,000 training images and 10,000 test images available for use.

Below is an example of what the images look like:

3.2 SVHN

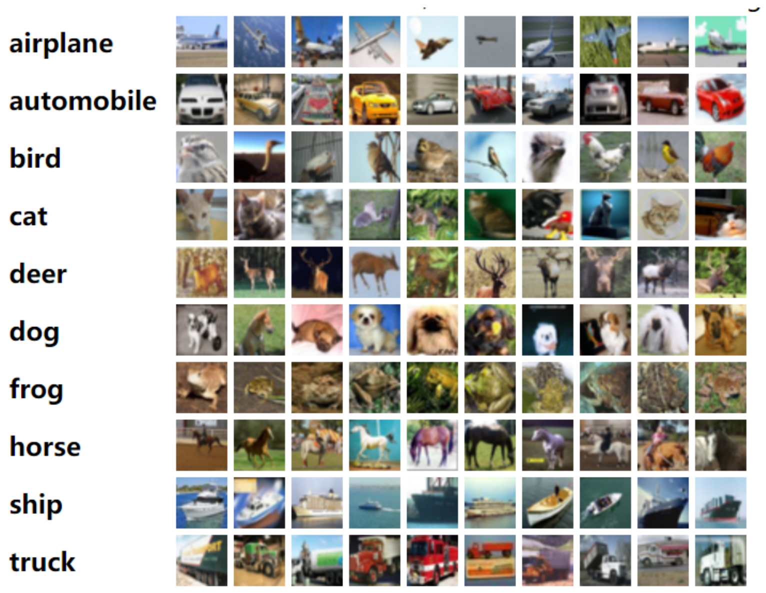
The SVHN dataset contains images of Street View House Numbers, represented as 32×32 matrices of pixel values. Unlike MNIST, these images are in color, and there are elements in the images that are not purely numbers as they are real images of house numbers. The house numbers contain digits ranging from 0 to 9, but since house numbers are multidigit, there are multiple classes to predict. This makes it a more difficult dataset to train on. There are 630,420 images available for use; however, approximately 80% of those are considered to be purely training data due to their less complex nature.

Below is an example of what the images look like:

3.3 CIFAR-10

The CIFAR-10 dataset contains images of various modes of transportation (cars, ships, and trucks) as well as various types of animals (birds, cats, deer, dogs, frogs, and horses). Similar to the SVHN dataset, these images are in color and are represented as 32×32 matrices of pixel values. This is the most complex dataset of the three, and as such is the most difficult to train on. There are 60,000 images available for use.

Below is an example of what the images look like:



3.4 IMDB (imdb_reviews)

This dataset contains a set of 50,000 highly polar movie reviews with additional unlabeled data for use as well. It is used for binary sentiment classification. Our LSTM model is trained on this dataset with the goal of analyzing positive and negative movie reviews.

4 Investigation and Analysis

Initially, we built an MLP in Tensorflow and an MLP and CNN in PyTorch using standard methods to use as baseline models in which to compare our results. The MLPs were trained on the MNIST dataset, and the CNN was trained on the SVHN and CIFAR-10 datasets. The code for our models is on the GitHub repository linked in [1]

4.1 Tensorflow Implementation

4.1.1 Baseline MLP

Our baseline MLP uses dropout and batch normalization layers, along with three hidden dense layers, where each dense layer represents perceptron nodes. The parameters used in the model are the same as the paper we followed, with a couple of exceptions. The paper uses a decaying learning rate, starting at 0.003 and ending at 0.0000003. Our model uses a static learning rate of 0.001 because this achieved the best results. The paper also uses 1000 epochs to train;

however, our model only uses 50. This is due to our model overfitting with more epochs.

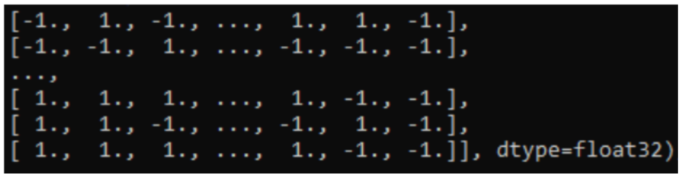
The inputs for our model have 784 features as the MNIST data are images of size 28x28 pixels. Each feature represents the value of a pixel. The labels are one-hot encoded, with each new column representing what number is displayed in the image.

The code for this model is in the file `tensorflow_mnist_fp.py` in [1].

4.1.2 Binzarized MLP

Our binarized MLP uses the same layer setup, parameters, and inputs as the baseline model. However, we also implemented an early stopper with a patience level of 1 (i.e. the model stops training after it fails to reduce the validation loss) to ensure our model does not overfit. The dense layers are also changed so that they use the binarization methods described in section 2.2. In order to do so, we used the Larq library for python, which modifies Tensorflow's classes to binarize the weights with a sign function and compute the gradient with a straight-through estimator.

Below is an array containing the binarized weights, showing that our model operates as expected:



```
[-1.,  1., -1., ...,  1.,  1., -1.],
[-1., -1.,  1., ..., -1., -1., -1.],
...,
[ 1.,  1.,  1., ...,  1., -1., -1.],
[ 1.,  1., -1., ..., -1.,  1., -1.],
[ 1.,  1.,  1., ...,  1., -1., -1.]], dtype=float32)
```

The code for this model is in the file `tensorflow_mnist_binary.py` in [1].

4.2 PyTorch Implementation

4.2.1 Model Structure Complexity Description

- BinLinear(x, y) means that the input dimension of this binary fully connected layer is x, and the output dimension is y.
- ShiftBatchNorm(y) is used to normalize the activation of channel number y. Each block is composed of a fully connected (FC) linear layer-ShiftBatchNorm layer-BinaryActivation layer (activation is reflected in the forward function).
- In TinyModel: define network structures if binary is set to True, we use a binary linear layer and shift batch norm layer. There are two blocks, followed by a DropOut, a final fully connected layer output category.

4.2.2 Function Output Description

- binary_quantization(x):
  ```
  tensor([1., -1., 1.], grad_fn=<AddBackward0>)
  ```
  ```
  tensor([1., 1., 0.])
  ```
  x_back is the actual tensor for gradient computation while $sign(x)$ is the forwarded tensor. And detach() can eliminate the gradient. The output shown is the binarized value and gradient of x.
  Feedforward: quantize all positive numbers in tensor to 1, and negative numbers to -1.
  Backpropagation: results as the gradient of the tensor are 1 in the range of -1 to 1, and the rest is 0.

- BinConv2d(nn.Conv2d):
  Define a binary convolution in input

  - channel = 5
  - output channel = 10
  - kernel size = 3

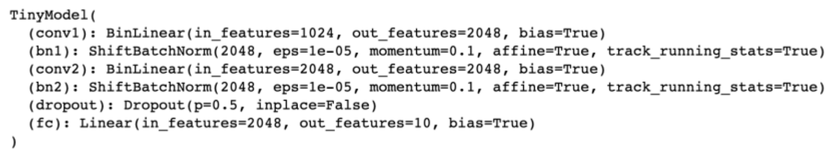
  output printed are the value of convolution and the binarized convolution, in weight value.

- BinLinear(nn.Linear):
  Define a binary fully connected layer with an input channel of 5 and an output channel of 10. Print the value of this fully connected weight and the binarized weight.

- LogAP2(x, eps=0.0):
  The purpose of this function is to convert any floating-point number x to its nearest second power 2^n. That is, for any x, we find n such that the difference between the n^{th} power of 2 and x is the smallest. When x is abnormally small, this function will overflow due to Log, so we add eps.

- ShiftBatchNorm(nn.BatchNorm1d):
  This is to test the performance of Shifted Batch Normalization. The function in code separately prints the results of average and var. The reason why the variance of this is not 1 is that it was originally meant to be divided by 0.333, but the LogAP2 function changed the number to be divided into 0.25. We set a parameter round_var, when it is set to False, the variance can be close to 1.

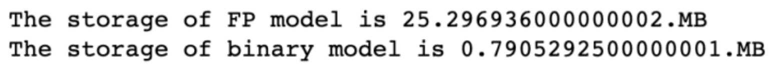
- TinyModel():
  Output print the structure of networks. When binarization is set to True, then we use a binary linear layer and shift the batch norm layer. Inside the function of forward in this part, there is a standard Conv-BN-Act block, note that the activation is not binarized for the first layer.



```
TinyModel(
  (conv1): BinLinear(in_features=1024, out_features=2048, bias=True)
  (bn1): ShiftBatchNorm(2048, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
  (conv2): BinLinear(in_features=2048, out_features=2048, bias=True)
  (bn2): ShiftBatchNorm(2048, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
  (dropout): Dropout(p=0.5, inplace=False)
  (fc): Linear(in_features=2048, out_features=10, bias=True)
)
```

- train() and test():
  The training function and testing function of Binary Models. We used the CUDA GPU as a parameter device to train and test. The parameter optimizer for train() is the Adam optimizer. Param epoch is the current training epoch. These two functions will return the trained model and tested model within binary methods.

- Full Precision Model:
  The first step of FP is to count the number of parameters. Then turn the number of parameters into megabytes: Floating point number 32 bits / 8 bits per byte / 1024 bytes to kB / 1024 bytes to MB. The parameter of the binary network is 1/32 of the original network because it only takes up one bit. The output:



```
The storage of FP model is 25.296936000000002.MB
The storage of binary model is 0.7905292500000001.MB
```

4.3 Binarized LSTM

For our LSTM, we used a dataset composed of IMDB reviews and their associated sentiment (positive or negative). These reviews were then padded to have a maximum sequence length of 300. We then embedded these reviews using 5000 unique works and an output dimension of 15. Next, we passed this matrix into an LSTM layer and finally a Dense output layer. Furthermore, we binarized the weights and activations of this model using the sign quantization method.

4.4 Quantized MLP

To further expand upon our work from the fall quarter, we re-ran our binarized models with more bits. For example, instead of only using ± 1 for the weights, we would use $\pm\frac{1}{3}$ and ± 1. We choose these values by using the equation found in section 2.2.3 equation (11).

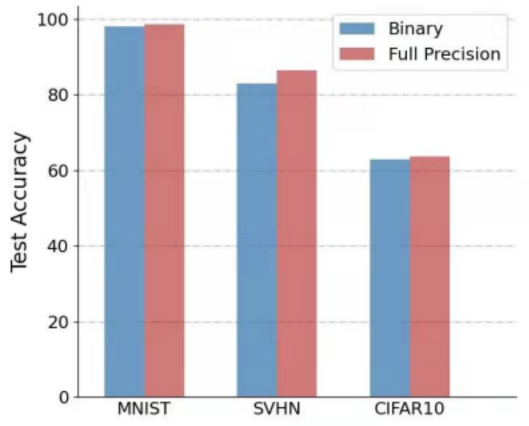
5 Results and Comparison

5.1 PyTorch Implementation

The PyTorch implementation of our models followed the functions and theory in the paper: "Quantized Neural Networks: Training Neural Networks with Low

Precision Weights and Activations". The code for these models is in the file DSC180A-A12 `Pytorch.ipynb`, which reproduces the binarization and batch normalization functions described in section 2.2 to create both full precision and binary models on the MNIST, SVHN, and CIFAR-10 datasets. Below are the results from these models.

Model	MNIST Accuracy	SVHN Accuracy	CIFAR10 Accuracy
Binary	98.04%	83.02%	62.94%
Full Precision	98.52%	86.42%	63.62%



Here we get the results showing that, in general, the MNIST dataset has the highest accuracy among the three, SVHN's accuracy is in the middle level, and CIFAR10 is in the lowest level. These outputs can be interpreted as representing the training difficulty of each dataset. Details on each dataset are included in section 3.

Specifically compared, The full precision model shows higher accuracy among all three datasets. MNIST FP model has a 0.48% accuracy higher than the MNIST Binary model. SVHN FP model has a 3.4% higher accuracy than its Binary model. And the CIFAR10 FP model has 0.68% higher accuracy than its Binary model. The full precision model of MNIST performs similarly to the Theano implementation in the paper we followed, 98.52% vs 98.7% while the Binary model shows only a 1% difference from the Theano implementation in the paper (98.04% vs. 99.04%). But for SVHN and CIFAR10, we still need more development to match some achievements in the QNN paper.

5.2 Quantized MLP

Below are the results for our quantized MLP on the MNIST dataset. The full precision accuracy is 98.54%.

Accuracy / # of Bits	Hard Tanh	Modified Tanh $f_2 = f_1(3x)$	Modified Tanh $f_2 = f_1(4x)$	Modified Tanh $f_2 = f_1(5x)$
1 Bit	97.70%	97.95%	98.27%	98.03%
2 Bit	97.64%	98.10%	98.17%	97.25%
3 Bit	98.33%	97.33%	98.08%	97.03%
4 Bit	97.78%	98.08%	98.06%	97.99%

The hard tanh activation function has the highest accuracy at 3 bits, with only 0.21% difference in accuracy compared to the full precision model. Our custom activation function performs best at 1 bit, with only 0.27% difference in accuracy with the full precision model.

What is interesting is that, while our full precision model performs similarly to the Theano implementation in the paper we followed (98.54% vs 98.7%), the binary version does not (97.70% vs 99.04%). One possible explanation for this is that the random number generation used is different and has a noticeable impact on model accuracy. Our implementation randomly selects $\frac{1}{6}$ of the training data to be used as a validation set. It is possible that the selected random seed results in a poor split in the data and causes our model to stop training too early.

5.3 Quantized CNN

5.3.1 CIFAR-10

Below are the results for our quantized CNN on the CIFAR-10 dataset. The full precision accuracy is 81.54%.

Accuracy / # of bits	Hard tanh	Modified tanh $f_2 = f_1(4x)$
1 Bit	78.86%	76.80%
2 Bit	80.01%	78.38%
3 Bit	79.64%	75.16%
4 Bit	78.84%	75.04%

For both activation functions, the accuracy is closest to the full precision accuracy at 2 bits. The hard tanh activation function performs better than our custom activation function with this model. Using 2 bits, there is a 1.53% loss in accuracy compared to the full precision model.

5.3.2 SVHN

Below are the results for our quantized CNN on the SVHN dataset. The full precision accuracy is 94.02%

Accuracy / # of bits	Hard tanh	Modified tanh $f_2 = f_1(4x)$
1 Bit	93.64%	92.35%
2 Bit	93.60%	92.18%
3 Bit	93.37%	91.14%
4 Bit	93.76%	91.80%

The hard tanh activation function performs best with 4 bits, whereas our custom activation function performs best with 1 bit. Similar to the previous model, the hard tanh activation function performs better overall. Using 4 bits, there is only a 0.26% accuracy loss compared to the full precision model.

5.4 Quantized LSTM

Below are the results for our quantized LSTM. The full precision accuracy is 87.66%.

Accuracy / # of bits	Hard tanh	Modified tanh $f_2 = f_1(4x)$
1 Bit	82.22%	83.00%
2 Bit	83.45%	80.72%
3 Bit	84.84%	77.91%
4 Bit	85.44%	83.51%

The hard tanh activation function performs best with 4 bits, having an accuracy 2.22% lower than the full precision model. Our custom activation function also performs best with 4 bits; however, it performs better than the hard tanh with only 1 bit.

6 Conclusion

The full precision and binarized models we created worked well on the MNIST data, obtaining accuracy in the high 90s for both the Tensorflow and PyTorch implementations. While the SVHN and CIFAR-10 models present lower accuracies relative to the MNIST models (83% $\sim$ 86% and 62% $\sim$ 64%, respectively), The full precision and binarized models have similar performance. On average, the accuracy of the binarized versions of our models is only 2% $\sim$ 4% lower than our full precision models.

In addition, as we increased the number of bits used for MLP, the accuracy decreased and then increased. For CNN, the trend of the change in accuracy is inconsistent across datasets in which it decreased for CIFAR-10 but decreased then increased for SVHN. However, the increase in bits clearly increased the accuracy for LSTM. To conclude, the effect of changing the number of bits is unclear for simpler models such as MLP and CNN but proves to be advantageous for more complex models such as LSTM. Nevertheless, binarizing LSTMs is possible, but more bits are necessary to achieve high accuracy for this model.

Moreover, it is interesting to see that the modified Tanh function, which we thought should perform better than hard Tanh, performs worse than hard Tanh for CNN and LSTM while having an unclear effect on MLP.

We would like to further investigate three ideas in the future. First, we would like to explore if different network sizes and complexities (by using different neuron sizes and a different number of layers) will significantly impact the performance of the NN. Second, we would like to compare the performance of different substitute activation functions for backpropagation and investigate the reason behind why the modified Tanh did not perform better than the hard Tanh function, when it is a better approximation of the quantization function, through testing different optimization strategies and different hyper-parameters. Finally, since we only used up to 4 bits for our experiments, we would like to test multi-bit quantization up to 32 bits in the future while keeping a record of the power consumption and time complexity for both the training and application phase of the model to produce not only a clearer picture of the effect of number bits being used in the model but also to have a more comprehensive metric to evaluate the efficiency of the model.

References

[1] https://github.com/lwilliams620/dsc180a-section DSC180AB Section A12 Repository.

[2] Mishra, Deepak. "Stochastic Gradient Descent Algorithm Tutorial". Metamug. Jul 09, 2018. https://metamug.com/article/groovy/stochastic-gradient-descent-tutorial-code-by-andrew-ng.html.

[3] Alake, Richmond. "Batch Normalization In Neural Networks Explained (Algorithm Breakdown)". Towards Data Science. Apr 21, 2020. https://towardsdatascience.com/batch-normalization-explained-algorithm-breakdown-23d2794511c.

[4] "MNIST in CSV". Pjreddie. https://pjreddie.com/projects/mnist-in-csv/.

[5] LeCun, Yann, et al. "THE MNIST DATABASE of handwritten digits". http://yann.lecun.com/exdb/mnist/.

[6] Loffe, Sergey, and Christian Szegedy. "Batch Normalization: Accelerating Deep Network Training by Reducing Internal Covariate Shift". Arxiv. Mar 02, 2015. https://arxiv.org/abs/1502.03167.

[7] Bernacki, Mariusz. "Principles of training multi-layer neural network using backpropagation". Katedra Elektroniki AGH. Jun 09, 2004. http://galaxy.agh.edu.pl/ vlsi/AI/backp$_{te}n/backprop.html$.

[8] Ognjanovski, Gavril. "Everything you need to know about Neural Networks and Backpropagation — Machine Learning Easy and Fun". Towards Data Science. Jan 14, 2019. https://towardsdatascience.com/everything-you-need-to-know-about-neural-networks-and-backpropagation-machine-learning-made-easy-e5285bc2be3a.

[9] Kanakiya, Bhargav. "Can LSTM Be Fully Quantized For Inference? · Issue #25563 · Tensorflow/Tensorflow". Github, 2022, https://github.com/tensorflow/tensorflow/issues/25563.

[10] Morais, Reuben. "Can't Do 'Weights' Quantization On An LSTM RNN · Issue #7949 · Tensorflow/Tensorflow". Github, 2022, https://github.com/tensorflow/tensorflow/issues/7949.

[11] Olah, Christopher. "Understanding LSTM Networks - Colah's Blog". Colah.Github.Io, 2022, https://colah.github.io/posts/2015-08-Understanding-LSTMs/.

[12] Z. Zhang, G. Ni and Y. Xu, "Ship Trajectory Prediction based on LSTM Neural Network," 2020 IEEE 5th Information Technology and Mechatronics Engineering Conference (ITOEC), 2020, pp. 1356-1364, doi: 10.1109/ITOEC49072.2020.9141702.

[13] S. Cui, C. Li, Z. Chen, J. Wang and J. Yuan, “Research on Risk Prediction of Dyslipidemia in Steel Workers Based on Recurrent Neural Network and LSTM Neural Network,” in IEEE Access, vol. 8, pp. 34153-34161, 2020, doi: 10.1109/ACCESS.2020.2974887.

[14] X. Wang, T. Peng, P. Zuo and X. Wang, “Spectrum Prediction Method for ISM Bands Based on LSTM,” 2020 5th International Conference on Computer and Communication Systems (ICCCS), 2020, pp. 580-584, doi: 10.1109/ICCCS49078.2020.9118535.

[15] Professor Saab Rayan’s in-section description of solutions.

[16] Hubara, Itay. “Quantized Neural Networks: Training Neural Networks with Low Precision Weights and Activations”, edited by Nando de Freitas. Journal of Machine Learning Research 18 (2018) 1-30.