Towards Accurate Binary Convolutional Neural Network

E4040.2021Fall.KFW.report
Heng Kan hk3127, Huaqing Fang hf2431, Jingyuan Wang jw4000
Columbia University

Abstract

This project tried to examine if using a combination of binary convolutional layers to approximate normal convolutional layers can obtain similar accuracies as full precision accuracies in LeNet, AlexNet and VGG16. In our experiments, the accuracies of LeNet and VGG16 get close to the full precision accuracies as N, the number of ApproxConv and M, the number of BinConv increase, which are consistent with the original paper. However, the accuracy of AlexNet stucks at 5% no matter how we changed N and M. The strategy of approximating convolutional layers using a linear combination of binary layers seems replicable in many ANN architectures.

1. Introduction

So far, data scientists have developed plenty of artificial neural network (ANN) architectures which could perform equally well as humans in image classification and speech recognition. However, such architectures are generally huge in size, resulting in exhaustive resource requirements for both memory and computational power. This makes the architectures hard to deploy and few people could really enjoy the advancement in deep learning. Considering that people recently spend most of their time with mobile terminals, such as mobile phones, ipads and laptops, an ideal situation would be that ANNs are compatible with these mobile devices. Therefore, people may want to store pre-trained low-memory architectures and develop some more efficient prediction algorithms. In other words, people could trade training time for prediction time and space. Previous work of using binary convolutional layers has shown that such improvements in memory and speed are attainable but the accuracy has a significant drop. Therefore, it is not eventually put into business use. "Towards Accurate Binary Convolution Neural Network" proposed using a linear combination of binary filters to approximate a real-valued convolutional filter. By using binary bits, one and zero, the architecture not only gets a significant reduction in memory size, but also benefits from bitwise computation, which in term translates to much more efficient prediction. The paper above achieves low memory and high efficiency while maintaining high accuracies with Resnet. We would like to reproduce the results and check if the same strategies can be replicated in different ANN architectures while reproducing similar results. If that is the case, we believe that there would be huge business potential and more advanced ANN architectures would really come to our lives. Therefore, we examine the effectiveness of the paper's strategy in ANNs of various sizes (ANNs with a small number of convolutional layers and ANNs with more convolutional layers) using LeNet, AlexNet and VGG16.

2. Summary of the Original Paper

2.1 Methodology of the Original Paper

The original paper introduces a novel method to train convolutional neural networks(CNNs) with filter weights and activations constrained to binary {-1,+1}. Using binary weights and activations will reduce memory size and have lower power consumption and faster test-time inference. However, accuracy will decrease for previous binarized CNNs. To address this problem, the original paper introduced two steps of training: (1) approximating weights with the linear combination of multiple binary weight bases; (2) approximating activations(inputs of convolution) with the linear combination of binary activations.

The authors put forward the following binarization methods.

1. Weight approximation:

They estimate the weight filter $W \in R^{w*h*c_{in}*c_{out}}$ using the linear combination of M binary filters $B_1, B_2, B_3 \ldots B_M \in \{-1, +1\}^{w*h*c_{in}*c_{out}}$ such that they approximate W by $\alpha_1 B_1 + \alpha_2 B_2 + \ldots + \alpha_M B_M$.

They fix B_i's as follows:

$$B_i = F_{u_i}(W) = sign(W - mean(W) + u_i std(W))$$

$$u_i = -1 + (i-1) * \frac{2}{M-1}, \text{ i =1,2,3…,M.} \quad \text{(a)}$$

This is based on the fact that the full-precision weights tend to have a symmetry, non-sparse distribution, which is close to the Gaussian distribution.

With B_i's fixed, they obtain α by solving the linear regression problem for a given w:

$$min_{\alpha} J(\alpha) = ||w - B\alpha||^2 \quad \text{(b)}$$

They then back-propagate through B_i's using the "straight-through estimator"(STE). Assume c is the cost

function, A and O are input and output respectively, they compute the forward and backward as follows:

Forward:

(1) $B_1, B_2, \ldots, B_M = F_{u_1}(W), F_{u_2}(W), \ldots, F_{u_M}(W)$

(2) Solve α by $min_{\alpha} J(\alpha) = ||w - B\alpha||^2$

(3) $O = \sum_{m=1}^{M} \alpha_m Conv(B_m, A)$

Backward:

$$\frac{dc}{dW} = \frac{dC}{dO}\left(\sum_{m=1}^{M} \alpha_m * \frac{dO}{dB_m} * \frac{dB_m}{dW}\right) = (by\ STE)$$

$$\frac{dC}{dO}\left(\sum_{m=1}^{M} \alpha_m * \frac{dO}{dB_m}\right) = \sum_{m=1}^{M} \alpha_m * \frac{dC}{dB_m} \qquad (e)$$

2. Multiple binary activations and bitwise convolution:

To utilize the bitwise operation, the authors binarize their activations (inputs of convolution) as well.

They transform the real-valued activation R into binary activations by using the following binarization function.

$$H_v(R) = 2I_{h_v(R)\geq 0.5} - 1$$

where h is a bounded activation function

$h_v(x) = clip(x + v, 0, 1)$ to ensure h(v) ∈ [0,1] and v is a shift parameter.

The forward and backward approach of the activation are given as follows:

Forward:

$$A = H_v(R)$$

Backward:

$$\frac{dC}{dR} = \left(\frac{dC}{dA}\right) I_{0<=R-v<=1}\ using\ STE$$

Firstly, they use batch normalization before activation to keep the distribution of activations relatively stable.

Secondly, they estimate the R by linear combination of N binary activations:

$$R \approx \beta_1 A_1 + \beta_2 A_2 + \ldots + \beta_N A_N$$

where $A_1, A_2, \ldots, A_N = H_{v_1}(R), H_{v_2}(R), \ldots, H_{v_N}(R)$

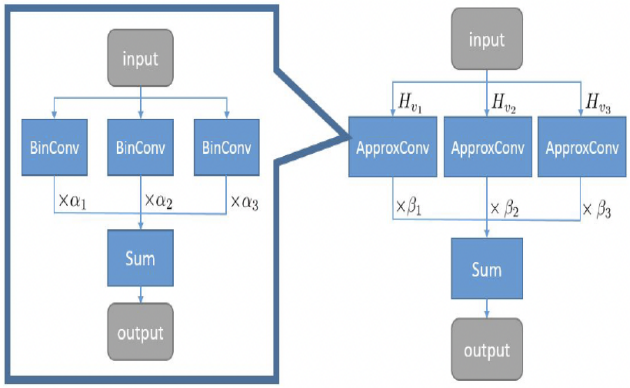
They notice that the parameters β_n's and v_n's are both trainable. Without an explicit linear regression approach, they are tuned by the neural network itself during the training process.

Combing the weight approximation and activation approximation, the whole convolution is given by:

$$Conv(W, R) \approx Con\left(\sum_{m=1}^{M} \alpha_m B_m, \sum_{n=1}^{N} \beta_n A_n\right)$$

$$= \sum_{m=1}^{M}\sum_{n=1}^{N} \alpha_m \beta_n Con(B_m, A_n) \qquad (c)$$

An example of the block structure of the ABC-Net is given as follows.Suppose M=N=3.



The left is the structure of ApproxConv, which approximates the conventional convolution with linear combination of binary convolutions as mentioned in Weight approximation. On the right is the overall block structure of the convolution in ABC-Net. The input is binarized activations using different functions $H_{v_1}, H_{v_2}, H_{v_3}$ passed into the corresponding ApproxConv's and then summed up after multiplying their corresponding β's. For this example, there are 9 BinConv's in total.

2.2 Key Results of the Original Paper

The dataset used for experiments here is Imagenet.

1. As the number of binary weight bases (M) and number of ApproxConv (N) increase, the prediction accuracy gets close to that of the full precision model with Resnet18 architecture. Specifically, if M keeps the same, increasing N will increase prediction accuracy and it is similar if M increases while N keeps constant.

2. Using 3 to 5 binary weight bases is adequate to approximate the full precision weights for Resnet18, Resnet34 and Resnet50 architectures.

3. By employing 5 binary activations and 5 binary weight bases, the Top-1 and Top-5 accuracy gaps caused by binarization are reduced to around 5% on ImageNet for all three architectures.

The results are displayed in the following table.

Table 2: Prediction accuracy (Top-1/Top-5) for ImageNet with different choices of M and N in a ABC-Net (approximate weights as a whole). "res18", "res34" and "res50" are short for Resnet-18, Resnet-34 and Resnet-50 network topology respectively. M and N refer to the number of weight bases and activations respectively.

Network	M-weight base	N-activation base	Top-1	Top-5	Top-1 gap	Top-5 gap
res18	1	1	42.7%	67.6%	26.6%	21.6%
res18	3	1	49.1%	73.8%	20.2%	15.4%
res18	3	3	61.0%	83.2%	8.3%	6.0%
res18	3	5	63.1%	84.8%	6.2%	4.4%
res18	5	1	54.1%	78.1%	15.2%	11.1%
res18	5	3	62.5%	84.2%	6.8%	5.0%
res18	5	5	65.0%	85.9%	**4.3%**	**3.3%**
res18	Full Precision		69.3%	89.2%	-	-
res34	1	1	52.4%	76.5%	20.9%	14.8%
res34	3	3	66.7%	87.4%	6.6%	3.9%
res34	5	5	68.4%	88.2%	**4.9%**	**3.1%**
res34	Full Precision		73.3%	91.3%	-	-
res50	5	5	70.1%	89.7%	**6.0%**	**3.1%**
res50	Full Precision		76.1%	92.8%	-	-

3. Methodology (of the Students' Project)

In this paper, we decided to reconstruct the ABC layer with the same method and structure used in the original paper. We initialized the ABC layer as a subclass of keras.layers.Conv2D, and then overrode the method convolution_op.

We firstly created linear combinations of weights. We generated B's as the definition indicated in the original paper. Here we reset the gradient of $\frac{dB}{dW} = 1$ by using the custom_gradient method in tensorflow. This was implemented as a straight-through estimator. Then we generated alpha's by solving minimization of linear regression problem $min_{\alpha} J(\alpha) = ||w - B\alpha||^2$. Then we could approximate kernel by linear combination of M binary filters $B_1, \ldots B_M$

$s.t. W \approx \alpha_1 B_1 + \alpha_2 B_2 + \ldots + \alpha_M B_M$.

Then in the second step, we constructed our real-value inputs into linear combinations of multiple binary activations. Firstly, we randomly generated values for beta's and v's by definition indicated in the original paper. Then we estimated inputs R s.t.

$R \approx \beta_1 H_{v_1}(R) + \beta_2 H_{v_2}(R) + \ldots + \beta_N H_{v_N}(R)$ where H is

$H_v(R) = 2I_{R+v \geq 0.5} - 1$ (d) which is equivalent to that defined in the original paper. Unlike α 's, we used add_weight method by adding new variables β 's and v's in the initialization of the ABC layer. Thus β and v can be tuned by the network itself during the training process. Here we reset the gradient of $\frac{dH}{dv} = 1$ and $\frac{dH}{dR} = 1$ by using the custom_gradient method in tensorflow.

In the third step, we combined the binary activations and binary weights together and for each combination of activation and weight, we created a bitwise convolution. In the end, we added M*N convolutions to get the whole convolution scheme. Thus we created an ABC layer by overriding the convolution_op method with the schemes above.

For the experiment part, we utilized *LeNet, AlexNet and VGG16* neural networks. And we used cifar100 in tensorflow as our dataset, which is much smaller and simpler. We experimented on the full-precision version which uses the original Con2D layers. We also experimented with our own version by replacing each Con2D layer with our created ABC layer. For each experiment, we explored the configuration of different combinations of number of weight bases and activations. We chose M to be 1,3,5 and N to be 1,3,5 where M,N refers to the number of weight bases and activations respectively. Here, for simplification, the parameters' settings of each ABC layer are the same as the original Conv2D layer, such as padding and strides. Also, we ran with the same epochs, batch size and Adam optimizer with the same learning rate for both original and ABC nets. We chose the best test prediction accuracy among 5 runs for each configuration of M,N of each model.

3.1. Objectives and Technical Challenges

We would like to reproduce the results and check if the same strategies can be replicated in different ANN architectures while reproducing similar results. The original paper utilized the ImageNet dataset for training. However, the ImageNet dataset is tremendously huge in its size(more than 1.2 million images) which is difficult for us to train the model due to time and memory constraints. Therefore, we decided to use the cifar100 dataset in keras which contains 50000 train samples and 10000 test samples which is much smaller than the ImageNet. Similarly, the original paper utilized Resnet18, Resnet34 and Resnet50 as network topology. However, resnet architecture is complicated and large for us to train the dataset. Therefore, we replaced it with some small and classical deep neural networks: *LeNet, AlexNet and VGG16* to fit on our created ABC layer and dataset.

Another challenge for us occurred when we reconstructed the forward and backward approaches for our weights and activations. Some part of the partial derivative in the backpropagation would be 0 if we deducted from the defined function in the original paper directly without straight-through estimators. To avoid this, we had to define the gradient ourselves otherwise it may encounter some problems in backprogopation process. We solved

this problem by using the custom_gradient method, which was implemented as straight-through estimators.

3.2.Problem Formulation and Design Description

The following training algorithm is the same as the original paper. We optionally apply MaxPooling and BatchNormalization after each convolution layer. There may be dense layers at the end of nets in the experiments and we don't consider them here.

Algorithm for training an L-layer ABC-Net
Requirements: inputs R and targets, number of binary weight bases M, number of binary activations N, initial weights W (kernel) and learning rate a.
Notation: Conv() and BackConv() specify how to do convolution and how to backpropagate through the convolution. Update() specifies how to update the parameters when their gradients are known.
Algorithm:
(1)Computing the parameters gradients:
(1.1) Forward propagation
for l = 1 to L do
Compute α_m and B_m, m = 1,2,...,N with equation (a),(b)
$a^l \leftarrow$ Con(W^L, R^{L-1}) with (c)
if l < L **then**
$A^l_n \leftarrow H_{v_n}(a^l)$, n = 1,2,...,N with (d)
end if
end for
(1.2) Backward propagation
Compute $g_{a^L} = \frac{dC}{da_L}$
for l = L to 1 do
if l < L then
$g_{a^l} \leftarrow \sum_{n=1}^{N} \beta_n * g^l_{A_n} I_{0<=a^l-v_n<=1}$
end if
$g_{v_m}, g_{\beta_m}, g_{B^l_m} \leftarrow$ BackConv(g_{a^l}, B_m^{l-1}, A_m^{l-1})
end for
(2)Accumulating the parameters gradients:
for l = 1 to L do
With $g_{B^l_m}$ known, compute g_{w^l} using (e)
$W^l \leftarrow$ Update(W^l, a , g_{w^l})
$\beta_m \leftarrow$ Update(β_m, a , g_{β_m}), m= 1,2,...,M
$v_m \leftarrow$ Update(v_m, a , g_{v_m}), m= 1,2,...,M
end for

4. Implementation

In this section, we introduce our data and deep learning network structures used in experiments.

4.1 Data

CIFAR-100 consists of 60000 32 * 32 colour images in 100 classes. Each class contains 600 images. There are 500 training images and 100 testing images per class. The 100 classes are grouped into 20 superclasses. Each image comes with a "fine" label (the class it belongs to) and a "coarse" label (the superclass it belongs to).

We felt that having only 500 instances per class in training is inadequate given the complexity of image recognition. Therefore, we used each image's superclass as its label. That is, in our experiment, we had 20 labels and 2500 instances per class. The labels we had are things like "aquatic mammals", "fruit and vegetables", "vehicles" and etc.

In our experiments, we split the data into training data, validation data and test data. There are 45000,5000,10000 images of each group.

4.2 Deep Learning Network

For small-sized models, we used LeNet for our experiments. The specific structure is the following: (It is the same for full precision and ABC nets except convolutional layers)

LeNet
Conv (128, 5*5, "same", "relu")
MaxPooling (2*2)
BatchNormalization ()
Conv (128, 5*5, "same", "relu")
MaxPooling (2*2)
BatchNormalization ()
Flatten ()
Dense (1024, "relu")
Dense (1024, "relu")
Dense (20, "SoftMax")

For large-sized models, we used VGG16 for our experiments. The specific structure is the following on the right: (It is the same for full precision and ABC nets except convolutional layers)

VGG 16
Conv (64, 3*3, “same”, “relu”)
BatchNormalization ()
Conv (64, 3*3, “same”, “relu”)
BatchNormalization ()
MaxPooling (2*2)
Conv (128, 3*3, “same”, “relu”)
BatchNormalization()
Conv (128, 3*3, “same”, “relu”)
BatchNormalization ()
MaxPooling (2*2)
Conv (256, 3*3, “same”, “relu”)
BatchNormalization ()
Dropout (0.4)
Conv (256, 3*3, “same”, “relu”)
BatchNormalization ()
Conv (256, 3*3, “same”, “relu”)
BatchNormalization ()
MaxPooling (2*2)
Conv (512, 3*3, “same”, “relu”)
BatchNormalization ()
Conv (512, 3*3, “same”, “relu”)
BatchNormalization ()
Conv (512, 3*3, “same”, “relu”)
BatchNormalization ()
MaxPooling (2*2)
Conv (512, 3*3, “same”, “relu”)
BatchNormalization ()
Dropout (0.4)
Conv (512, 3*3, “same”, “relu”)
BatchNormalization ()
Conv (512, 3*3, “same”, “relu”)
BatchNormalization ()
MaxPooling (2*2)
Flatten ()
Dense (128, “relu”)
Dense (20, “SoftMax”)

For medium-sized models, we used AlexNet for our experiments. The specific structure is the following: (It is the same for full precision and ABC nets except convolutional layers)

AlexNet
Conv (128, 5*5, “same”, “relu”)
MaxPooling (2*2)
Conv (256, 3*3, “same”, “relu”)
MaxPooling (2*2)
Conv (512, 5*5, “same”, “relu”)
Conv (256, 5*5, “same”, “relu”)
Conv (256, 5*5, “same”, “relu”)
MaxPooling (2*2)
Dropout (0.5)
Flatten ()
Dense (1024, “relu”)
Dense (256, “relu”)
Dense (64, “relu”)
Dense (20, “SoftMax”)

4.3 Software Design

We took full advantage of pre-implemented APIs in tensorflow and the “convolution_op” method in class keras.layers.Conv2d in the recently released tensorflow 2.7.0. We used the tf.custom_gradient decorator to wrap our forward and backward functions and defined our own “ABCConv” class which inherits “layers.Conv2d”. In this class, by overriding the “convolution_op” method, we implemented one convolutional step as described in the pseudo code (3.1). By doing this, we did not have to implement lower level logic like caching the intermediates, computing loss etc. More importantly, by inheritance, our “ABCConv” layer could directly interact with other pre-defined APIs in tensorflow such as “keras.Sequential” and “MaxPooling”, which saved us a ton of time and avoided unnecessary mistakes.

For the full precision version or each combination of M,N of each neural network, we implemented it in a single jupyter notebook respectively. Each notebook has the same procedures, including data loading, “ABCConv” class construction, model construction and model training plus evaluation.

5. Results

5.1 Project Results

The experiment details are shown in section3. The following table presents the results of ABC-Net with different configurations. Here top1 and top3 are test accuracies and top1-gap as well as top3-gap are gaps to full prediction accuracies.

Network	M	N	Top1	Top3	Top1 gap	Top3 gap
VGG16	3	1	5.00%	15.00%	58.80%	68.80%
	3	3	19.50%	40.50%	44.30%	43.30%
	3	5	26.20%	50.60%	37.60%	33.20%
	5	3	27.30%	52.40%	36.50%	31.20%
	5	5	25.40%	50.40%	38.40%	33.40%
	Full Precision		63.80%	83.80%	-	-
AlexNet	3	3	5.00%	15.00%	42.70%	55.60%
	Full Precision		46.70%	70.60%	-	-
LeNet	3	1	23.30%	46.20%	17.50%	19.90%
	3	3	32.60%	57.10%	8.20%	9.00%
	3	5	30.20%	54.00%	10.60%	12.10%
	5	5	28.20%	52.70%	12.60%	13.40%
	Full Precision		40.80%	66.10%	-	-

Regarding training times, the full precision of Lenet, AlexNet and VGG16 have a shorter time, no more than 25 minutes, with VGG16 the longest. While with binary weight and binary activations, the training time is much longer. It takes about 3.5 and 4 hours for VGG16, 2.5 hours for AlexNet and 30 minutes for Lenet.

5.2 Comparison of the Results Between the Original Paper and Students' Project

Generally, both the original paper and our project have shown that, as M and N increase, the top1 and top3 accuracies get close to the full precision accuracy. In other words, we get a better approximation of the weight layer.

However, there are 3 differences:

1. The top1 and top3 gaps in our project are significantly higher than the gaps in the original paper. The original paper successfully reduced the gaps to as small as 5%, however, in our project, we can only get half the full precision when using binary approximation.
2. There are 2 anomalies in our results: VGG16 with M=3 and N=1 and AlexNet with M=3 and N=3. Both gave 5% top1 accuracy and 15% accuracy, which is no better than random guessing, considering that there are only 20 labels.
3. We observe overfitting in our results. VGG16 starts to overfit after M=5 and N=3. Specifically, top1-gap and top3-gap with M=5,N=5 are higher for those of the configuration with M=3,N=5 and M=5,N=3. While LeNet starts to overfit after M=3 and N=3. Specifically, top1-gap and top3-gap with M=3,N=5 are higher for those of the configuration with M=3,N=3. And it is similar for the combination of M=5,N=5

5.3 Discussion of Insights Gained

1. The original paper proposed that v and β are trainable and in their experiments, they set initial values of v and β to be hyperparameters. This fact is said in table S4 of the paper. In our project, however, the initial values of v and β are following random normal distributions. This may introduce more noises in training. Also, we used a different dataset, CIFAR-100. And for simplification, we did not run many epochs since our time is limited. These are the possible reasons for the big gaps between full accuracy and our approximation accuracy. We could set initial values of v and β for improvements.
2. We suspect that the model stuck at some local extremum during training. As mentioned above, training v and β with random normal initializers makes the results more volatile and therefore, this case is more likely to take place in our project. When it happens, the model is not learning and its accuracy just stays unchanged. Maybe we could try a larger learning rate for some particular cases to jump out of the local extremum.
3. To some extent, this is what we have expected. LeNet is a very tiny model as compared to VGG16. Due to its small size, it has much lower model complexity. Therefore, LeNet should overfit with smaller M and N.

Lastly, for each combination of M,N of a network, we used the same parameters like padding, stride and num_of_filters at the same layer and same batch size as well as number of epochs during training for the purpose of better comparison. However, the original paper does not mention if they conducted experiments under the same conditions/hyperparameters. This might be another source of difference.

6. Future Work

We may want to do the experiments here without using built-in classes in tensorflow. That is to write everything in the lower level including convolutional layers, model class, stepping and training functions just like what we have done in assignment 2. However, we feel it is worthwhile afterwards to finish the program especially given the unstable results we mentioned earlier.

As we relied on the tensorflow API to do training in this project, we did not have full control or access to the intermediate results. What happened during training was merely a black box for us. Even though we do realize that the approximated convolutional layers were not performing as well as expected in terms of accuracy, the results did show the right trend. If we do implement the model from a lower level, we would have a better chance of figuring out why the issues occur by storing or outputting some intermediate results during training.

Besides, we could set initial values of v's and β's as hyperparameters like what the original paper did in experiments. We expect that there may be better results with a good setting.

7. Conclusion

This project tried to examine if using a combination of binary convolutional layers to approximate normal convolutional layers can obtain similar accuracies as full precision accuracies in LeNet, AlexNet and VGG16. In our experiments, the accuracies of LeNet and VGG16 get close to the full precision accuracies as N, the number of ApproxConv and M, the number of BinConv increase, which are consistent with the original paper. However, there are still some anomalies for these two nets, and the accuracy of AlexNet stucks at 5% no matter how we changed N and M. This might be due to difference of parameters setting, random initialization of v's and β's and model sizes. In conclusion, we have seen the right trends and it seems that the strategy of approximating convolutional layers using a linear combination of binary layers is replicable in many ANN architectures. We would try some more models and set initial values of v's and β's as hyperparameters for future work.

8. Acknowledgement

[1]Customizing the convolution operation of a Conv2D layer:
https://keras.io/examples/keras_recipes/subclassing_conv_layers/

9. References

[1]Project code github link:
https://github.com/ecbme4040/e4040-2021fall-project-KFW-hk3127-hf2431-jw4000

[2] X.Lin, C.Zhao, W.Pan, "Towards Accurate Binary Convolution Neural Network", DJI Innovations Inc, Shenzhen, China.

10. Appendix

10.1 Individual Student Contributions in Fractions

	hk3127	hf2431	jw4000
Last Name	Kan	Fang	Wang
Fraction of (useful) total contribution	1/3	1/3	1/3
What I did 1	Modeling for part of VGG models (M=5,N=3 M=5,N=5 and Full Precision)	Modeling for AlexNet and Lenets	Modeling for part of VGG models (M=3,N=1 M=3,N=3, M=5,N=5)
What I did 2	Report writing (section4.2 ,section4.3 ,section5 to section10)	Report writing (section1, section4.1, 4.2)	Report writing (Abstract, section2, section3)
What I did 3			