MayNet: A Neural Network Ensemble Approach Based on May's Theorem for Improved Classification

Jincheng Zhang [1,*], Jindong Zhang [2]
[1] Faculty of Science and Technology, Rajabhat Maha Sarakham University, Maha Sarakham 44000, Thailand
[2] Independent Researcher, Putian, Fujian 351144, China
Email: [1] zjc1639834588@gmail.com
*Corresponding Author

***Abstract*—In this study, we explored the possibility of applying May's Theorem to neural networks and proposed a new unified network architecture called MayNet. MayNet achieves category prediction by integrating multiple neural network "voters" and uses majority voting to determine the final classification result. Experimental results show that MayNet outperforms traditional single neural networks on CIFAR-10 and MedMNIST datasets and has high robustness. The paper compares the performance of MayNet with popular convolutional neural networks (such as ResNet18) on various datasets and demonstrates its superior performance. May's Theorem provides a solid theoretical foundation for the majority voting mechanism in neural network ensembles, ensuring improved decision accuracy through collective judgments of independent voters. MayNet's architecture innovatively integrates multiple independently trained convolutional neural networks as voters, leveraging majority voting to combine their outputs effectively. This design enhances classification accuracy, robustness, and generalization ability.**

***Keywords*—Neural Network Ensemble, May's Theorem, Classification, Deep Learning**

I. INTRODUCTION

Neural network ensemble technology is widely used in various tasks. Especially when facing complex problems, ensemble methods can combine the advantages of multiple models and significantly improve the final performance [1]-[6]. Ensemble learning usually achieves better performance than a single model by aggregating features from multiple weak classifiers [7]-[11]. Traditional single neural networks often have problems such as overfitting and local optimality. Ensemble methods reduce the impact of these problems and improve the stability and accuracy of predictions through the synergy of multiple models [12]-[18]. Especially in scenarios with large data volumes and complex functions, ensemble methods can better understand the underlying patterns in the data [19]-[25].

In ensemble methods, voting mechanism is a common decision-making method. Multiple models make decisions independently and the final output is determined by majority vote. May's Theorem provides theoretical support for this method, especially in the framework of multiple decision makers, helping us effectively understand how to make more rational decisions through voting mechanisms. The theorem states that multiple independent decision makers can make more accurate final decisions through collective majority opinions. This principle can be successfully applied to ensemble learning of neural networks. By integrating multiple neural network models, classification performance can be effectively improved and the error of a single model can be reduced.

In this paper, we propose a novel integrated network MayNet, which integrates multiple neural network models and implements a majority voting classification decision mechanism based on the idea of May's theorem. MayNet can effectively improve the accuracy of classification tasks and enhance the robustness of the model. In addition, we study the architectural design of MayNet and its performance on various datasets, and demonstrate its potential in unified neural networks.

May's theorem strictly defines the conditions under which majority voting can produce optimal decisions, such as voter independence and symmetry. Unlike existing ensemble methods that often lack theoretical foundations or rely on homogeneous models, MayNet introduces a theoretically sound framework inspired by social choice theory and encourages model diversity to enhance decision independence. This makes it particularly suitable for real-world noisy environments such as medical image classification.

II. MAY'S THEOREM AND ITS APPLICATIONS

A. Introduction to May's Theorem

May's Theorem is a mathematical theorem about collective decision-making, which provides theoretical support for the majority voting framework in particular [26]-[30]. It mainly discusses how to make the best choice through the collective decision-making of multiple independent decision makers. According to the theorem, when multiple independent decision makers make choices based on their own judgments on a certain issue, majority voting can effectively reduce the bias of individual decisions and ultimately produce results that reflect the intentions of the group.

The core idea of May's theorem is that in a decision-making process, the independence between individuals and a certain degree of decision balance ensure that the decision of the majority is ultimately correct [31]-[33]. In other words, when individual decision makers make decisions relatively independently, their choices are based on the same information, and they have similar standards, the results of

majority voting tend to accurately reflect the intentions of the group. This idea not only provides a basis for social choice theory, but also provides strong support for ensemble learning methods in machine learning.

B. Application to Neural Networks

In the field of neural networks, MayNet introduces the concept of May's Theorem into ensemble learning, maximizing the synergy between multiple neural network "voters" to improve model performance. MayNet uses multiple independently trained neural networks as voters. Each neural network independently predicts input data and outputs its own classification decision. All network decisions are determined by the majority voting mechanism to determine the final classification result.

This mechanism makes MayNet perform particularly well when processing complex data. By integrating the judgments of multiple networks, MayNet can effectively reduce overfitting and improve the generalization ability of the model. Each voter is an independent neural network, and their training processes are independent of each other, avoiding the training traps that may occur in a single network. By aggregating the learning results of multiple neural networks, MayNet can maximize the advantages of each model and provide more stable prediction results.

Compared with traditional single neural network models, MayNet has significant advantages in classification accuracy, robustness, etc. Especially when dealing with diverse and complex data sets, MayNet can significantly improve the accuracy and stability of the model through ensemble learning. Therefore, MayNet provides an innovative way to fuse neural networks, which can effectively improve the performance of neural networks in various practical applications.

III. MAYNET ARCHITECTURE

MayNet is an integrated neural network architecture that consists of multiple independent neural networks, each of which participates in the final decision as a "voter". These voters are similar in structure, each is a standard convolutional neural network (CNN), but they generate unique outputs through different initializations or paths during training. This design makes the predictions of each voter independent, and the integrated results can improve the final classification performance by integrating the opinions of multiple voters. MayNet's integration method is based on the majority voting principle, in which each voter makes his own judgment on the classification task, and the final classification result is determined by the collective majority opinion. In this way, MayNet can effectively improve classification accuracy and reduce the risk of overfitting or misjudgment caused by a single model.

The overall architecture of MayNet includes multiple voters, each of which is an independent convolutional neural network. Voters share the same network structure and hyperparameter configuration, but they produce different classification results through different initializations or training paths during training. This design can effectively simulate diverse decision-making mechanisms and further enhance the robustness of the model by integrating the results of multiple independent models. The output of each voter is combined and a majority voting mechanism is used to determine the final prediction category. The goal of MayNet is to improve the performance of the entire system in various practical tasks by integrating the advantages of multiple networks.

A. Voter

Each MayVoter is a standard convolutional neural network (CNN), whose structure includes multiple convolutional layers, pooling layers, and fully connected layers. Each voter will perform feature extraction when inputting data, and output the final classification result after several layers of convolution, pooling, activation, etc. The structure of each voter is as follows:

- Input layer: Receives input data, usually three-channel (RGB) image data. The image data is input into the network after preprocessing.
- Convolutional layer: Contains two convolutional layers, each with a convolution kernel size of 3x3, which is used to extract low-level features from the image. Each convolutional layer is followed by a ReLU activation function to introduce nonlinear factors and enhance the model's expressiveness.
- Pooling layer: After each convolutional layer, there is a maximum pooling layer. Pooling is used to reduce the size of feature maps, retain the main features, and reduce computational complexity.
- Fully connected layer: The output of the convolution and pooling layers is flattened and input to the fully connected layer for final classification. The fully connected layer is responsible for synthesizing the extracted feature information and finally outputting the category prediction.

MayVoter is designed to capture the local features of the image through convolution operations and make the final classification judgment through the fully connected layer. Since the structure of each voter is fixed, all voters are consistent in structure and hyperparameters, but due to different training paths, they will show diversity in decision making. This design allows MayNet to use the independent judgments of multiple voters to improve the final classification accuracy.

B. MayNet

MayNet is a neural network model composed of multiple MayVoters, each of which makes predictions independently and generates classification results. MayNet stacks the prediction results of all voters and finally determines the final prediction category through a majority voting mechanism. Specifically, each MayVoter makes a classification decision on the input data, and MayNet aggregates these decision results to form a voting pool. The prediction results of each voter will have an impact on the final decision, and the final classification result is determined by the majority voting results, that is, the category with the most occurrences is the final output. This ensemble method based on majority voting can effectively reduce the risk caused by the bias of a single model, especially when facing complex data sets, the ensemble results are often more accurate than the predictions of any single model. In practical applications, the classification accuracy of MayNet is often better than that of a single network, because it combines the learning results of multiple networks, can effectively avoid overfitting, and improve the robustness of the model.

C. Voting Process

The voting process of MayNet is its core component. On each input sample, MayNet calculates the prediction results of each voter. Then, by counting the prediction results of all voters, the category with the most occurrences is selected as the final classification result. The specific implementation process is as follows:

Each voter independently predicts the input sample and outputs its classification result (usually a probability distribution).

The prediction results of all voters are collected together and counted according to category.

For each category, count its frequency of occurrence among all voters, and select the category with the most occurrences as the final prediction category.

The final prediction result is output by MayNet as a classification decision for the input data. The specific code implementation is as follows:

```
class MayNet(nn.Module):
def __init__(self, num_voters=5):
super(MayNet, self).__init__()
# Creating multiple MayVoter voters
self.voters = nn.ModuleList([MayVoter() for _ in range(num_voters)])
def forward(self, x):
# Get the prediction results for each voter
logits_list = [voter(x) for voter in self.voters]
# Get the predicted category for each voter
preds = torch.stack([logit.argmax(dim=1) for logit in logits_list], dim=0)
final_preds = []
# For each sample, vote
for i in range(preds.size(1)):
votes = preds[:, i].tolist()
# Select the category with the most votes
most_common = Counter(votes).most_common(1)[0][0]
final_preds.append(most_common)
final_preds = torch.tensor(final_preds).to(x.device)
return final_preds
```

The above code shows how MayNet performs classification by integrating multiple voters. Each voter makes a prediction for the input data, and the final classification decision is made by majority voting. This method can effectively improve the accuracy of classification and enhance the model's adaptability to various input data.

The complete pseudo code of MayNet is as follows:

```
Initialize:
  Set random seed for reproducibility
  Set device to GPU if available, otherwise CPU
Data Preprocessing:
  For CIFAR-10:
    - Load training and test datasets
    - Apply normalization (mean = 0.5, std = 0.5)
    - Use only the first 1000 training samples
  For MedMNIST (PathMNIST):
    - Load training and test datasets with normalization
    - Use only the first 1000 training samples
Define MayVoter Module:
  A single CNN classifier consisting of:
    - Conv2D → ReLU → MaxPool
    - Conv2D → ReLU → MaxPool
    - Flatten
    - Fully Connected → ReLU → Fully Connected → Output Layer
Define MayNet Module:
  - Initialize with N voters (default: 5)
  - Each voter is a separate instance of MayVoter
MayNet Forward Pass:
  - For each input image batch:
    - Obtain prediction logits from all voters
    - Convert logits to predicted class labels (argmax)
    - Stack predictions from all voters
    - For each image in the batch:
      - Use majority voting (most common label among voters)
    - Return final predicted labels
Train MayNet:
  For E epochs (default: 5):
    For each batch of images and labels:
      For each voter in MayNet:
        - Zero gradients
        - Compute output logits
        - Compute cross-entropy loss with true labels
        - Backpropagate loss
        - Update weights using Adam optimizer
Evaluate MayNet:
  - Set model to evaluation mode
  - For each batch of test images:
    - Predict using MayNet
    - Compare predictions with ground truth labels
    - Collect predictions and labels for computing metrics
  - Compute:
    - Accuracy
    - Macro-averaged Recall
    - Macro-averaged F1 Score
  - Return metrics
Train and Evaluate ResNet18 (Baseline):
  - Initialize standard ResNet18 with adjusted output classes
  - Train for E epochs using same training loop (no ensemble)
  - Evaluate using same accuracy, recall, and F1 metrics
Main Experiment Loop:
  Repeat 10 times:
    - Train and evaluate MayNet
    - Record accuracy, recall, F1, and training time
    - Train and evaluate ResNet18
    - Record accuracy, recall, F1, and training time
  - Compute mean and standard deviation for all metrics
  - Print final averaged results with standard deviations
```

IV. Experimental Methods and Datasets

To evaluate the performance of MayNet, we used two popular datasets: CIFAR-10 and MedMNIST. CIFAR-10 contains 10 categories of color images, while MedMNIST contains 9 categories of medical images. We used MayNet and ResNet18 (as a baseline model) for training and evaluation, respectively.

A. Data Preprocessing

For the CIFAR-10 dataset, we used standard image preprocessing methods such as normalization and standardization. For MedMNIST, we used similar preprocessing methods to ensure that the data is balanced when input to the network.

B. Experimental Setup

In each experiment, we used the Adam optimizer with a learning rate of 0.001. During training, we used the cross entropy loss function for optimization. In our experiments, we divided the training process into five epochs and calculated the training loss for each epoch as well as the final test precision, recall, and F1 score.

Python code used in Experiment 1:

```
import torch
import torch.nn as nn
import torch.nn.functional as F
import torchvision
import torchvision.transforms as transforms
from torch.utils.data import DataLoader
import numpy as np
import random
from collections import Counter
from torchvision.models import resnet18
from sklearn.metrics import recall_score, f1_score
import time
# Set random seed for reproducibility
```

```
    def set_seed(seed=42):
        torch.manual_seed(seed)
        np.random.seed(seed)
        random.seed(seed)
        if torch.cuda.is_available():
            torch.cuda.manual_seed(seed)
    set_seed()
    device = torch.device("cuda" if torch.cuda.is_available() else "cpu")
    # Data Preprocessing
    transform = transforms.Compose([
        transforms.ToTensor(),
        transforms.Normalize((0.5, 0.5, 0.5), (0.5, 0.5, 0.5))
    ])
    # Use only 1000 images from CIFAR10
    train_dataset = torchvision.datasets.CIFAR10(root='./data', train=True,
                                                 download=True, transform=transform)
    train_loader = DataLoader(train_dataset, batch_size=64, shuffle=True)
    # Take the first 1000 images
    train_loader = torch.utils.data.DataLoader(train_dataset, batch_size=64,
sampler=torch.utils.data.SubsetRandomSampler(range(1000)))
    test_dataset = torchvision.datasets.CIFAR10(root='./data', train=False,
                                                download=True, transform=transform)
    test_loader = DataLoader(test_dataset, batch_size=64, shuffle=False)
    # ----------------------
    # 🎯 Custom Neural Network: MayNet
    # ----------------------
    class MayVoter(nn.Module):
        """ Single voting network module """
        def __init__(self):
            super(MayVoter, self).__init__()
            self.net = nn.Sequential(
                nn.Conv2d(3, 32, 3, padding=1),
                nn.ReLU(),
                nn.MaxPool2d(2),
                nn.Conv2d(32, 64, 3, padding=1),
                nn.ReLU(),
                nn.MaxPool2d(2),
                nn.Flatten(),
                nn.Linear(64 * 8 * 8, 256),
                nn.ReLU(),
                nn.Linear(256, 10)
            )
        def forward(self, x):
            return self.net(x)
    class MayNet(nn.Module):
        """ Ensemble majority voting network using May's Theorem """
        def __init__(self, num_voters=5):
            super(MayNet, self).__init__()
            self.voters = nn.ModuleList([MayVoter() for _ in range(num_voters)])
            self.num_voters = num_voters
        def forward(self, x):
            logits_list = [voter(x) for voter in self.voters]
            # Stack predictions from all voters
            preds = torch.stack([logit.argmax(dim=1) for logit in logits_list], dim=0)
            # Majority voting
            final_preds = []
            for i in range(preds.size(1)):
                votes = preds[:, i].tolist()
                most_common = Counter(votes).most_common(1)[0][0]
                final_preds.append(most_common)
            final_preds = torch.tensor(final_preds).to(x.device)
            return final_preds
    # MayNet outputs the final predicted class, not logits, which is
suitable for accuracy evaluation, but not for cross-entropy training
    def train_maynet(model, dataloader, optimizer, criterion, epochs=5):
        model.train()
        for epoch in range(epochs):
            total_loss = 0
            for images, labels in dataloader:
                images, labels = images.to(device), labels.to(device)
                for voter in model.voters:
                    optimizer.zero_grad()
                    outputs = voter(images)
                    loss = criterion(outputs, labels)
                    loss.backward()
                    optimizer.step()
                    total_loss += loss.item()
            print(f"Epoch {epoch+1}, Loss: {total_loss:.4f}")
    def evaluate(model, dataloader):
        model.eval()
        correct = 0
        total = 0
        all_preds = []
        all_labels = []
        with torch.no_grad():
            for images, labels in dataloader:
                images, labels = images.to(device), labels.to(device)
                preds = model(images)
                correct += (preds == labels).sum().item()
                total += labels.size(0)
                all_preds.extend(preds.cpu().numpy())
                all_labels.extend(labels.cpu().numpy())
        acc = correct / total
        recall = recall_score(all_labels, all_preds, average='macro')
        f1 = f1_score(all_labels, all_preds, average='macro')
        print(f"Test Accuracy: {acc * 100:.2f}%, Recall: {recall:.4f}, F1-
Score: {f1:.4f}")
        return acc, recall, f1
    # ----------------------
    # 🧪 Comparison Experiments
    # ----------------------
    # 🅰 Experiment 1: MayNet
    def run_maynet():
        maynet = MayNet(num_voters=5).to(device)
        optimizer_may = torch.optim.Adam(maynet.parameters(),
lr=0.001)
        criterion = nn.CrossEntropyLoss()
        print("Training MayNet...")
        train_maynet(maynet, train_loader, optimizer_may, criterion,
epochs=5)
        print("Evaluating MayNet...")
        return evaluate(maynet, test_loader)
    # 🅱 Experiment 2: ResNet18 (baseline)
    def train_resnet(model, dataloader, optimizer, criterion, epochs=5):
        model.train()
        for epoch in range(epochs):
            total_loss = 0
            for images, labels in dataloader:
                images, labels = images.to(device), labels.to(device)
                optimizer.zero_grad()
                outputs = model(images)
                loss = criterion(outputs, labels)
                loss.backward()
                optimizer.step()
                total_loss += loss.item()
            print(f"[ResNet18] Epoch {epoch+1}, Loss: {total_loss:.4f}")
    def evaluate_resnet(model, dataloader):
        model.eval()
        correct = 0
        total = 0
        all_preds = []
        all_labels = []
        with torch.no_grad():
            for images, labels in dataloader:
                images, labels = images.to(device), labels.to(device)
                outputs = model(images)
                preds = outputs.argmax(dim=1)
                correct += (preds == labels).sum().item()
                total += labels.size(0)
                all_preds.extend(preds.cpu().numpy())
                all_labels.extend(labels.cpu().numpy())
        acc = correct / total
        recall = recall_score(all_labels, all_preds, average='macro')
        f1 = f1_score(all_labels, all_preds, average='macro')
        print(f"[ResNet18] Test Accuracy: {acc * 100:.2f}%, Recall:
{recall:.4f}, F1-Score: {f1:.4f}")
```

```
        return acc, recall, f1
    def run_resnet():
        resnet = resnet18(num_classes=10).to(device)
        optimizer_resnet = torch.optim.Adam(resnet.parameters(), lr=0.001)
        criterion = nn.CrossEntropyLoss()
        print("\nTraining ResNet18...")
        train_resnet(resnet, train_loader, optimizer_resnet, criterion, epochs=5)
        print("Evaluating ResNet18...")
        return evaluate_resnet(resnet, test_loader)
    # ----------------------
    # Running the experiments 10 times and calculating mean and standard deviation
    # ----------------------
    def run_experiments():
        maynet_results = {"acc": [], "recall": [], "f1": [], "time": []}
        resnet_results = {"acc": [], "recall": [], "f1": [], "time": []}
        for _ in range(10):
            start_time = time.time()
            acc_may, recall_may, f1_may = run_maynet()
            maynet_results["acc"].append(acc_may)
            maynet_results["recall"].append(recall_may)
            maynet_results["f1"].append(f1_may)
            maynet_results["time"].append(time.time() - start_time)
            print(f'MayNet - Accuracy: {acc_may:.4f}, Recall: {recall_may:.4f}, F1-Score: {f1_may:.4f}, Time: {maynet_results['time'][-1]:.4f}s")
            start_time = time.time()
            acc_resnet, recall_resnet, f1_resnet = run_resnet()
            resnet_results["acc"].append(acc_resnet)
            resnet_results["recall"].append(recall_resnet)
            resnet_results["f1"].append(f1_resnet)
            resnet_results["time"].append(time.time() - start_time)
            print(f'ResNet18 - Accuracy: {acc_resnet:.4f}, Recall: {recall_resnet:.4f}, F1-Score: {f1_resnet:.4f}, Time: {resnet_results['time'][-1]:.4f}s")
        # Calculate mean and standard deviation
        def calculate_statistics(results):
            mean = {key: np.mean(value) for key, value in results.items()}
            std = {key: np.std(value) for key, value in results.items()}
            return mean, std
        maynet_mean, maynet_std = calculate_statistics(maynet_results)
        resnet_mean, resnet_std = calculate_statistics(resnet_results)
        print("\nFinal Results (Averaged over 10 runs):")
        print(f'MayNet - Accuracy: {maynet_mean['acc']:.4f} ± {maynet_std['acc']:.4f}, Recall: {maynet_mean['recall']:.4f} ± {maynet_std['recall']:.4f}, F1-Score: {maynet_mean['f1']:.4f} ± {maynet_std['f1']:.4f}, Time: {maynet_mean['time']:.4f} ± {maynet_std['time']:.4f}")
        print(f'ResNet18 - Accuracy: {resnet_mean['acc']:.4f} ± {resnet_std['acc']:.4f}, Recall: {resnet_mean['recall']:.4f} ± {resnet_std['recall']:.4f}, F1-Score: {resnet_mean['f1']:.4f} ± {resnet_std['f1']:.4f}, Time: {resnet_mean['time']:.4f} ± {resnet_std['time']:.4f}")
    run_experiments()
    Python code used in Experiment 2:
    import torch
    import torch.nn as nn
    import torch.nn.functional as F
    from torch.utils.data import DataLoader, SubsetRandomSampler
    import numpy as np
    import random
    from collections import Counter
    from torchvision.models import resnet18
    from sklearn.metrics import recall_score, f1_score
    import time
    import medmnist
    from medmnist import INFO
    from medmnist import PathMNIST
    import torchvision.transforms as transforms
    # Set random seed for reproducibility
    def set_seed(seed=42):
        torch.manual_seed(seed)
        np.random.seed(seed)
        random.seed(seed)
        if torch.cuda.is_available():
            torch.cuda.manual_seed(seed)
    set_seed()
    device = torch.device("cuda" if torch.cuda.is_available() else "cpu")
    # Download MedMNIST PathMNIST dataset
    info = INFO['pathmnist']
    DataClass = getattr(medmnist, info['python_class'])
    # Normalization
    transform = transforms.Compose([
        transforms.ToTensor(),
        transforms.Normalize(mean=[.5], std=[.5])
    ])
    # Load the dataset
    train_dataset = DataClass(split='train', transform=transform, download=True)
    test_dataset = DataClass(split='test', transform=transform, download=True)
    # Use only 1000 images for training
    train_loader = DataLoader(train_dataset, batch_size=64, sampler=SubsetRandomSampler(range(1000)))
    test_loader = DataLoader(test_dataset, batch_size=64, shuffle=False)
    # ----------------------
    # 🎯 MayNet (Same as Original)
    # ----------------------
    class MayVoter(nn.Module):
        def __init__(self):
            super(MayVoter, self).__init__()
            self.net = nn.Sequential(
                nn.Conv2d(3, 32, 3, padding=1),
                nn.ReLU(),
                nn.MaxPool2d(2),
                nn.Conv2d(32, 64, 3, padding=1),
                nn.ReLU(),
                nn.MaxPool2d(2),
                nn.Flatten(),
                nn.Linear(64 * 7 * 7, 256),  # PathMNIST image size is 28x28
                nn.ReLU(),
                nn.Linear(256, 9)  # PathMNIST has 9 classes
            )
        def forward(self, x):
            return self.net(x)
    class MayNet(nn.Module):
        def __init__(self, num_voters=5):
            super(MayNet, self).__init__()
            self.voters = nn.ModuleList([MayVoter() for _ in range(num_voters)])
        def forward(self, x):
            logits_list = [voter(x) for voter in self.voters]
            preds = torch.stack([logit.argmax(dim=1) for logit in logits_list], dim=0)
            final_preds = []
            for i in range(preds.size(1)):
                votes = preds[:, i].tolist()
                most_common = Counter(votes).most_common(1)[0][0]
                final_preds.append(most_common)
            final_preds = torch.tensor(final_preds).to(x.device)
            return final_preds
    def train_maynet(model, dataloader, optimizer, criterion, epochs=5):
        model.train()
        for epoch in range(epochs):
            total_loss = 0
            for images, labels in dataloader:
                images, labels = images.to(device), labels.squeeze().long().to(device)
                for voter in model.voters:
                    optimizer.zero_grad()
                    outputs = voter(images)
                    loss = criterion(outputs, labels)
                    loss.backward()
                    optimizer.step()
                    total_loss += loss.item()
            print(f"Epoch {epoch+1}, Loss: {total_loss:.4f}")

    def evaluate(model, dataloader):
        model.eval()
        correct = 0
```

```
        total = 0
        all_preds = []
        all_labels = []
        with torch.no_grad():
            for images, labels = images.to(device), labels.squeeze().long().to(device)
                preds = model(images)
                correct += (preds == labels).sum().item()
                total += labels.size(0)
                all_preds.extend(preds.cpu().numpy())
                all_labels.extend(labels.cpu().numpy())
        acc = correct / total
        recall = recall_score(all_labels, all_preds, average='macro')
        f1 = f1_score(all_labels, all_preds, average='macro')
        print(f"Test Accuracy: {acc * 100:.2f}%, Recall: {recall:.4f}, F1-Score: {f1:.4f}")
        return acc, recall, f1
    def run_maynet():
        maynet = MayNet(num_voters=5).to(device)
        optimizer_may = torch.optim.Adam(maynet.parameters(), lr=0.001)
        criterion = nn.CrossEntropyLoss()
        print("Training MayNet...")
        train_maynet(maynet, train_loader, optimizer_may, criterion, epochs=5)
        print("Evaluating MayNet...")
        return evaluate(maynet, test_loader)
    # -----------------------
    # 🅱 ResNet18 for Comparison
    # -----------------------
    def train_resnet(model, dataloader, optimizer, criterion, epochs=5):
        model.train()
        for epoch in range(epochs):
            total_loss = 0
            for images, labels = images.to(device), labels.squeeze().long().to(device)
                optimizer.zero_grad()
                outputs = model(images)
                loss = criterion(outputs, labels)
                loss.backward()
                optimizer.step()
                total_loss += loss.item()
            print(f"[ResNet18] Epoch {epoch+1}, Loss: {total_loss:.4f}")
    def evaluate_resnet(model, dataloader):
        model.eval()
        correct = 0
        total = 0
        all_preds = []
        all_labels = []
        with torch.no_grad():
            for images, labels = images.to(device), labels.squeeze().long().to(device)
                outputs = model(images)
                preds = outputs.argmax(dim=1)
                correct += (preds == labels).sum().item()
                total += labels.size(0)
                all_preds.extend(preds.cpu().numpy())
                all_labels.extend(labels.cpu().numpy())
        acc = correct / total
        recall = recall_score(all_labels, all_preds, average='macro')
        f1 = f1_score(all_labels, all_preds, average='macro')
        print(f"[ResNet18] Test Accuracy: {acc * 100:.2f}%, Recall: {recall:.4f}, F1-Score: {f1:.4f}")
        return acc, recall, f1
    def run_resnet():
        resnet = resnet18(num_classes=9).to(device) # PathMNIST has 9 classes
        optimizer_resnet = torch.optim.Adam(resnet.parameters(), lr=0.001)
        criterion = nn.CrossEntropyLoss()
        print("\nTraining ResNet18...")
        train_resnet(resnet, train_loader, optimizer_resnet, criterion, epochs=5)
        print("Evaluating ResNet18...")
        return evaluate_resnet(resnet, test_loader)
    # -----------------------
    # Main Experiment Loop
    # -----------------------
    def run_experiments():
        maynet_results = {"acc": [], "recall": [], "f1": [], "time": []}
        resnet_results = {"acc": [], "recall": [], "f1": [], "time": []}
        for _ in range(10):
            start_time = time.time()
            acc_may, recall_may, f1_may = run_maynet()
            maynet_results["acc"].append(acc_may)
            maynet_results["recall"].append(recall_may)
            maynet_results["f1"].append(f1_may)
            maynet_results["time"].append(time.time() - start_time)
            start_time = time.time()
            acc_resnet, recall_resnet, f1_resnet = run_resnet()
            resnet_results["acc"].append(acc_resnet)
            resnet_results["recall"].append(recall_resnet)
            resnet_results["f1"].append(f1_resnet)
            resnet_results["time"].append(time.time() - start_time)
        def calculate_statistics(results):
            mean = {key: np.mean(value) for key, value in results.items()}
            std = {key: np.std(value) for key, value in results.items()}
            return mean, std

        maynet_mean, maynet_std = calculate_statistics(maynet_results)
        resnet_mean, resnet_std = calculate_statistics(resnet_results)
        print("\nFinal Results (Averaged over 10 runs):")
        print(f"MayNet - Accuracy: {maynet_mean['acc']:.4f} ± {maynet_std['acc']:.4f}, Recall: {maynet_mean['recall']:.4f} ± {maynet_std['recall']:.4f}, F1-Score: {maynet_mean['f1']:.4f} ± {maynet_std['f1']:.4f}, Time: {maynet_mean['time']:.4f} ± {maynet_std['time']:.4f}")
        print(f"ResNet18 - Accuracy: {resnet_mean['acc']:.4f} ± {resnet_std['acc']:.4f}, Recall: {resnet_mean['recall']:.4f} ± {resnet_std['recall']:.4f}, F1-Score: {resnet_mean['f1']:.4f} ± {resnet_std['f1']:.4f}, Time: {resnet_mean['time']:.4f} ± {resnet_std['time']:.4f}")

                    run_experiments()
```

C. *Experimental Results*

After 10 experimental runs, the performance results of MayNet and ResNet18 on the CIFAR-10 dataset are as follows:

Final Results (Averaged over 10 runs):

MayNet - Accuracy: 0.4032 ± 0.0109, Recall: 0.4033 ± 0.0109, F1-Score: 0.3888 ± 0.0110, Time: 95.4677 ± 15.9435

ResNet18 - Accuracy: 0.3280 ± 0.0121, Recall: 0.3280 ± 0.0121, F1-Score: 0.3179 ± 0.0163, Time: 309.3771 ± 48.8502

After 10 experimental runs, the performance results of MayNet and ResNet18 on the MedMNIST PathMNIST dataset are as follows:

Final Results (Averaged over 10 runs):

MayNet - Accuracy: 0.5825 ± 0.0309, Recall: 0.5035 ± 0.0208, F1-Score: 0.4770 ± 0.0271, Time: 39.9136 ± 4.1372

ResNet18 - Accuracy: 0.4926 ± 0.1140, Recall: 0.4726 ± 0.0922, F1-Score: 0.4410 ± 0.0937, Time: 71.2365 ± 9.4099

The results show that MayNet outperforms ResNet18 in accuracy, recall, and F1 score, and the training time is significantly shortened. This shows that MayNet performs well on the CIFAR-10 dataset and the MedMNIST PathMNIST dataset.

V. DISCUSSION

The experimental results show that MayNet performs significantly better than traditional single neural network models on multiple standard datasets, especially on the

CIFAR-10 and MedMNIST datasets. These results verify the effectiveness and advantages of ensemble learning methods in deep learning. By combining multiple independent voting models, the ensemble method not only improves the accuracy of the final prediction results, but also shows stronger robustness and adaptability when facing different types of datasets. Specifically, MayNet effectively balances the limitations of each single model and significantly improves the classification ability of the overall system by integrating the prediction results of multiple neural networks. The advantage of this integration strategy is that it makes full use of the features learned by each model in different training processes, thereby alleviating the overfitting problem and bias that may exist in a single model, resulting in a more stable and efficient classification performance.

Compared with a single neural network, MayNet can better handle complex classification tasks with the cooperation of multiple voters. Each voter is an independently trained neural network, so it presents a different decision-making perspective when processing data. This diversity makes MayNet's final prediction more accurate, enabling it to adapt to the diversity of data and reduce the impact of noise and outliers on the final decision. In addition, the experimental results show that MayNet has superior computational performance. Despite the integration of multiple voters, the training time remains within a reasonable range and the inference speed is effectively guaranteed. This proves that MayNet not only improves the classification performance of the model, but also maintains a high training and inference efficiency, and has the potential to be deployed in practical applications.

However, although MayNet's integration method can significantly improve the classification performance, it also has limitations. For example, increasing the number of voters will lead to a significant increase in training and inference time, especially when dealing with large datasets. Therefore, how to optimize the computational cost in the integration process while ensuring model accuracy remains an important direction for future research. Moreover, MayNet's voting strategy is based on the majority voting mechanism, which can achieve ideal results in most cases. However, in some cases, more complex weighted voting strategies may be required to further improve the integration effect. Therefore, in the future, we can explore combining other integration strategies (such as weighted voting, random forest) to further improve the performance of MayNet on more complex tasks and datasets.

VI. Conclusion

The MayNet model proposed in this paper successfully applies the majority voting principle in May's Theorem to the ensemble learning of neural networks. Experimental results show that MayNet has achieved significant performance improvements on CIFAR-10 and MedMNIST datasets. Compared with traditional single neural networks, MayNet not only improves the classification accuracy of the model, but also enhances the robustness of the model, showing strong generalization ability and stability. Moreover, MayNet's efficient training and inference capabilities can reduce the consumption of computing resources while ensuring the accuracy of processing practical application tasks. Therefore, MayNet has wide applicability in the field of deep learning, especially under the condition of limited computing resources.

Future research can further explore the scalability of MayNet in more complex tasks and larger datasets, and explore how to optimize the training process to cope with the challenges of different types of datasets. In addition, combining more advanced ensemble learning techniques such as weighted voting and adaptive voting strategies can further improve the performance of MayNet. With the continuous development of neural network ensemble methods, MayNet is expected to become a standard model for various application scenarios and promote the widespread application of ensemble learning in practical problems.

Moreover, MayNet's modular design makes it flexible, adaptable, and easy to integrate into existing deep learning architectures. MayNet can serve as a unified wrapper for various basic neural networks, including CNNs, Transformers, and even hybrid models, thus extending its application from image classification to a wider range of machine learning tasks.

References

[1] M. Mamun, A. Farjana, M. Al Mamun and M. S. Ahammed, "Lung cancer prediction model using ensemble learning techniques and a systematic review analysis," *2022 IEEE World AI IoT Congress (AIIoT)*, pp. 187-193, 2022, https://doi.org/10.1109/AIIoT54504.2022.9817326.

[2] F. Matloob *et al*., "Software Defect Prediction Using Ensemble Learning: A Systematic Literature Review," *IEEE Access*, vol. 9, pp. 98754-98771, 2021, https://doi.org/10.1109/ACCESS.2021.3095559.

[3] S. Lin, H. Zheng, B. Han, Y. Li, C. Han, and W. Li, "Comparative performance of eight ensemble learning approaches for the development of models of slope stability prediction," *Acta Geotech.*, vol. 17, no. 4, pp. 1477-1502, 2022, https://doi.org/10.1007/s11440-021-01440-1.

[4] A. Abbasi, A. R. Javed, F. Iqbal, Z. Jalil, T. R. Gadekallu, and N. Kryvinska, "Authorship identification using ensemble learning," *Scientific Reports*, vol. 12, no. 1, p. 9537, 2022, https://doi.org/10.1038/s41598-022-13690-4.

[5] N. Thockchom, M. M. Singh, and U. Nandi, "A novel ensemble learning-based model for network intrusion detection," *Complex & Intelligent Systems*, vol. 9, no. 5, pp. 5693-5714, 2023, https://doi.org/10.1007/s40747-023-01013-7.

[6] Z. Mian *et al*., "A literature review of fault diagnosis based on ensemble learning," *Engineering Applications of Artificial Intelligence*, vol. 127, p. 107357, 2024, https://doi.org/10.1016/j.engappai.2023.107357.

[7] A. A. Khan, O. Chaudhari, and R. Chandra, "A review of ensemble learning and data augmentation models for class imbalanced problems: Combination, implementation and evaluation," *Expert Systems with Applications*, vol. 244, p. 122778, 2024, https://doi.org/10.1016/j.eswa.2023.122778.

[8] I. D. Mienye and Y. Sun, "A Survey of Ensemble Learning: Concepts, Algorithms, Applications, and Prospects," *IEEE Access*, vol. 10, pp. 99129-99149, 2022, https://doi.org/10.1109/ACCESS.2022.3207287.

[9] T. Mahmud, A. Barua, M. Begum, E. Chakma, S. Das and N. Sharmen, "An Improved Framework for Reliable Cardiovascular Disease Prediction Using Hybrid Ensemble Learning," *2023 International Conference on Electrical, Computer and Communication Engineering (ECCE)*, pp. 1-6, 2023, https://doi.org/10.1109/ECCE57851.2023.10101564.

[10] M. Y. Junior, R. Z. Freire, L. O. Seman, S. F. Stefenon, V. C. Mariani, and L. dos Santos Coelho, "Optimized hybrid ensemble learning approaches applied to very short-term load forecasting," *International Journal of Electrical Power & Energy Systems*, vol. 155, p. 109579, 2024, https://doi.org/10.1016/j.ijepes.2023.109579.

[11] J. Dong, Q. Zhang, X. Huang, Q. Tan, D. Zha, and Z. Zihao, "Active ensemble learning for knowledge graph error detection," *Proceedings of the Sixteenth ACM International Conference on Web Search and Data Mining*, pp. 877-885, 2023, https://doi.org/10.1145/3539597.3570368.

[12] W. Kang, L. Lin, B. Zhang, X. Shen, S. Wu, and Alzheimer's Disease Neuroimaging Initiative, "Multi-model and multi-slice ensemble learning architecture based on 2D convolutional neural networks for Alzheimer's disease diagnosis," *Computers in Biology and Medicine*, vol. 136, p. 104678, 2021, https://doi.org/10.1016/j.compbiomed.2021.104678.

[13] M. Hasan, M. Z. Abedin, P. Hajek, K. Coussement, M. N. Sultan, and B. Lucey, "A blending ensemble learning model for crude oil price forecasting," *Annals of Operations Research*, pp. 1-31, 2024, https://doi.org/10.1007/s10479-023-05810-8.

[14] T. T. Khoei, G. Aissou, W. C. Hu and N. Kaabouch, "Ensemble Learning Methods for Anomaly Intrusion Detection System in Smart Grid," *2021 IEEE International Conference on Electro Information Technology (EIT)*, pp. 129-135, 2021, https://doi.org/10.1109/EIT51626.2021.9491891.

[15] M. A. Muslim *et al*., "New model combination meta-learner to improve accuracy prediction P2P lending with stacking ensemble learning," *Intelligent Systems with Applications*, vol. 18, p. 200204, 2023, https://doi.org/10.1016/j.iswa.2023.200204.

[16] D. Müller, I. Soto-Rey and F. Kramer, "An Analysis on Ensemble Learning Optimized Medical Image Classification With Deep Convolutional Neural Networks," *IEEE Access*, vol. 10, pp. 66467-66480, 2022, https://doi.org/10.1109/ACCESS.2022.3182399.

[17] H. Zhou, Y. Xin, and S. Li, "A diabetes prediction model based on Boruta feature selection and ensemble learning," *BMC Bioinformatics*, vol. 24, no. 1, p. 224, 2023, https://doi.org/10.1186/s12859-023-05300-5.

[18] J. Yan *et al*., "LightGBM: accelerated genomically designed crop breeding through ensemble learning," *Genome Biology*, vol. 22, p. 1-24, 2021, https://doi.org/10.1186/s13059-021-02492-y.

[19] A. C. Mazari, N. Boudoukhani, and A. Djeffal, "BERT-based ensemble learning for multi-aspect hate speech detection," *Cluster Computing*, vol. 27, no. 1, pp. 325-339, 2024, https://doi.org/10.1007/s10586-022-03956-x.

[20] C. Hazman, A. Guezzaz, S. Benkirane, and M. Azrour, "lIDS-SIoEL: intrusion detection framework for IoT-based smart environments security using ensemble learning," *Cluster Computing*, vol. 26, no. 6, pp. 4069-4083, 2023, https://doi.org/10.1007/s10586-022-03810-0.

[21] H. Alamro, W. Mtouaa, S. Aljameel, A. S. Salama, M. A. Hamza and A. Y. Othman, "Automated Android Malware Detection Using Optimal Ensemble Learning Approach for Cybersecurity," *IEEE Access*, vol. 11, pp. 72509-72517, 2023, https://doi.org/10.1109/ACCESS.2023.3294263.

[22] V. Jaiswal, P. Saurabh, U. K. Lilhore, M. Pathak, S. Simaiya, and S. Dalal, "A breast cancer risk predication and classification model with ensemble learning and big data fusion," *Decision Analytics Journal*, vol. 8, p. 100298, 2023, https://doi.org/10.1016/j.dajour.2023.100298.

[23] W. Zehra, A. R. Javed, Z. Jalil, H. U. Khan, and T. R. Gadekallu, "Cross corpus multi-lingual speech emotion recognition using ensemble learning," *Complex & Intelligent Systems*, vol. 7, no. 4, pp. 1845-1854, 2021, https://doi.org/10.1007/s40747-020-00250-4.

[24] E. S. M. El-Kenawy *et al*., "Sunshine duration measurements and predictions in Saharan Algeria region: An improved ensemble learning approach," *Theoretical and Applied Climatology*, vol. 147, pp. 1015-1031, 2022, https://doi.org/10.1007/s00704-021-03843-2.

[25] S. Mohammadi, Z. Narimani, M. Ashouri, R. Firouzi, and M. H. Karimi-Jafari, "Ensemble learning from ensemble docking: Revisiting the optimum ensemble size problem," *Scientific Reports*, vol. 12, no. 1, p. 410, 2022, https://doi.org/10.1038/s41598-021-04448-5.

[26] M. Charikar and P. Ramakrishnan, "Metric distortion bounds for randomized social choice," *Proceedings of the 2022 Annual ACM-SIAM Symposium on Discrete Algorithms*, pp. 2986-3004, 2022, https://doi.org/10.1137/1.9781611977073.116.

[27] X. Bei, X. Lu, and W. Suksompong, "Truthful cake sharing," *Social Choice and Welfare*, vol. 64, no. 1, pp. 309-343, 2025, https://doi.org/10.1007/s00355-023-01503-0.

[28] D. Müller and S. Renes, "Fairness views and political preferences: evidence from a large and heterogeneous sample," *Social Choice and Welfare*, vol. 56, no. 4, pp. 679-711, 2021, https://doi.org/10.1007/s00355-020-01289-5.

[29] E. Anshelevich and W. Zhu, "Ordinal approximation for social choice, matching, and facility location problems given candidate positions," *ACM Transactions on Economics and Computation (TEAC)*, vol. 9, no. 2, pp. 1-24, 2021, https://doi.org/10.1145/3434417.

[30] L. Kellerhals and J. Peters, "Proportional fairness in clustering: A social choice perspective," *ArXiv*, 2024, https://doi.org/10.48550/arXiv.2310.18162.

[31] M. Brill, J. Israel, E. Micha, and J. Peters, "Individual representation in approval-based committee voting," *Social Choice and Welfare*, vol. 64, no. 1, pp. 69-96, 2025, https://doi.org/10.1007/s00355-024-01563-w.

[32] M. I. Jones, A. D. Sirianni, and F. Fu, "Polarization, abstention, and the median voter theorem," *Humanities and Social Sciences Communications*, vol. 9, no. 1, pp. 1-12, 2022, https://doi.org/10.1057/s41599-022-01056-0.

[33] L. Bulteau, N. Hazon, R. Page, A. Rosenfeld and N. Talmon, "Justified Representation for Perpetual Voting," *IEEE Access*, vol. 9, pp. 96598-96612, 2021, https://doi.org/10.1109/ACCESS.2021.3095087.