Proceedings of the 4th International Conference on Software Engineering and Machine Learning
DOI: 10.54254/2755-2721/2026.AS32438

Comprehensive Analysis of Network Depth and Numerical Precision on MNIST Classification: Training Dynamics and Deployment Efficiency

Yuhao Bi

School of Integrated Circuit, Chongqing University of Posts and Telecommunications, Chongqing, China
2845181394@qq.com

Abstract: Today's deep learning field is developing rapidly, and its focus is no longer limited to pursuing the ultimate accuracy of models on large servers, but paying more and more attention to how to improve model efficiency in order to deploy it to edge devices. In this study, we focus on how adjusting the two key knobs of network depth and numerical precision affects the performance of neural networks. We used the classic MNIST dataset as a test benchmark to examine fully connected neural networks with depths ranging from 2 to 10 layers.In the experimental part, we compared traditional FP32 training with FP16 mixed-precision training, and further studied the impact of introducing INT8 dynamic quantization after training. The results show that FP16 mixed-precision training is very effective: compared with the FP32 model, its accuracy loss is minimal (usually less than 0.2%) and can benefit from hardware acceleration. At the same time, in the simulated deployment experiment, we found that INT8 dynamic quantization can greatly reduce the model size by about 70–75%, while the decrease in accuracy is very limited (only about 0.5–1.2%). This result confirms our conjecture: in resource-constrained environments such as Internet of Things devices or mobile processors, reducing numerical precision is not only practical, but also an efficient means for large-scale applications.

Keywords: Deep Learning, Floating Point Precision, MNIST, Model Quantization, Dynamic Quantization, Mixed Precision Training.

1. Introduction

We are now having a really rapid speed in deep learning development. During this period, deep learning algorithms have achieved performance close to or even surpassed human beings in many fields, and their scope of application is also very wide, such as medical diagnosis and language processing. However, this rapid development also brings an obvious problem: the scale of the model is getting bigger and bigger. As Han et al. [1] pointed out in the study, with the increasing complexity of neural networks, we will encounter many difficulties when we try to deploy these models from data centers to everyday devices (such as smartphones or embedded sensors).

In an edge device environment, we do not have unlimited computing power and storage resources like data centers. Therefore, in this case, the focus of the research is no longer just to improve the

© 2026 The Authors. This is an open access article distributed under the terms of the Creative Commons Attribution License 4.0 (https://creativecommons.org/licenses/by/4.0/).

DOI: 10.54254/2755-2721/2026.AS32438

accuracy rate by 0.1%, but to find a more appropriate balance between the intelligence and operational efficiency of the model, so that the model can complete the task without taking up too many computing resources.

This study mainly analyzes two factors that help the system balance between performance and efficiency: network depth and numerical accuracy in the calculation process.

For a long time, it has been widely believed that "the deeper the network, the better the effect".With the increase in the number of network layers, the model can learn more abstract and hierarchical feature representation. However, is this assumption valid for all tasks? Or,when the network depth reaches a certain level, the performance improvement will gradually stop. And further increasing the number of network layers will only lead to the expansion of the model scale and calculate the increase in cost?

At the same time, numerical precision is also a matter that needs attention. Most research models usually use 32-bit floating-point numbers (FP32) for training, as this is a relatively safe choice. However, as Gholami et al. [2] emphasized in their review study on quantization methods, using 32-bit precision for each weight is often an overdesign. It's like measuring the distance to the grocery store in millimeters, which sounds a bit exaggerated, because in actual situations, such high precision is not necessary.

Therefore, reducing numerical precision (such as 16-bit floating-point numbers FP16, or 8-bit integers INT8) becomes a solution. Theoretically, this can reduce memory usage by half or even a quarter while maintaining the same performance. But is this really the case?

To answer those questions more formally, we took a step back and used the MNIST data set. Though it may seem simplistic in comparison to today's titans such as ImageNet,...MNIST is a perfect "con-trolled benchmark", or the "fruit fly" for ML research (see e.g., LeCunet al. [3]). The simplicity makes it possible for us to remove the noise of complicated data augmentation or architectures and get straight into what happens when the hardware meets the math.

In the present paper, we do not only run codes but also try to address these three questions:

1. **Analysis of Network Depth:** In a simplified feature space, does increasing the network depth from 2 layers all the way to 10 actually help the model learn better, or does it just make it harder to train?
2. **Impact of Numerical Precision:** Can we drop from FP32 to FP16 without the model collapsing? How robust is the training process?
3. **The Real-World Cost:** When we compress the model to INT8 for deployment, exactly how much space do we save, and what is the "tax" we pay in terms of accuracy?

2. Methodology

2.1. Dataset and preprocessing strategy

Our experimental foundation is the MNIST dataset [3], a collection of 70,000 grayscale images of handwritten digits, each sized at 28×28 pixels. Before feeding these into our network, we perform a typical pre-processing operation which normalizes the pixel intensities by dividing them into [0, 1] ranges as this makes training more stable due to the gradient calculations in back propagation . The normalized images are then passed through:each image is flattened, i.e., reshaped to obtain an one-dimensional vector of size 784.

What's more special is that I don't use any data enhancement technology. Although research based on MNIST will improve its generalization ability through operations such as rotation, translation or scaling, this study deliberately excludes these transformations. It aims to explore how network depth

DOI: 10.54254/2755-2721/2026.AS32438

and numerical accuracy affect training behavior and deployment efficiency under controlled conditions. If additional preprocessing steps are added, it will be difficult to judge whether the performance change comes from the model design itself or the adjustment of the input sample. Therefore, after completing the data standardization and leveling processing, we maintain the original form of the data set to ensure that the comparison under different experimental conditions is as fair and transparent as possible.

```
from torch.utils.data import DataLoader

train_dl = DataLoader(mnist_train, batch_size=100, shuffle=False)

dataiter = iter(train_dl)
images, labels = next(dataiter)
viz = torchvision.utils.make_grid(images, nrow=10, padding=2).numpy()

fig, ax = plt.subplots(figsize=(8, 8))
ax.imshow(np.transpose(viz, (1, 2, 0)))
ax.set_xticks([])
ax.set_yticks([])
plt.show()
```

Listing 1: Code for MNIST data loading and preprocessing.

Listing 1 shows the code snippet used for MNIST data loading and preprocessing in our experiment.The samples themselves, as seen in Figure 1, are simple but varied enough to test the network's generalization capabilities.



Figure1. Visualization of standard MNIST handwritten digit samples.

DOI: 10.54254/2755-2721/2026.AS32438

2.2. Architectural configurations and depth variations

In order to study the depth effect in a controlled way, we have built a series of neural networks that are fully connected to PyTorch. Instead of evaluating a single architecture, a model with 2 to 10 non-representative layer depths was formed. This design allows you to compare lower-layer and relatively deep networks under the same data set, optimizer and training process.

All our models have several fixed parameters. The width of the cache is fixed on 128 units in each layer. The size is sufficient for MNIST classification, and the model is lightweight and easy to learn. As an activation function, ReLU is used to simplify the stable operation of multi-layer networks. Adam was chosen as the optimizer because it provides a more reliable approximation, especially in the preliminary test of the depth model, which is more reliable than SGD. Since these elections have not changed, the main variable in this part of the experiment is network depth.

Table 1 summarizes the configurations we tested.

Table 1: Summary of Fully-Connected Neural Network Architectures

Attribute	Specification
Input Dimension	784 (Flattened 28×28)
Hidden Layer Depth	$\{2, 4, 6, 8, 10\}$
Hidden Layer Width	128
Activation	ReLU
Weight Initialization	Xavier/Glorot Uniform
Output Layer	10-way Linear with Softmax

2.3. Numerical precision and quantization implementation

This part of this study focuses on the role of digital accuracy in teaching and delivery. We have considered three parameters: FP32 standard learning, FP16 mixed precision learning and dynamic quantification after INT8 learning. These three cases represent different stages of the actual process; in-depth learning; development; resources

FP32 is used as a reference parameter. In this format, the parameters and averages are recorded as 32-bit floating points, which provides stable numerical behavior and is the standard choice for many training tasks. However, the disadvantage is that it requires more memory and data movement during the calculation process.

For the FP16 experiment, we use mixed-precision training instead of converting the entire training process to pure FP16. In practice, the most computationally intensive matrix operations are carried out in FP16, while numerically sensitive operations remain in higher precision to preserve stability. In addition, gradient scaling is introduced to reduce the risk of underflow during backpropagation, following the mixed-precision training strategy proposed in e.g. Micikevicius, P. [4]. Using this parameter, we can test whether lower accuracy can accelerate the drive without causing major damage.

For deployment, we further applied post-training dynamic quantification to convert the trained model into INT8 form. In this method, the linear layer weight is stored in 8-bit integer format, and the activated scale is dynamically processed during the reasoning process. One of the advantages of this method is that it does not need to retrain the model, which makes it easy to apply after the main training stage. Therefore, this is a practical choice when model size and reasoning efficiency are more

DOI: 10.54254/2755-2721/2026.AS32438

important than absolute numerical accuracy.The implementation details of FP16 gradient scaling and post-training INT8 dynamic quantization are shown in Listing 2.

```
# FP16 Mixed Precision Training
scaler = torch.cuda.amp.GradScaler()
for xb, yb in train_loader:
    # Autocast allows the model to determine which ops can run in FP16
    with torch.cuda.amp.autocast():
        loss = F.cross_entropy(model(xb), yb)

    # Scale loss to prevent gradient underflow in FP16
    scaler.scale(loss).backward()
    scaler.step(optimizer)
    scaler.update()

# INT8 Dynamic Quantization (Post-Training)
# This converts FP32 weights to INT8, reducing file size dramatically
qmodel = torch.ao.quantization.quantize_dynamic(
    model.cpu(), {torch.nn.Linear}, dtype=torch.qint8
)
```

Listing 2: Implementation of FP16 Scaling and INT8 Quantization

3. Results and analysis

3.1. Training dynamics: The law of diminishing returns

The training curves show that increasing depth does not continuously improve performance on MNIST. The 2-layer and 4-layer models converged quickly and reached strong results within a small number of epochs. When the depth increased to 8 or 10 layers, the optimization process became slower and less stable, but the final improvement was limited.

This behavior is closely related to the difficulty of the task itself. MNIST is relatively simple compared with more complex image datasets, so a moderate-size multilayer perceptron is already capable of capturing most of the useful patterns. Once that capacity is sufficient, adding more layers mainly increases optimization difficulty rather than providing new representational benefits. In other words, deeper models introduce extra training burden, but on this dataset the additional depth does not translate into a meaningful gain.This trend is illustrated in Figure 3, where deeper models do not produce a meaningful reduction in test loss and instead show less stable convergence.

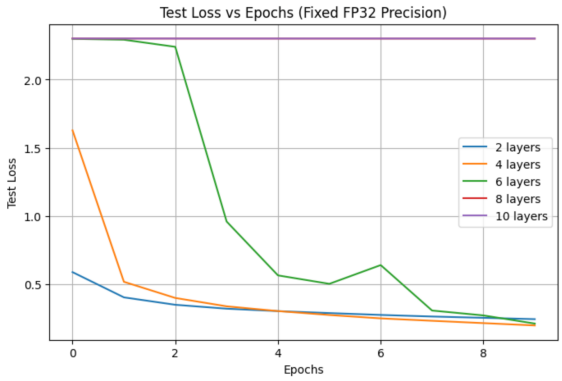


Figure2. Test Loss vs. Epochs for varying network depths (Fixed FP32 Precision).

DOI: 10.54254/2755-2721/2026.AS32438

3.2. Stability across the precision spectrum

When we switched gears to test precision, the results were incredibly encouraging. **Figure 3** overlays the loss curves for FP32 and FP16 training. If you look closely, you can barely tell them apart.

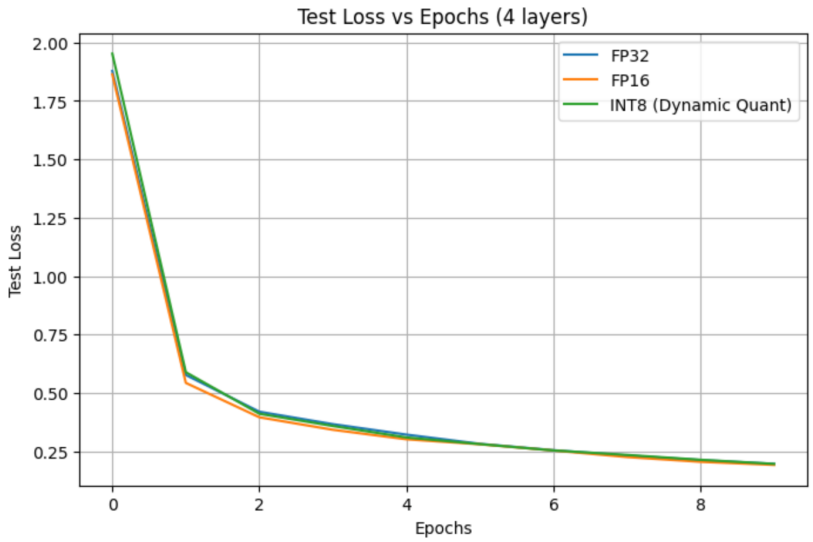


Figure3. Comparison of training stability across FP32, FP16, and INT8 (fixed depth).

3.3. The efficiency win: size vs. performance

The real headline of this study comes from the quantization results shown in **Figure 4**.

When we convert our models to INT8 using dynamic quantization, the file size drops off a cliff—in a good way. We consistently observed a **70% to 75% reduction in model size**. For a mobile developer, this is huge. It means an app update that downloads a model is four times smaller.

However, how much did it cost in terms of accuracy? This is where we were surprised by the low tradeoff: across all depths, we observed that INT8 models had a loss of approximately 0.5-1.2% in accuracy. For most real world applications,trading1% accuracy for a 4x reduction in size and memory is a deal anyone would take. Which is consistent to what we find in [5],which suggests the fact that neural nets are highly over-parameterized and they can withstand a large amount of “noise” (quantization error) on its weights before it breaks. Figure 5: Quantitative results for the FP32, FP16, and INT8 model at various depth.

DOI: 10.54254/2755-2721/2026.AS32438

```
===== FP32 Table =====
   Layers  Train Loss  Train Acc  Test Loss  Test Acc
0       2    0.265731   0.924767   0.251932    0.9285
1       4    0.211669   0.940117   0.207260    0.9375
2       6    0.246884   0.931417   0.236799    0.9344
3       8    2.295906   0.112367   2.291639    0.1135
4      10    2.301225   0.112367   2.301012    0.1135

===== FP16 Table =====
   Layers  Train Loss  Train Acc  Test Loss  Test Acc
0       2    0.258702   0.927233   0.247399    0.9323
1       4    0.211120   0.938683   0.203089    0.9420
2       6    0.216560   0.938983   0.222046    0.9352
3       8    2.300509   0.112367   2.300166    0.1135
4      10    2.301238   0.112367   2.301114    0.1135

===== INT8 Table =====
   Layers  Train Loss  Train Acc  Test Loss  Test Acc
0       2    0.263810   0.925683   0.250597    0.9301
1       4    0.215414   0.938800   0.202977    0.9418
2       6    0.234052   0.935367   0.266039    0.9224
3       8    2.285448   0.120983   2.264645    0.1727
4      10    2.301219   0.112367   2.301045    0.1135
```

Figure4. Summary of metrics for FP32, FP16, and INT8 models across different depths.

4. Discussion and theoretical context

This result helps to distinguish between two related but not exactly the same goals: improving training efficiency and reasoning efficiency. The mixing accuracy is mainly beneficial to the initial training stage. On the supported hardware, it can reduce memory pressure and accelerate the calculation speed of matrix operations. However, these advantages do not automatically apply to every deployment environment, especially when reasoning is performed on a general-purpose CPU or low-power device.

In contrast, the relationship between INT8 and deployment is more direct. It can efficiently process integer operations in many edge processors, and quantification models also require less storage and memory bandwidth. From this perspective, quantitative gain is not limited to model compression; it can also make reasoning more practical on devices with limited resources.

Another important result is that the trained network still has a considerable tolerance for the reduction of parameter accuracy. Even after quantification, the model can still maintain strong performance. This shows that the parameter space of learning contains a certain amount of redundancy, so the small disturbances introduced quantitatively will not immediately destroy the useful structure captured during the training process. For simple classification tasks, such as MNIST, this robustness makes low-precision deployment particularly good.

5. Conclusion and future work

This study explores how network depth and numerical accuracy affect the fully connected network on MNIST. Judging from the experimental results, three points are more obvious. First of all, increasing the number of layers to a medium level does not bring obvious benefits to this task, but it does make optimization more difficult. Secondly, FP16 hybrid precision training is very close to FP32 in performance, which indicates that when the training program is configured at that time, the reduced accuracy can be safely used. Third, INT8 dynamic quantification greatly reduces the size of the model

DOI: 10.54254/2755-2721/2026.AS32438

and has limited impact on accuracy, making it the most useful strategy for deployment optimization in this work.

There are still several directions worth exploring. One is static quantization, which can further improve the efficiency of reasoning when the calibration data is available. The other is quantitative perception training, which introduces quantitative effects during training rather than after training. It helps the model to adapt to low-precision constraints more effectively. In addition, knowledge distillation can be combined with lightweight or quantitative models so that smaller networks can inherit useful behaviors from stronger teacher models. These extensions will shift this study from controlled MNIST settings to broader and more realistic deployment scenarios.For completeness, the full experimental workflow is summarized in Appendix.

References

[1] Han, S., Pool, J., Tran, J., and Dally, W. Learning both Weights and Connections for Efficient Neural Network. In *Advances in Neural Information Processing Systems (NIPS)*, 2015.

[2] Gholami, A., Kim, S., Dong, Z., Yao, Z., Gholami, M. W., and Keutzer, K. A Survey of Quantization Methods for Efficient Neural Network Inference. *Low-Power Computer Vision*, 2021.

[3] LeCun, Y., Bottou, L., Bengio, Y., and Haffner, P. Gradient-based learning applied to document recognition. *Proceedings of the IEEE*, 86(11):2278–2324, 1998.

[4] Micikevicius, P., Narang, S., Alben, J., Diamos, G., Elsen, E., Garcia, D., et al. Mixed Precision Training. *arXiv preprint arXiv:1710.03740*, 2017.

[5] Nagel, M., Amjad, R. A., van Baalen, M., Louizos, C., and Blankevoort, T. A White Paper on Neural Network Quantization. *arXiv preprint arXiv:2106.08295*, 2021.

Appendix: Full Pipeline Structure

For clarity, the overall experimental procedure is summarized below in pseudocode form. Each network depth was evaluated under the same workflow.

```
for depth in [2, 4, 6, 8, 10]:
    # 1. Initialize model with specific depth
    # We maintain constant width to isolate depth effects
    model = BuildModel(layers=depth, units=128)

    # 2. FP32/FP16 Training Comparison
    # We record loss curves, training time, and peak memory usage
    # This helps us understand the training-time trade-offs
    train_metrics = RunTraining(model, precisions=['fp32', 'fp16'])

    # 3. Deployment Optimization via Post-Training Quantization
    # Convert the trained model to INT8 representation
    quantized_model = QuantizeDynamic(model)

    # 4. Inference Performance Evaluation
    # Measure accuracy on the test set and model file size
    int8_metrics = Evaluate(quantized_model)

    # 5. Systematic Logging for Comparative Analysis
    # All data is saved to CSV for the final plots
    LogAll(train_metrics, int8_metrics)
```