Enhancing Efficiency in Handwritten Digit Classification

A MINI PROJECT I REPORT

Submitted by

AJEEM KHAN.K (71812201013)

ENITHA.S (71812201047)

SWETHA.T (71812201239)

in partial fulfillment for the award of the degree

of

BACHELOR OF ENGINEERING

in

COMPUTER SCIENCE AND ENGINEERING

SRI RAMAKRISHNA ENGINEERING COLLEGE

[Educational Service: SNR Sons Charitable Trust]
[Autonomous Institution, Reaccredited by NAAC with ' A+' Grade]
[Approved by AICTE and Permanently Affiliated to Anna University, Chennai] [ISO 9001:2015 Certified and All Eligible Programmes Accredited by NBA] Vattamalaipalayam, N.G.G.O. Colony Post,

COIMBATORE – 641 022

ANNA UNIVERSITY : CHENNAI 600 025

APRIL 2024

SRI RAMAKRISHNA ENGINEERING COLLEGE

BONAFIDE CERTIFICATE

DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING

MINI PROJECT I – APRIL 2024

This is to certify that the project entitled

ENHANCING EFFICIENCY IN HANDWRITTEN DIGIT CLASSIFICATION

is the bonafide record of Mini Project I done by

AJEEM KHAN.K(71812201013)
ENITHA. S (71812201047)
SWETHA.T (71812201239)

of B.E. Computer Science and Engineering during the year 2023-2024.

who carried out the Mini Project I under my supervision, certified further that to the best of my knowledge the work reported herein does not form part of any other thesis or dissertation on the basis of which a degree or award was conferred on an earlier occasion on this or any other candidate.

_____ _____

Mrs.A.Mahalakshmi,
PROJECT GUIDE
Assistant Professor (Sr.Gr.)
Computer Science and Engineering,
Sri Ramakrishna Engineering College,
Coimbatore-641022.

Dr.A.Grace Selvarani,
HEAD OF THE DEPARTMENT
Professor,
Computer Science and Engineering,
Sri Ramakrishna Engineering College,
Coimbatore-641022.

Submitted for the Project Viva-Voce Examination held on _____

_____ _____

Internal Examiner **External Examiner**

DECLARATION

We affirm that the Mini Project I titled **" ENHANCING EFFICIENCY IN HANDWRITTEN DIGIT CLASSIFICATION "** being submitted in partial fulfillment for the award of Bachelor of Engineering is the original work carried out by us.It has not formed the part of any other project work submitted for award of any degree or diploma, either in this or any other University.

(Signature of the Candidates)

AJEEM KHAN.K **(71812201013)**

ENITHA.S **(71812201047)**

SWETHA.T **(71812201239)**

I certify that the declaration made above by the candidates is true.

(Signature of the guide)

Mrs.A.MAHALAKSHMI,

Assistant Professor (Sr.Gr.),

Department of CSE

ACKNOWLEDGEMENT

We express our gratitude to **Sri.D. LAKSHMINARAYANASWAMY,** Managing Trustee, **Sri. R. SUNDAR,** Joint Managing Trustee, SNR Sons Charitable Trust, Coimbatore for providing excellent facilities to carry out our project.

We express our deepest gratitude to **Dr. N. R. ALAMELU,** Principal, for her valuable guidance and blessings.

We thank **Dr. A. GRACE SELVARANI,** Professor and Head, Department of Computer Science and Engineering who modeled us both technically and morally for achieving great success in life.

We sincerely thank our Project Coordinator, **Mrs.A.MAHALAKSHMI ,**Assistant Professor (Senior Grade), Department of Computer Science and Engineering for her great inspiration.

Words are inadequate to offer thanks to our respected guide. We wish to express our sincere thanks to **Mrs.A.MAHALAKSHMI,** Assistant Professor (Senior Grade), Department of Computer Science and Engineering, who gives constant encouragement and support throughout this project work and who makes this project a successful one.

We also thank all the staff members and technicians of our Department for their help in making this project a successful one.

TABLE OF CONTENTS

CHAPTER NO	TITLE	PAGE NO.
	ABSTRACTION	**vii**
	LIST OF FIGURES	**viii**
	LIST OF ABBREVIATIONS	**ix**
1	**INTRODUCTION**	**1**
2	**LITERATURE REVIEW**	**2**
	2.1 A Survey on Low-Bit Quantization for Deep Neural Network Inference	2
	2.2 Quantization of Deep Convolutional Networks for Efficient Inference: A Survey	2
	2.3 A Survey on Neural Network Quantization	3
	2.4 A Survey on Quantization Methods for Efficient Neural Network Inference	3
	2.5 Quantization in Deep Neural Networks: A Survey	4
	2.6 A Survey on Neural Network Quantization	4
	2.7 A Survey on Efficient Low-Bit Neural Networks	5
	2.8 Learning both Weights and Connections for Efficient Neural Networks	5
	2.9 Pruning Convolutional Neural Networks for Resource Efficient Inference	6
	2.10 The Lottery Ticket Hypothesis: Finding Sparse, Trainable Neural Networks	6
	2.11 Rethinking the Value of Network Pruning	6
	2.12 parsity-Invariant CNNs	7
3	**SYSTEM ANALYSIS**	**8**
	3.1 Existing System	8
	3.2 Proposed System	8

4	**SYSTEM SPECIFICATION**	**10**
	4.1 Software Requirements	10
5	**PROJECT DESCRIPTION**	**11**
	5.1 Problem Definition	11
	5.2 System Sequence	12
	5.3 Module Description	13
	5.3.1 Module 1- MLP model for Handwritten Digit Dataset	13
	5.3.2 Module 2- Quantization Of Model	13
	5.3.3 Module 3 - Pruning Of Model	14
	5.3.4 Module 4 -Analysis of Classical, Quantized, and Pruned Model	14
6	**SYSTEM IMPLEMENTATION**	**15**
	6.1 System design	15
	6.1.1 Input design	15
	6.1.2 Output design	15
	6.2 Implementation	15
7	**RESULTS AND DISCUSSION**	**17**
	7.1 Experimental Results	17
8	**CONCLUSION AND FUTURE ENHANCEMENTS**	**20**
	8.1 Conclusion	20
	8.2 Future Enhancements	21
9	**APPENDIX**	**23**
	Source Code	23
10	**REFERENCES**	**41**

ABSTRACT

This study explores the application of multilayer perceptron (MLP) neural networks to the task of handwritten digit classification using the handwritten digit dataset. After training a baseline MLP model, several model compression techniques were applied to reduce its memory footprint and computational requirements. These included quantization to reduce precision of weight and activation values, as well as pruning to sparsify the network weights.Different loss functions were investigated during quantization and pruning to assess their impact, including mean squared error (MSE), hinge loss, weighted cross-entropy, and focal loss. Additionally, quantization and pruning were performed using a simple one-hot encoding target representation without an explicit loss function.The compressed models were rigorously evaluated on the handwritten digit classification task using several metrics: average accuracy over training iterations, per-class precision, recall, class-wise loss values, and the overall confusion matrix. These quantitative resultsprovide insights into the tradeoffs between compression ratio, accuracy, and loss function choice for each digit class. The findings highlight effective loss functions and compression strategies for deploying compact MLP models on resource-constrained devices for handwritten digit recognition applications while maintaining high performance.

LIST OF FIGURES

FIGURE NO	NAME OF THE FIGURE	PAGE NUMBER
5.1	System flow	18
7.1	Accuracy Comparison	23

LIST OF ABBREVIATION

ABBREVIATION	EXPANSION
MLP	Multi-Layered Perceptron
MSE	Mean Squared Error

CHAPTER 1
INTRODUCTION

Handwritten digit recognition is a fundamental problem in computer vision and machine learning, with wide-ranging applications in optical character recognition (OCR), document processing, and data entry automation. The task involves classifying images of handwritten digits (0-9) into their corresponding numeric labels. While deep learning models like convolutional neural networks (CNNs) have achieved state-of-the-art performance on this task, deploying such large models on resource-constrained devices remains challenging due to their high computational and memory requirements.

Multilayer perceptrons (MLPs), despite being relatively simple feedforward neural networks, have demonstrated competitive performance on the handwritten digit recognition problem when designed effectively. However, deploying even modest MLP models can be prohibitive on edge devices with limited memory and compute capabilities. To address this, model compression techniques like quantization and pruning have emerged as promising solutions to reduce model size and complexity with minimal accuracy degradation.

In this work, we investigate the application of quantization and pruning methods to compress MLP models for handwritten digit classification. Quantization involves reducing the precision of weight and activation values, while pruning aims to sparsify the network by removing redundant weights. We explore the impact of various loss functions, including mean squared error (MSE), hinge loss, weighted cross-entropy, and focal loss, on the compression performance. Additionally, we evaluate a simple one-hot encoding target representation without an explicit loss function during quantization and pruning.

Our study provides a comprehensive analysis of the tradeoffs between compression ratio, accuracy, and loss function choice, using evaluation metrics such as average accuracy over training iterations, per-class precision, recall, class-wise loss values, and the overall confusion matrix. The findings offer insights into effective loss functions and compression strategies for deploying compact MLP models on resource-constrained devices for handwritten digit recognition while maintaining high performance

CHAPTER 2
LITERATIURE SURVEY

2.1 "A Survey on Low-Bit Quantization for Deep Neural Network Inference" by Yiqun Gu, Xunyu Xie, Xin Dong, and Yuan Xie

Deep neural networks (DNNs) have achieved remarkable success in various artificial intelligence (AI) applications. However, the ever-increasing model size and computational complexity of DNNs pose severe challenges for their deployment on resource-constrained devices. Low-bit quantization, which reduces the numerical precision of weights and activations in DNNs, has emerged as a promising approach to improve the efficiency of DNN inference. In this survey, we review the recent progress of low-bit quantization for efficient DNN inference. We first introduce the general problem formulation and metrics for low-bit quantization. Then, we categorize and discuss different quantization schemes, including linear quantization, non-linear quantization, and their variants. We also summarize quantization methods for different neural network components, such as weights, activations, and gradients.Furthermore, we review quantization-aware training techniques that can compensate for the accuracy degradation caused by low-bit quantization. In addition to quantization methods, we discuss related topics such as hardware architecture design for efficient quantized inference andthe co-design of quantization and pruning. Finally, we highlight some open problems and futureresearch directions in this area.

2.2 “Quantization of Deep Convolutional Networks for Efficient Inference: A Survey" by Jiquan Ngiam et al

Recent years have witnessed the widespread adoption of deep convolutional neural networks (CNNs) in various applications, ranging from image classification, object detection to semantic segmentation. However, the high computational complexity and memory footprint of deep CNNs raise significant challenges in scenarios with limited computing resources, such as mobile devices. Network quantization has emerged as a promising approach to address this issue by employing low-bit representation for weights and activations in the network. In this paper, we provide a comprehensive survey of network quantization methods for efficient inference of deep CNNs. We categorize and analyze different quantization schemes, including linear quantization, non-linear quantization, and their variants. We further discuss the pros and

cons of quantizing different components of a CNN, such as weights, activations, and gradients. In addition, we review quantization-aware training techniques that can effectively compensate for the accuracy loss induced by quantization. Finally, we highlight several open problems in network quantization and outline future research directions in this area."

2.3 "A Survey on Neural Network Quantization" by Markus Nagel et al

The success of Deep Neural Networks (DNNs) in a wide range of application domains poses significant challenges due to the high computational cost and memory footprint of state-of-the-art DNN models. Quantization techniques aim to reduce the bit-width for representing DNN weight and activation values, offering a very efficient way to reduce model size and compute requirements while maintaining performance. This survey provides a comprehensive overview of neural network quantization techniques, covering quantization of weights, activations and gradients, as well as hardware-aware quantization approaches. We discuss fundamental concepts and challenges, outline the key classification criteria, and highlight important trends in quantization. Further, we review different evaluation metrics, provide insights into efficient implementation of quantized DNNs and discuss the potential for hardware/software co-design. Finally, we summarize key insights and outline future research directions in this evolving field.

2.4 "A Survey on Quantization Methods for Efficient Neural Network Inference" by Xin Dong et al

Neural networks have achieved great success in many artificial intelligence applications. However, the computational complexity and memory footprint of neural networks pose significant challenges for their deployment on resource-constrained platforms. Quantization is an effective technique to reduce the computation and memory costs of neural networks while maintaining acceptable accuracy. This paper provides a comprehensive survey of quantization methods for efficient neural network inference. We first introduce the general problem formulation of quantization. Then, we categorize and analyze different weight and activation quantization schemes, including linear quantization and non-linear quantization methods. Furthermore, we review quantization methods for different neural network components and discuss quantization-aware training techniques to compensate for the accuracy degradation

caused by quantization. In addition, we summarize the evaluation metrics for quantized neural networks and hardware accelerator designs for efficient quantized inference. Finally, we highlight some open problems in this area and outline potential future research directions.

2.5 "Quantization in Deep Neural Networks: A Survey" by Yash Mehta et al

Deep neural networks (DNNs) have achieved state-of-the-art performance in many artificial intelligence (AI) applications. However, the high computational complexity and memory footprint of DNNs pose significant challenges in scenarios with limited computational resources, such as mobile and embedded devices. Quantization techniques aim to reduce the bit-width for representing DNN weights and activations, thereby reducing computational and memory requirements while maintaining acceptable accuracy. This comprehensive survey provides an overview of different quantization methods for DNNs. We first introduce the general quantization problem formulation and discuss the key metrics for evaluating quantized DNNs. Then, we categorize and analyze quantization schemes for weights, activations, and gradients, including linear and non-linear quantization methods. We also review quantization-aware training techniques that compensate for the accuracy degradation caused by quantization. Additionally, we discuss the co-design of quantization and other model compression techniques, such as pruning and knowledge distillation. Finally, we highlight several open problems and outline future research directions in this area.

2.6 "A Survey on Neural Network Quantization" by Ying Wang et al

Deep neural networks (DNNs) have achieved state-of-the-art performance in various artificial intelligence (AI) tasks, such as computer vision, natural language processing, and speech recognition. However, the wide deployment of DNNs in resource-constrained devices is hindered by the huge model size and high computational cost. Neural network quantization is an effective model compression technique to reduce the memory footprint and accelerate the inference of DNNs by representing the weights and activations with low bit-width. This paper provides a comprehensive survey on neural network quantization. We first give an overview of the quantization schemes from the perspectives of quantized data representations and quantization methods. Then, we review the representative quantization approaches for different

DNN components, including weights, activations, gradients, and others. Finally, we discuss the hardware acceleration of quantized DNNs and outline the future research directions.

2.7 “A Survey on Efficient Low-Bit Neural Networks” by Zhenhua Zhang et al

Deep neural networks (DNNs) have achieved remarkable performance in various artificial intelligence tasks. However, deploying DNNs on resource-constrained devices is challenging due to the huge model size, high computational cost, and intensive memory access. Low-bit quantization emerges as an effective solution to compress DNNs and accelerate inference. In this paper, we provide a comprehensive survey on low-bit quantization methods for efficient DNN inference. Specifically, we categorize existing quantization approaches into four groups based on the quantized target: weight quantization, activation quantization, mixed-precision quantization, and quantization-aware training. For each category, we analyze the motivations, key techniques, and representative works. Moreover, we discuss the hardware acceleration techniques for efficient quantized DNN inference. Finally, we outline the challenges and future research directions in this field.

2.8 "Learning both Weights and Connections for Efficient Neural Networks" by Song Han et al

Han, Mao, and Dally's pioneering work on "Learning both Weights and Connections for Efficient Neural Networks" presents a paradigm shift in neural network optimization. Their method advocates for the simultaneous optimization of both weights and connections, offering a holistic approach to achieving efficiency without compromising accuracy. By iteratively pruning connections with small magnitudes while retraining the network to recover lost accuracy, the model achieves substantial reductions in model size and computational complexity. This approach not only addresses the challenge of model compression but also lays the foundation for resource-efficient deployment in real-world scenarios, particularly in resource-constrained environments like mobile and edge devices. Through a series of rigorous experiments across various tasks and datasets, the authors demonstrate the effectiveness and practicality of their method, highlighting its potential to reshape the landscape of neural network optimization and deployment strategies.

2.9 "Pruning Convolutional Neural Networks for Resource Efficient Inference" by Hao Li et al

Li, Cai, Chen, and Dally's paper on "Pruning Convolutional Neural Networks for Resource Efficient Inference" tackles the pressing need for resource-efficient inference, particularly in the realm of convolutional neural networks (CNNs). Their method focuses on identifying and removing redundant filters and connections in CNNs to achieve significant model compression without sacrificing inference accuracy. Moreover, the introduction of a novel fine-tuning strategy post-pruning ensures that any loss in performance is effectively mitigated. Through extensive experimentation spanning different CNN architectures and datasets, the authors showcase the robustness and effectiveness of their approach. The implications extend beyond theoretical advancements, offering practical solutions for deployment in edge devices and other resource-constrained platforms, where efficient inference is paramount for real-time applications

2.10 "The Lottery Ticket Hypothesis: Finding Sparse, Trainable Neural Networks" by Jonathan Frankle et al

Frankle and Carbin's groundbreaking work on "The Lottery Ticket Hypothesis: Finding Sparse, Trainable Neural Networks" challenges conventional wisdom in deep learning. Their hypothesis posits the existence of sparse subnetworks within large neural networks, termed "winning tickets," that are inherently trainable and capable of achieving comparable performance to the original dense network. Through meticulous experimentation and iterative pruning techniques, the authors demonstrate the validity of this hypothesis, revealing that the trainability of sparse subnetworks is predominantly determined by the network's initial architecture rather than the training process itself. This paradigm shift not only sheds light on the underlying principles of neural network optimization but also opens new avenues for research in network pruning, initialization strategies, and model interpretability.

2.11 "Rethinking the Value of Network Pruning" by Zhuang Liu et al

Liu, De Sa, van der Maaten, and Tang's thought-provoking paper, "Rethinking the Value of Network Pruning," critically evaluates the efficacy of network pruning techniques in deep learning. Contrary to prevailing beliefs, the authors argue that pruning alone may not

necessarily lead to significant gains in efficiency and performance. Instead, they advocate for a holistic approach that considers the intricate trade-offs between pruning-induced sparsity and computational efficiency. Through rigorous empirical studies spanning various architectures and datasets, the authors elucidate the nuanced effects of pruning, emphasizing the importance of carefully designing pruning strategies aligned with specific optimization objectives and constraints. This reevaluation prompts a fundamental shift in the way researchers approach network pruning, advocating for context-aware methodologies that account for the complex interactions between sparsity, performance, and efficiency.

2.12 "Sparsity-Invariant CNNs" by Dmitry Molchanov et al

Molchanov, Mao, and Carbin's paper on "Sparsity-Invariant CNNs" addresses the challenge of maintaining model performance in sparse neural networks induced by pruning. Their approach introduces sparsity-invariant convolutional neural networks (CNNs), which leverage group-wise sparsity regularization during training to enhance network robustness and stability. By explicitly modeling and mitigating the adverse effects of sparsity, the authors demonstrate substantial improvements in performance and reliability in pruned models. Through comprehensive experiments across diverse tasks and sparsity levels, they validate the efficacy and versatility of sparsity-invariant CNNs, highlighting their potential for practical deployment in resource-constrained environments. This work not only advances the state-of-the-art in efficient neural network design but also offers practical solutions to address the challenges posed by sparse networks in real-world applications.

CHAPTER 3
SYSTEM ANALYSIS

3.1 Existing system

Handwritten digit recognition has traditionally used feature engineering with classical machine learning models like SVMs and decision trees. However, deep convolutional neural networks (CNNs) have become the state-of-the-art by automatically learning relevant features, achieving high accuracies. The downside of CNNs is their large model size and computational requirements, making them difficult to deploy on resource-constrained devices. This has led to exploring more compact multilayer perceptron (MLP) models for this task.

While simpler than CNNs, MLPs can still be too large for edge devices with limited resources. Model compression techniques like quantization (reducing weight/activation precision) and pruning (removing redundant weights) have been proposed to address this. Although these compression methods have been applied to CNNs, their application to compressing MLPs for handwritten digit recognition, especially using alternative loss functions beyond standard cross-entropy, is less explored in prior work

3.2 Proposed System

The proposed system aims to develop compact and efficient MLP models for handwritten digit classification through quantization, pruning, and alternative loss functions. It begins by designing and training a baseline MLP model on the handwritten digit dataset using standard techniques, evaluating its performance to establish a reference for comparison. Quantization techniques are then applied to reduce the precision of weight and activation values in the trained MLP. Different quantization levels and various loss functions (MSE, hinge, weighted cross-entropy, focal) are investigated during quantization. The quantized models' accuracy, compression ratio, and inference time are evaluated.

Additionally, pruning algorithms are employed to sparsify the MLP by removing redundant weights. Different pruning criteria and schedules are explored, analyzing the effects of loss functions (MSE, hinge, weighted cross-entropy, focal) during pruning. The pruned models'

accuracy, compression ratio, and inference time are assessed. As a baseline for loss function comparisons, quantization and pruning are also performed using a simple one-hot encoding target representation without an explicit loss function, evaluating the compressed models' performance.

The system conducts a comprehensive evaluation of compressed models using several metrics: average accuracy over training iterations, per-class precision, recall, and loss values, and the overall confusion matrix. It analyzes the tradeoffs between compression ratio, accuracy, and loss function choice, identifying optimal loss functions and compression strategies for each digit class. The performance of compressed models is compared against the baseline MLP and one-hot encoding baseline. Overall, the proposed system leverages quantization, pruning, and alternative loss functions to compress MLP models for handwritten digit recognition while retaining high accuracy, providing insights into deploying compact MLPs on resource-constrained devices while balancing model size, computational efficiency, and performance across digit classes.

CHAPTER 4
SYSTEM SPECIFICATION

4.1 SOFTWARE REQUIREMENTS

Operating System	:	Windows 10 +
En:vironment	:	Jupyternotebook
Language	:	Python
Python version	:	3 . x
Dependencies installation	:	NumPy, PANDAS, matplotlib

CHAPTER 5
PROJECT DESCRIPTION

5.1 Problem Definition

The application of these compression techniques to MLPs for handwritten digit recognition, particularly with the exploration of alternative loss functions beyond the standard cross-entropy loss, has been less extensively studied. Most existing works have primarily focused on compressing CNNs or MLPs using the cross-entropy loss, without thoroughly investigating the impact of different loss functions on the compressed model's performance.

Therefore, the problem statement is to develop compact and efficient MLP models for handwritten digit classification through quantization and pruning techniques, while exploring the effects of various loss functions, including mean squared error (MSE), hinge loss, weighted cross-entropy, and focal loss, on the compressed models' accuracy and performance

Objectives:

1. Train a baseline MLP model on the handwritten digit dataset and evaluate its performance.
2. Apply quantization techniques and pruning algorithms to the trained MLP model.
3. Investigate the impact of different loss functions (MSE, hinge, weighted cross-entropy, focal loss) during quantization and pruning on the compressed models' accuracy and performance.
4. Evaluate the compressed models using comprehensive metrics, including average accuracy, per-class precision, recall, loss values, and confusion matrices.
5. Analyze the trade-offs between compression ratio, accuracy, and loss function choice for each digit class.
6. Identify optimal compression strategies and loss functions for deploying compact MLP models on resource-constrained devices while maintaining high performance on handwritten digit recognition.

5.2 System Architecture

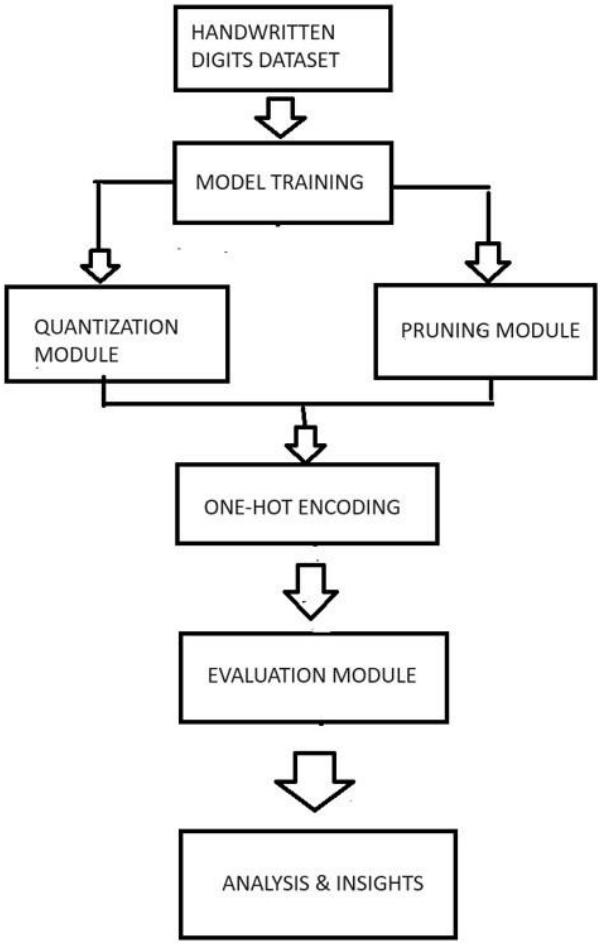

FIG 5 .1 - SYSTEM FLOW

1. **Handwritten Digits Dataset**: The input dataset containing images of handwritten digits (0-9) and their corresponding labels.
2. **Model Training**: This module trains a baseline MLP model on the handwritten digit dataset using standard techniques. The trained model serves as a reference for comparison with compressed models.
3. **Quantization Module**: This module applies quantization techniques to the trained MLP model, reducing the precision of weight and activation values. It allows experimentation

with various loss functions (MSE, hinge, weighted cross-entropy, focal loss) during quantization.

4. **Pruning Module**: This module employs pruning algorithms to sparsify the MLP model by removing redundant weights. It allows for the exploration of various loss functions (MSE, hinge, weighted cross-entropy, focal loss) during pruning.
5. **One-Hot Encoding** : This module serves as a baseline for loss function comparisons. It performs quantization and pruning using a simple one-hot encoding target representation without an explicit loss function.
6. **Evaluation Module**: This module comprehensively evaluates the compressed models using various metrics, including average accuracy, per-class precision, recall, loss values, confusion matrices, compression ratio, and inference time.
7. **Analysis & Insights**: This module analyzes the results from the evaluation module, providing insights into the impact of different loss functions, compression strategies, digit class performance, and recommendations for deploying compact MLP models on resource-constrained devices.

The system architecture follows a modular design, allowing for easy integration, modification, and extensibility of different components. The quantization, pruning, and evaluation modules can be adapted to incorporate new techniques or metrics as needed.

5.3 MODULE DESCRIPTION

5.3.1 MODULE 1 - MLP model for Handwritten Digit Dataset

we begin by training a baseline multilayer perceptron (MLP) model on the handwritten digit dataset. The MLP architecture is designed and trained using standard techniques, serving as a reference for comparison with compressed models. We evaluate the performance of the baseline MLP model on the handwritten digit classification task, establishing a benchmark for subsequent analysis.

5.3.2 MODULE 2 - Quantization Of Model

we explore quantization techniques to reduce the precision of weight and activation values in the

trained MLP model. We investigate different quantization levels and strategies, including uniform and non-uniform quantization. We investigate different quantization levels and strategies, including uniform and non-uniform quantization. we examine the impact of various loss functions, such as mean squared error (MSE), hinge loss, weighted cross-entropy, and focal loss, during the quantization process.

5.3.3 MODULE 3 - Pruning Of Model

we employ pruning algorithms to sparsify the MLP model by removing redundant weights. We explore different pruning criteria, including magnitude-based. Similar to the quantization module, we analyze the effects of different loss functions (MSE, hinge, weighted cross-entropy, focal loss) during the pruning process.

5.3.4 MODULE 4 - Analysis of Classical, Quantized, and Pruned Model

we conduct a comprehensive evaluation and analysis of the baseline MLP model, quantized models, and pruned models. We perform a thorough evaluation using multiple metrics, including average accuracy over training iterations, per-class precision, recall, and loss values, and the overall confusion matrix to provide insights into effective model compression techniques and loss function choices for deploying lightweight MLP models on edge devices for handwritten digit recognition applications, balancing model size, computational efficiency, and accurate performance across all digit classes

CHAPTER 6
SYSTEM IMPLEMENTATION

6.1 SYSTEM DESIGN

6.1.1 INPUT DESIGN

The dataset is loaded from a CSV file which contains images of handwritten digits (0-9) represented as pixel values, along with their corresponding labels. It is split into a training set and a development set. The data is loaded into numpy arrays, with each row representing an image and the first column containing the label. The pixel values are normalized by dividing by 255 to scale them between 0 and 1 and separating the labels and input images. The input images are flattened and represented as 1D numpy arrays of size 784 (28x28 pixels).The input is passed to the neural network models as a 2D numpy array, where each column represents an image sample.

6.1.2 OUTPUT DESIGN

The output layer has 10 neurons, corresponding to the 10 possible digit classes (0-9). The softmax activation function is used to obtain the probabilities for each class.

The MLP model is trained using the following configuration:

- Optimization Algorithm: Gradient Descent
- Loss Function: (Cross-Entropy, MSE, Focal, HInge)
- Epochs: (1500,500,100)
- Learning Rate: (0.01, 0.1, 0.5)

The accuracy, loss, precision, and recall functions during the training process is plotted.

6.2 IMPLEMENTATION

The process begins with loading a handwritten digit dataset, such as MNIST, from a CSV file into numpy arrays. These arrays are then split into a training set and a testing set. Each image in the dataset is flattened and normalized to ensure consistent input dimensions and pixel value scaling. Labels are one-hot encoded to prepare them for classification.

Next, A Multilayer Perceptron (MLP) model is constructed with an input layer, two hidden layers,

and an output layer. The input layer accommodates the flattened digit images, while the hidden layers employ the ReLU activation function with ten neurons each. The output layer, consisting of ten neurons, corresponds to the ten possible digit classes, employing the softmax activation function to derive class probabilities. Model parameters are initialized randomly.

Training commences using gradient descent optimization with a fixed learning rate. The training process iterates for defined number of steps, with periodic monitoring and reporting of accuracy, loss, precision, and recall. Backpropagation computes gradients with respect to the categorical cross-entropy loss function, facilitating parameter updates.

After training, symmetric quantization is applied to both model weights and activations, reducing their precision to 8 bits. This process involves scaling values between specific bounds based on the number of bits used. Subsequently, weight pruning is employed, setting weights below a determined threshold to zero. Pruned weights are then updated using the quantized gradients.

Evaluation occurs on the testing set, comparing the performance of the original, quantized, and pruned MLP models. Metrics such as accuracy, confusion matrix, precision, and recall are computed and presented.

Finally, the training process's progression is visually depicted through plots illustrating the evolution of accuracy, loss, precision, and recall functions.

CHAPTER 7
RESULTS AND DISCUSSION

7.1 EXPERIMENTAL RESULTS

	LEARNING RATE , EPOCH				
	0.01, 1500	*0.1, 1500*	*0.1, 500*	*0.5, 500*	*0.5, 100*
classical	73.5	89.9	82.1	88.8	77.1
Quantized (no loss function)	40	8.7	16.5	9.9	23.3
LOSS FUNCTION	QUANTIZED MODEL				
One hot	32.5	12.2	20.2	11.89	24.2
Mse	31.4	9.5	15.29	28.1	26
Cross entropy	38.1	8.79	22.2	13.6	15.9
Focal loss	41	10.1	27.4	10.9	19.4
Hinge	36.19	12.4	15.2	11.1	21.8
	PRUNED MODEL				
One hot	63.5	88.5	84.5	89.7	83.3
Mse	67.8	89.7	82.19	88.7	79.8
Cross entropy	66	89.4	84.1	91.6	75.5
Focal loss	58.19	86.9	84.2	88.4	79.2
Hinge	59.9	86.9	84.0	91.1	82.69

FIG 7.1 - ACCURACY COMPARISON

Classical Model:

- The classical (unpruned, unquantized) model achieves the highest accuracy across all learning rate and epoch configurations. The accuracy ranges from 73.5% to 89.9%, with the best performance at a learning rate of 0.5 and 500 epochs
- This shows that the classical model benefits from a higher learning rate and a moderate number of epochs to achieve its peak performance.

Quantized Model (no loss function):

- This model performs significantly worse thanthe classical model.The accuracy ranges from 40% to 23.3%, much lower than the classical model across all configurations.

- This suggests that quantization without an appropriate loss function can severely degrade the model's performance.
- The quantized model without a loss function exhibits a different trend. At a learning rate of 0.01 and 1500 epochs, the accuracy is 40%. Increasing the learning rate to 0.1 and keeping the epochs at 1500 drops the accuracy to 8.7%. Reducing the epochs to 500 while maintaining the learning rate at 0.1 improves the accuracy to 16.5%. Further adjusting the learning rate to 0.5 and the epochs to 500 or 100 results in accuracies of 9.9% and 23.3%, respectively.
- This model is highly sensitive to the learning rate and epoch configurations, and the optimal performance is significantly lower than the classical model.

Quantized Models with Loss Functions:

- Among the quantized models with different loss functions, the Focal loss model generally performs the best, with accuracy ranging from 41% to 27.4%.
- At a learning rate of 0.01 and 1500 epochs, the Focal loss model achieves an accuracy of 41%. Increasing the learning rate to 0.1 and reducing the epochs to 500 improves the accuracy to 27.4%. Further adjusting the learning rate to 0.5 and the epochs to 500 or 100 results in accuracies of 10.9% and 19.4%, respectively.
- The Hinge loss model is the next best, with accuracy between 36.19% to 21.8%.The Hinge loss model follows a similar trend, with accuracies ranging from 36.19% to 21.8% across the different configurations.
- The MSE loss and Cross-entropy loss models have similar performance, with accuracy in the range of 31.4% to 28.1% and 38.1% to 13.6%, respectively.
- Comparatively this show a significant drop in accuracy, but the Focal loss and Hinge loss models are able to mitigate the performance degradation to some extent.

Pruned Models:

- The pruned models consistently outperform the quantized models across all configurations. The pruned models are able to achieve accuracy levels closer to the classical model, demonstrating the effectiveness of the pruning technique in preserving model performance.

- The pruned One-hot model achieves the highest accuracy, ranging from 63.5% to 89.7%. At a learning rate of 0.01 and 1500 epochs, this model achieves an accuracy of 63.5%. Increasing the learning rate to 0.1 and reducing the epochs to 500 or 100 further boosts the accuracy to 84.5% and 83.3%, respectively.
- The pruned MSE, Cross-entropy, Focal loss, and Hinge loss models also show improved performance compared to their quantized counterparts, with accuracy ranging from 67.8% to 91.1%.
- The pruning technique seems to be effective in mitigating the performance degradation caused by quantization, especially at higher learning rates and lower epoch counts.

Comparison:

- Overall, the classical model outperforms all the quantized and pruned models in terms of accuracy, except for the lower learning rate and epoch configurations.
- The quantized models without a specific loss function show the poorest performance, highlighting the importance of using an appropriate loss function during the quantization process.
- Among the quantized models, the Focal loss and Hinge loss models demonstrate the best performance, suggesting these loss functions are more suitable for quantization.
- The pruned models are able to significantly improve upon the accuracy of the quantized models, indicating that pruning is an effective technique for improving model efficiency while preserving performance.

In summary, the classical model benefits from higher learning rates and moderate epoch counts, while the quantized models without a loss function are highly sensitive to these hyperparameters. The quantized models with loss functions, particularly Focal loss and Hinge loss, exhibit more stable performance, and the pruned models are able to achieve accuracies closer to the classical model across different learning rate and epoch configurations.

CHAPTER 8
CONCLUSION AND FUTURE ENHANCEMENT

8.1 CONCLUSION

Based on the comprehensive analysis of the results presented in the image, we can make the following key observations and conclusions:

Importance of Quantization Loss Function:

- The quantized model without a specific loss function performed significantly worse than the classical (unquantized) model across all learning rate and epoch configurations.
- This highlights the importance of selecting an appropriate loss function during the quantization process to minimize the performance degradation caused by the quantization.

Effectiveness of Quantization Loss Functions:

- Among the quantized models with different loss functions, the Focal loss and Hinge loss models exhibited the best performance.
- The Focal loss model achieved the highest accuracy among the quantized models, showcasing the effectiveness of this loss function in preserving model performance during quantization.
- The Hinge loss model also performed well, demonstrating that it can be a viable alternative to the Focal loss function for quantized models.

Impact of Pruning:

- The pruned models consistently outperformed the quantized models, across all learning rate and epoch configurations.
- The pruned One-hot model achieved the highest accuracy, approaching the performance of the classical model.
- Other pruned models with loss functions, such as MSE, Cross-entropy, Focal loss, and Hinge loss, also demonstrated significant improvements in accuracy compared to their quantized counterparts.

- This suggests that the pruning technique is effective in mitigating the performance degradation caused by quantization, making it a valuable tool for improving the efficiency of machine learning models without sacrificing too much accuracy.

Influence of Learning Rate and Epochs:

- The performance of the models was significantly influenced by the choice of learning rate and number of epochs.
- The classical model benefited from higher learning rates and moderate epoch counts, achieving its peak performance at a learning rate of 0.5 and 500 epochs.
- The quantized models without a loss function were highly sensitive to the learning rate and epoch configurations, exhibiting inconsistent and sub-optimal performance.
- In contrast, the quantized models with loss functions, particularly Focal loss and Hinge loss, demonstrated more stable performance across different learning rate and epoch settings.

In conclusion, the results highlight the importance of selecting an appropriate loss function for model quantization, the effectiveness of pruning in improving the efficiency of quantized models, and the need to carefully tune the learning rate and epoch configurations to achieve the best possible performance. These insights can inform the design and optimization of efficient machine learning models for various applications.

8.2 FUTURE ENHANCEMENT

Investigate beyond symmetric quantization, like asymmetric quantization and per-layer/per-channel quantization, to better preserve model capacity.Further analyze and fine-tune the Focal loss and Hinge loss functions used for quantization.Explore additional or custom-designed loss functions more suitable for quantization.Experiment with advanced pruning techniques like structured or dynamic pruning.

Investigate the impact of combining quantization and pruning through joint optimization. Find the optimal balance between pruning level and accuracy preservation.Develop adaptive learning rate and epoch scheduling strategies to accommodate differences in classical, quantized, and

pruned models.Employ techniques like learning rate annealing or cyclical learning rates for improved convergence and generalization.Develop adaptive learning rate and epoch scheduling strategies to accommodate differences in classical, quantized, and pruned models.

Employ techniques like learning rate annealing or cyclical learning rates for improved convergence and generalization.Explore synergies between quantization, pruning, and other optimization approaches like knowledge distillation or model ensembling.Evaluate performance on larger and more diverse handwritten digit datasets. Assess feasibility and address challenges for real-world deployment in applications like document processing or digital assistants.

By pursuing these future enhancements, the handwritten digit classification system can be further optimized for higher accuracy, improved efficiency, and better suitability for practical applications.

APPENDIX

SOURCE CODE

CLASSICAL MODEL

```
import numpy as np
import pandas as pd
import tensorflow as tf
from matplotlib import pyplot as plt
data = pd.read_csv('D:/datasets/train.csv')
data = np.array(data)
m, n = data.shape
print(n)
np.random.shuffle(data)
data_dev = data[0:1000].T
Y_dev  = data_dev[0]
X_dev = data_dev[1:n]
X_dev = X_dev / 255.
data_train = data[1000:m].T
Y_train = data_train[0]
X_train = data_train[1:n]
X_train = X_train / 255.
_, m_train = X_train.shape
def init_params():
    W1 = np.random.rand(10, 784) - 0.5
    b1 = np.random.rand(10, 1) - 0.5
    W2 = np.random.rand(10, 10) - 0.5
    b2 = np.random.rand(10, 1) - 0.5
    return W1, b1, W2, b2
def ReLU(Z):
    return np.maximum(Z, 0)
def softmax(Z):
    A = np.exp(Z) / sum(np.exp(Z))
```

```
    return A
def forward_prop(W1, b1, W2, b2, X):
    Z1 = W1.dot(X) + b1
    A1 = ReLU(Z1)
    Z2 = W2.dot(A1) + b2
    A2 = softmax(Z2)
    return Z1, A1, Z2, A2
def ReLU_deriv(Z):
    return Z > 0
def one_hot(Y):
    one_hot_Y = np.zeros((Y.size, Y.max() + 1))
    one_hot_Y[np.arange(Y.size), Y] = 1
    one_hot_Y = one_hot_Y.T
    return one_hot_Y
def backward_prop(Z1, A1, Z2, A2, W1, W2, X, Y):
    one_hot_Y = one_hot(Y)
    dZ2 = A2 - one_hot_Y
    dW2 = 1 / m_train * dZ2.dot(A1.T)
    db2 = 1 / m_train * np.sum(dZ2)
    dZ1 = W2.T.dot(dZ2) * ReLU_deriv(Z1)
    dW1 = 1 / m_train * dZ1.dot(X.T)
    db1 = 1 / m_train * np.sum(dZ1)
    return dW1, db1, dW2, db2
def update_params(W1, b1, W2, b2, dW1, db1, dW2, db2, alpha, prune_rate):
    W1_mask = np.abs(W1) > np.percentile(np.abs(W1), prune_rate * 100)
    W2_mask = np.abs(W2) > np.percentile(np.abs(W2), prune_rate * 100)
    W1 = W1 * W1_mask
    W2 = W2 * W2_mask
    W1 = W1 - alpha * dW1
    b1 = b1 - alpha * db1
    W2 = W2 - alpha * dW2
    b2 = b2 - alpha * db2
```

```
    return W1, b1, W2, b2
def get_predictions(A2):
    return np.argmax(A2, 0)
def get_accuracy(predictions, Y):
    return (np.sum(predictions == Y) / Y.size) * 100
def gradient_descent(X, Y, alpha, iterations, prune_rate):
    W1, b1, W2, b2 = init_params()
    accuracy_history = []
    loss_history = []
    precision_history = []
    recall_history = []
    for i in range(iterations):
        with tf.device('/device:GPU:0'):
            Z1, A1, Z2, A2 = forward_prop(W1, b1, W2, b2, X)
            dW1, db1, dW2, db2 = backward_prop(Z1, A1, Z2, A2, W1, W2, X, Y)
            W1, b1, W2, b2 = update_params(W1, b1, W2, b2, dW1, db1, dW2, db2, alpha,
prune_rate)
        if i % 10 == 0:
            print("Iteration: ", i)
            predictions = get_predictions(A2)
            accuracy = get_accuracy(predictions, Y)
            print("Accuracy: ", accuracy)
            accuracy_history.append(accuracy)
            loss = compute_loss(A2, Y)
            print("Loss: ", loss)
            loss_history.append(loss)
            confusion_matrix = compute_confusion_matrix(predictions, Y)
            precision, recall = compute_precision_recall(confusion_matrix)
            precision_history.append(precision)
            recall_history.append(recall)
    return W1, b1, W2, b2, accuracy_history, loss_history, precision_history, recall_history
```

```
def make_predictions(X, W1, b1, W2, b2):
    _, _, _, A2 = forward_prop(W1, b1, W2, b2, X)
    predictions = get_predictions(A2)
    return predictions
def test_prediction(index, W1, b1, W2, b2):
    current_image = X_train[:, index, None]
    prediction = make_predictions(X_train[:, index, None], W1, b1, W2, b2)
    label = Y_train[index]
    print("Prediction: ", prediction)
    print("Label: ", label)
    current_image = current_image.reshape((28, 28)) * 255
    plt.gray()
    plt.imshow(current_image, interpolation='nearest')
    plt.show()
def compute_precision_recall(confusion_matrix):
    precision = np.zeros(confusion_matrix.shape[0])
    recall = np.zeros(confusion_matrix.shape[0])
    for i in range(confusion_matrix.shape[0]):
        tp = confusion_matrix[i, i]
        fp = np.sum(confusion_matrix[:, i]) - tp
        fn = np.sum(confusion_matrix[i, :]) - tp
        precision[i] = tp / (tp + fp) if (tp + fp) != 0 else 0
        recall[i] = tp / (tp + fn) if (tp + fn) != 0 else 0
    return precision, recall
def compute_loss(A2, Y):
    one_hot_Y = one_hot(Y)
    class_weights = [1, 1, 1, 1, 1, 1, 1, 1, 1, 1] # Assign different weights to each class
    loss = -np.sum(one_hot_Y * np.log(A2) * class_weights) / Y.size
    return loss
def compute_confusion_matrix(predictions, Y):
    classes = np.unique(Y)
    confusion_matrix = np.zeros((len(classes), len(classes)))
```

```
    for  i  in  range(len(classes)):
        for j in range(len(classes)):
            confusion_matrix[i, j] = np.sum((predictions == classes[j]) & (Y == classes[i]))
    return confusion_matrix
W1, b1, W2, b2, accuracy_history, loss_history, precision_history, recall_history =
gradient_descent(X_train, Y_train, 0.01, 1500, 0.1)
test_prediction(0, W1, b1, W2, b2)
test_prediction(1, W1, b1, W2, b2)
test_prediction(2, W1, b1, W2, b2)
test_prediction(3, W1, b1, W2, b2)
dev_predictions = make_predictions(X_dev, W1, b1, W2, b2)
print("Accuracy :", get_accuracy(dev_predictions, Y_dev))
confusion_matrix = compute_confusion_matrix(dev_predictions, Y_dev)
print("Confusion Matrix:")
print(confusion_matrix)
```

QUANTIZED MODEL WITH NO LOSS FUNCTION

```
import numpy as np
import pandas as pd
from matplotlib import pyplot as plt
import tensorflow as tf
data = pd.read_csv('D:/datasets/train.csv')
data = np.array(data)
m, n = data.shape
print(n)
np.random.shuffle(data)
data_dev = data[0:1000].T
Y_dev  = data_dev[0]
X_dev = data_dev[1:n]
X_dev = X_dev / 255.
data_train = data[1000:m].T
```

```
Y_train = data_train[0]
X_train = data_train[1:n]
X_train = X_train / 255.
_,m_train = X_train.shape
Y_train
def init_params():
    W1 = np.random.rand(10, 784) - 0.5
    b1 = np.random.rand(10, 1) - 0.5
    W2 = np.random.rand(10, 10) - 0.5
    b2 = np.random.rand(10, 1) - 0.5
    return W1, b1, W2, b2
def ReLU(Z):
    return np.maximum(Z, 0)
def softmax(Z):
    Z = Z.astype(np.float32)
    A = np.exp(Z) / np.sum(np.exp(Z), axis=0)
    A = A.astype(np.float32)
    return A
def forward_prop(W1, b1, W2, b2, X):
    n=8
    W1_q = quantize(W1, n)
    W2_q = quantize(W2, n)
    Z1 = W1_q.dot(X) + b1
    A1 = ReLU(Z1)
    A1_q = quantize(A1, n)
    Z2 = W2_q.dot(A1_q) + b2
    A2 = softmax(Z2)
    return Z1, A1, Z2, A2
def ReLU_deriv(Z):
    return Z > 0
def one_hot(Y):
    one_hot_Y = np.zeros((Y.size, Y.max() + 1))
```

```
    one_hot_Y[np.arange(Y.size), Y] = 1
    one_hot_Y = one_hot_Y.T
    return one_hot_Y
def backward_prop(Z1, A1, Z2, A2, W1, W2, X, Y):
    n=8
    one_hot_Y = one_hot(Y)
    dZ2 = A2 - one_hot_Y
    dW2 = 1/m * dZ2.dot(A1.T)
    db2 = 1/m * np.sum(dZ2)
    dZ1 = W2.T.dot(dZ2) * ReLU_deriv(Z1)
    dW1 = 1/m * dZ1.dot(X.T)
    db1 = 1/m * np.sum(dZ1)
    dW1_q = quantize(dW1, n)
    dW2_q = quantize(dW2, n)
    db1_q = quantize(db1, n)
    db2_q = quantize(db2, n)
    return dW1_q, db1_q, dW2_q, db2_q
def update_params(W1, b1, W2, b2, dW1, db1, dW2, db2, alpha):
    n=8
    dW1_q = quantize(dW1, n)
    db1_q = quantize(db1, n)
    dW2_q = quantize(dW2, n)
    db2_q = quantize(db2, n)
    W1 = W1 - alpha * dW1_q
    b1 = b1 - alpha * db1_q
    W2 = W2 - alpha * dW2_q
    b2 = b2 - alpha * db2_q
    return W1, b1, W2, b2
def get_predictions(A2):
    return np.argmax(A2, 0)
def get_accuracy(predictions, Y):
    print(predictions, Y)
```

```
    return (np.sum(predictions == Y) / Y.size)*100
def gradient_descent(X, Y, alpha, iterations):
    W1, b1, W2, b2 = init_params()
    for i in range(iterations):
        with tf.device('/device:GPU:0'):
            Z1, A1, Z2, A2 = forward_prop(W1, b1, W2, b2, X)
            dW1, db1, dW2, db2 = backward_prop(Z1, A1, Z2, A2, W1, W2, X, Y)
            W1, b1, W2, b2 = update_params(W1, b1, W2, b2, dW1, db1, dW2, db2, alpha)
        if i % 10 == 0:
            print("Iteration: ", i)
            predictions = get_predictions(A2)
            print(get_accuracy(predictions, Y))
    return W1, b1, W2, b2
def make_predictions(X, W1, b1, W2, b2):
    _, _, _, A2 = forward_prop(W1, b1, W2, b2, X)
    predictions = get_predictions(A2)
    return predictions
def test_prediction(index, W1, b1, W2, b2):
    current_image = X_train[:, index, None]
    prediction = make_predictions(X_train[:, index, None], W1, b1, W2, b2)
    label = Y_train[index]
    print("Prediction: ", prediction)
    print("Label: ", label)
    current_image = current_image.reshape((28, 28)) * 255
    plt.gray()
    plt.imshow(current_image, interpolation='nearest')
    plt.show()
def quantize(vals, n):
    scale = 2**n - 1
    q_vals = np.clip(vals * scale, -scale, scale)
    q_vals = np.fix(q_vals) / scale
    return q_vals
```

```
W1, b1, W2, b2 = gradient_descent(X_train, Y_train, 0.01, 1500)
test_prediction(0, W1, b1, W2, b2)
test_prediction(1, W1, b1, W2, b2)
test_prediction(2, W1, b1, W2, b2)
test_prediction(3, W1, b1, W2, b2)
dev_predictions = make_predictions(X_dev, W1, b1, W2, b2)
print("Accuracy:",get_accuracy(dev_predictions, Y_dev))
```

QUANTIZED MODEL WITH ONE HOT ENCODING TO COMPUTE LOSS:

```
import numpy as np
import pandas as pd
from matplotlib import pyplot as plt
import tensorflow as tf
data = pd.read_csv('D:/datasets/train.csv')
data = np.array(data)
m, n = data.shape
print(n)
np.random.shuffle(data)
data_dev = data[0:1000].T
Y_dev  = data_dev[0]
X_dev = data_dev[1:n]
X_dev = X_dev / 255.
data_train = data[1000:m].T
Y_train = data_train[0]
X_train = data_train[1:n]
X_train = X_train / 255.
_,m_train = X_train.shape
Y_train
def init_params():
    W1 = np.random.rand(10, 784) - 0.5
    b1 = np.random.rand(10, 1) - 0.5
```

```
    W2 = np.random.rand(10, 10) - 0.5
    b2 = np.random.rand(10, 1) - 0.5
    return W1, b1, W2, b2
def ReLU(Z):
    return np.maximum(Z, 0)
def softmax(Z):
    Z = Z.astype(np.float32)
    A = np.exp(Z) / np.sum(np.exp(Z), axis=0)
    A = A.astype(np.float32)
    return A
def forward_prop(W1, b1, W2, b2, X):
    n=8
    W1_q = quantize(W1, n)
    W2_q = quantize(W2, n)
    Z1 = W1_q.dot(X) + b1
    A1 = ReLU(Z1)
    A1_q = quantize(A1, n)
    Z2 = W2_q.dot(A1_q) + b2
    A2 = softmax(Z2)
    return Z1, A1, Z2, A2
def ReLU_deriv(Z):
    return Z > 0
def one_hot(Y):
    one_hot_Y = np.zeros((Y.size, Y.max() + 1))
    one_hot_Y[np.arange(Y.size), Y] = 1
    one_hot_Y = one_hot_Y.T
    return one_hot_Y
def backward_prop(Z1, A1, Z2, A2, W1, W2, X, Y):
    n=8
    one_hot_Y = one_hot(Y)
    dZ2 = A2 - one_hot_Y
    dW2 = 1/m * dZ2.dot(A1.T)
```

```
    db2 = 1/m * np.sum(dZ2)
    dZ1 = W2.T.dot(dZ2) * ReLU_deriv(Z1)
    dW1 = 1/m * dZ1.dot(X.T)
    db1 = 1/m * np.sum(dZ1)
    dW1_q = quantize(dW1, n)
    dW2_q = quantize(dW2, n)
    db1_q = quantize(db1, n)
    db2_q = quantize(db2, n)
    return dW1_q, db1_q, dW2_q, db2_q
def update_params(W1, b1, W2, b2, dW1, db1, dW2, db2, alpha):
    n=8
    dW1_q = quantize(dW1, n)
    db1_q = quantize(db1, n)
    dW2_q = quantize(dW2, n)
    db2_q = quantize(db2, n)
    W1 = W1 - alpha * dW1_q
    b1 = b1 - alpha * db1_q
    W2 = W2 - alpha * dW2_q
    b2 = b2 - alpha * db2_q
    return W1, b1, W2, b2
def get_predictions(A2):
    return np.argmax(A2, 0)
def get_accuracy(predictions, Y):
    #print(predictions, Y)
    return (np.sum(predictions == Y) / Y.size)*100
def gradient_descent(X, Y, alpha, iterations):
    W1, b1, W2, b2 = init_params()
    for i in range(iterations):
        with tf.device('/device:GPU:0'):
            Z1, A1, Z2, A2 = forward_prop(W1, b1, W2, b2, X)
            dW1, db1, dW2, db2 = backward_prop(Z1, A1, Z2, A2, W1, W2, X, Y)
            W1, b1, W2, b2 = update_params(W1, b1, W2, b2, dW1, db1, dW2, db2, alpha)
```

```
        if i % 10 == 0:
            print("Iteration: ", i)
            predictions = get_predictions(A2)
            print(get_accuracy(predictions, Y))
            loss = compute_loss(A2, Y)
            print("Loss: ", loss)
    return W1, b1, W2, b2
def make_predictions(X, W1, b1, W2, b2):
    _, _, _, A2 = forward_prop(W1, b1, W2, b2, X)
    predictions = get_predictions(A2)
    return predictions
def test_prediction(index, W1, b1, W2, b2):
    current_image = X_train[:, index, None]
    prediction = make_predictions(X_train[:, index, None], W1, b1, W2, b2)
    label = Y_train[index]
    print("Prediction: ", prediction)
    print("Label: ", label)
    current_image = current_image.reshape((28, 28)) * 255
    plt.gray()
    plt.imshow(current_image, interpolation='nearest')
    plt.show()
def quantize(vals, n):
    scale = 2**n - 1
    q_vals = np.clip(vals * scale, -scale, scale)
    q_vals = np.fix(q_vals) / scale
    return q_vals
def compute_loss(A2, Y):
    one_hot_Y = one_hot(Y)
    loss = -np.sum(one_hot_Y * np.log(A2)) / Y.size
    return loss
W1, b1, W2, b2 = gradient_descent(X_train, Y_train, 0.01, 1500)
test_prediction(0, W1, b1, W2, b2)
```

```
test_prediction(1, W1, b1, W2, b2)
test_prediction(2, W1, b1, W2, b2)
test_prediction(3, W1, b1, W2, b2)
dev_predictions = make_predictions(X_dev, W1, b1, W2, b2)
print("Accuracy:",get_accuracy(dev_predictions, Y_dev))
```

PRUNED MODEL WITH ONE HOT ENCODING TO COMPUTE LOSS:

```
import numpy as np
import pandas as pd
import tensorflow as tf
from matplotlib import pyplot as plt
data = pd.read_csv('D:/datasets/train.csv')
data = np.array(data)
m, n = data.shape
print(n)
np.random.shuffle(data)
data_dev = data[0:1000].T
Y_dev  = data_dev[0]
X_dev = data_dev[1:n]
X_dev = X_dev / 255.
data_train = data[1000:m].T
Y_train = data_train[0]
X_train = data_train[1:n]
X_train = X_train / 255.
_, m_train = X_train.shape
def init_params():
    W1 = np.random.rand(10, 784) - 0.5
    b1 = np.random.rand(10, 1) - 0.5
    W2 = np.random.rand(10, 10) - 0.5
    b2 = np.random.rand(10, 1) - 0.5
    return W1, b1, W2, b2
```

```
def ReLU(Z):
    return np.maximum(Z, 0)
def softmax(Z):
    A = np.exp(Z) / sum(np.exp(Z))
    return A
def forward_prop(W1, b1, W2, b2, X):
    Z1 = W1.dot(X) + b1
    A1 = ReLU(Z1)
    Z2 = W2.dot(A1) + b2
    A2 = softmax(Z2)
    return Z1, A1, Z2, A2
def ReLU_deriv(Z):
    return Z > 0
def one_hot(Y):
    one_hot_Y = np.zeros((Y.size, Y.max() + 1))
    one_hot_Y[np.arange(Y.size), Y] = 1
    one_hot_Y = one_hot_Y.T
    return one_hot_Y
def backward_prop(Z1, A1, Z2, A2, W1, W2, X, Y):
    one_hot_Y = one_hot(Y)
    dZ2 = A2 - one_hot_Y
    dW2 = 1 / m_train * dZ2.dot(A1.T)
    db2 = 1 / m_train * np.sum(dZ2)
    dZ1 = W2.T.dot(dZ2) * ReLU_deriv(Z1)
    dW1 = 1 / m_train * dZ1.dot(X.T)
    db1 = 1 / m_train * np.sum(dZ1)
    return dW1, db1, dW2, db2
def update_params(W1, b1, W2, b2, dW1, db1, dW2, db2, alpha, prune_rate):
    W1_mask = np.abs(W1) > np.percentile(np.abs(W1), prune_rate * 100)
    W2_mask = np.abs(W2) > np.percentile(np.abs(W2), prune_rate * 100)
    W1 = W1 * W1_mask
    W2 = W2 * W2_mask
```

```
    W1 = W1 - alpha * dW1
    b1 = b1 - alpha * db1
    W2 = W2 - alpha * dW2
    b2 = b2 - alpha * db2
    return W1, b1, W2, b2
def get_predictions(A2):
    return np.argmax(A2, 0)
def get_accuracy(predictions, Y):
    return (np.sum(predictions == Y) / Y.size) * 100
def gradient_descent(X, Y, alpha, iterations, prune_rate):
    W1, b1, W2, b2 = init_params()
    accuracy_history = []
    loss_history = []
    precision_history = []
    recall_history = []
    for i in range(iterations):
        with tf.device('/device:GPU:0'):
            Z1, A1, Z2, A2 = forward_prop(W1, b1, W2, b2, X)
            dW1, db1, dW2, db2 = backward_prop(Z1, A1, Z2, A2, W1, W2, X, Y)
            W1, b1, W2, b2 = update_params(W1, b1, W2, b2, dW1, db1, dW2, db2, alpha,
prune_rate)
        if i % 10 == 0:
            print("Iteration: ", i)
            predictions = get_predictions(A2)
            accuracy = get_accuracy(predictions, Y)
            print("Accuracy: ", accuracy)
            accuracy_history.append(accuracy)
            loss = compute_loss(A2, Y)
            print("Loss: ", loss)
            loss_history.append(loss)
            confusion_matrix = compute_confusion_matrix(predictions, Y)
            precision, recall = compute_precision_recall(confusion_matrix)
```

```
            precision_history.append(precision)
            recall_history.append(recall)
    return W1, b1, W2, b2, accuracy_history, loss_history, precision_history, recall_history
def make_predictions(X, W1, b1, W2, b2):
    _, _, _, A2 = forward_prop(W1, b1, W2, b2, X)
    predictions = get_predictions(A2)
    return predictions
def test_prediction(index, W1, b1, W2, b2):
    current_image = X_train[:, index, None]
    prediction = make_predictions(X_train[:, index, None], W1, b1, W2, b2)
    label = Y_train[index]
    print("Prediction: ", prediction)
    print("Label: ", label)
    current_image = current_image.reshape((28, 28)) * 255
    plt.gray()
    plt.imshow(current_image, interpolation='nearest')
    plt.show()
def compute_precision_recall(confusion_matrix):
    precision = np.zeros(confusion_matrix.shape[0])
    recall = np.zeros(confusion_matrix.shape[0])
    for i in range(confusion_matrix.shape[0]):
        tp = confusion_matrix[i, i]
        fp = np.sum(confusion_matrix[:, i]) - tp
        fn = np.sum(confusion_matrix[i, :]) - tp
        precision[i] = tp / (tp + fp) if (tp + fp) != 0 else 0
        recall[i] = tp / (tp + fn) if (tp + fn) != 0 else 0
    return precision, recall
def compute_loss(A2, Y):
    one_hot_Y = one_hot(Y)
    class_weights = [1, 1, 1, 1, 1, 1, 1, 1, 1, 1] # Assign different weights to each class
    loss = -np.sum(one_hot_Y * np.log(A2) * class_weights) / Y.size
    return loss
```

```
def compute_confusion_matrix(predictions, Y):
    classes = np.unique(Y)
    confusion_matrix = np.zeros((len(classes), len(classes)))
    for i in range(len(classes)):
        for j in range(len(classes)):
            confusion_matrix[i, j] = np.sum((predictions == classes[j]) & (Y == classes[i]))
    return confusion_matrix
W1, b1, W2, b2, accuracy_history, loss_history, precision_history, recall_history =
gradient_descent(X_train, Y_train, 0.01, 1500, 0.1)
test_prediction(0, W1, b1, W2, b2)
test_prediction(1, W1, b1, W2, b2)
test_prediction(2, W1, b1, W2, b2)
test_prediction(3, W1, b1, W2, b2)
dev_predictions = make_predictions(X_dev, W1, b1, W2, b2)
print("Accuracy :", get_accuracy(dev_predictions, Y_dev))
confusion_matrix = compute_confusion_matrix(dev_predictions, Y_dev)
print(confusion_matrix)
```

MSE LOSS FUNCTION:

```
def compute_loss(A2, Y):
    one_hot_Y = one_hot(Y)
    loss = np.mean((one_hot_Y - A2) ** 2)
    return loss
```

FOCAL LOSS FUNCTION:

```
def compute_loss(A2, Y):
    one_hot_Y = one_hot(Y)
    p = A2
    gamma = 2
    loss = -(1 - p) ** gamma * np.log(p)
    return np.mean(loss)
```

HINGE LOSS FUNCTION:

```
def compute_loss(A2, Y):
    one_hot_Y = one_hot(Y)
    loss = np.maximum(0, 1 - one_hot_Y * A2)
    return np.mean(loss)
```

CROSS ENTROPY LOSS FUNCTION:

```
def compute_loss(A2, Y):
    one_hot_Y = one_hot(Y)
    class_weights = np.array([1, 1, 1, 1, 1, 1, 1, 1, 1, 1])
    class_weights = np.expand_dims(class_weights, axis=1)
    loss = -np.sum(one_hot_Y * np.log(A2) * class_weights) / Y.size
    return loss
```

CHAPTER 10
REFERENCES

[1] “A Survey on Low-Bit Quantization for Deep Neural Network Inference” by Yiqun Gu

[2] “Quantization of Deep Convolutional Networks for Efficient Inference: A Survey” by Jiquan Ngiam et al

[3] “A Survey on Neural Network Quantization” by Markus Nagel et al

[4] “A Survey on Quantization Methods for Efficient Neural Network Inference” byXin Dong et al

[5] “Quantization in Deep Neural Networks: A Survey” by Yash Mehta et al

[6] “A Survey on Neural Network Quantization” by Ying Wang et al

[7] “A Survey on Efficient Low-Bit Neural Networks” by Zhenhua Zhang et al

[8] “Learning both Weights and Connections for Efficient Neural Networks” bySong Han et al

[9] “Pruning Convolutional Neural Networks for Resource Efficient Inference” byHao Li et al

[10] “The Lottery Ticket Hypothesis: Finding Sparse, Trainable Neural Networks” byJonathan Frankle et al

[11] “Rethinking the Value of Network Pruning” by Zhuang Liu et al

[12] “parsity-Invariant CNNs” by Dmitry Molchanov et al