A Novel Incentive-Compatible Neural Network Optimization Model (ICNNOM) with Optimal Contract Structure

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, [2] 172825661@qq.com
*Corresponding Author

***Abstract*—In this paper, we propose a novel neural network optimization framework called the Incentive Compatible Neural Network Optimization Model (ICNNOM). This model combines the incentive compatibility idea in game theory with the optimal contract theory to simulate the "incentive and effort" mechanism between the internal layers of a deep neural network, aiming to improve the learning effect of the network. This paper uses two sets of codes with different architectures to conduct experiments on the CIFAR-10 and CIFAR-100 datasets and compares them with traditional neural network models. The results show that ICNNOM outperforms traditional models in multiple evaluation indicators such as accuracy, precision, recall, and F1 value, proving the effectiveness of introducing incentive mechanisms for model optimization. Incentive compatibility (IC) refers to designing mechanisms so that each participant's best interest aligns with truthful or cooperative behavior, while optimal contract theory studies designing agreements to maximize benefits under informational asymmetry. By integrating these concepts, ICNNOM explicitly coordinates the effort of each neural network layer to improve overall training consistency and efficiency.**

Keywords*—*Incentive Compatibility, Neural Networks, Game Theory, Model Performance

I. INTRODUCTION

In recent years, deep neural networks (DNNs) have made breakthroughs in multiple sub-fields of artificial intelligence, such as computer vision and natural language processing, and have promoted the practical application of technologies such as autonomous driving, speech recognition, and intelligent question answering [1]-[6]. Due to its powerful feature extraction capabilities and end-to-end learning mechanism, DNNs have become the core of modern artificial intelligence models [7]-[12]. However, although these models continue to break new records in task performance, traditional neural networks still have limitations in structural design, especially the lack of effective multi-layer collaborative optimization mechanisms [13]-[20].

Specifically, existing neural network models usually process information layer by layer. Although global optimization of parameters can be achieved through the error backpropagation algorithm, each layer usually only receives gradient updates from the global error signal during training, and does not receive independent explicit activation feedback [21]-[26]. This "passive update" mechanism is prone to information redundancy and target deviation in the process of feature representation and information compression in the intermediate layers, further affecting the overall performance of the model. In other words, there is a lack of "behavior coordination mechanism" between the layers of the neural network. As a learning goal, it is a challenge for each layer to maintain a consistent positive contribution to the final output target, which may lead to potential bottlenecks in training efficiency and generalization ability.

In response to the above problems, this paper introduces the concept of incentive compatibility in game theory, and uses the incentive design concept of the "master-agent model" to compare the neural network training process to a cooperative game between a "master organization" and multiple "agent layers". In this framework, each hidden layer of the network is regarded as an independent rational agent. Its main structure is to design a reasonable contract reward mechanism to guide each layer to improve its output performance while being consistent with the overall goal. To this end, this paper proposes a novel neural network optimization method - ICNNOM (Incentive Compatible Neural Network Optimization Mechanism). Based on the original network structure, this method introduces the "contract reward module" and "effort reward" loss function to simulate the rational interaction process between multi-layer neural units.

ICNNOM explicitly models the consistency between the performance of each hidden layer and the final prediction target, and feeds back to each agent layer through the loss function. While maintaining the simplicity of the original model structure, it effectively improves the efficiency of inter-layer coordination and information flow, and improves the interpretability and performance stability of the model.

ICNNOM explicitly models the consistency between the performance of each hidden layer and the final prediction target, and feeds back to each agent layer through the loss function. While maintaining the simplicity of the original model structure, it effectively improves the efficiency of inter-layer coordination and information flow, and improves the interpretability and performance stability of the model.

II. MODEL DESIGN

In order to effectively achieve incentive compatibility in neural networks, this paper constructs a model architecture with the feature of "agent interaction", namely the incentive compatible neural network optimization mechanism

(ICNNOM). ICNNOM not only retains the structural advantages of traditional feedforward neural networks, but also establishes a cross-layer, goal-consistent optimization mechanism by introducing contract modules and incentive loss mechanisms [27]-[32]. In this section, we elaborate on our model construction ideas from three aspects: basic structure, incentive module design, and overall loss function.

A. Infrastructure

The basic architecture of ICNNOM is a typical fully connected feedforward neural network, including an input layer, two hidden layers, and an output layer. The hierarchical information transfer method is used for feature transformation. Its structural design is as follows:

From the input layer to the first hidden layer, a combination of Linear→LeakyReLU→Dropout is used. Linear implements linear transformation, LeakyReLU introduces nonlinear activation (to prevent neurons from "dying"), and Dropout is used to improve the generalization ability of the model and reduce the risk of overfitting.

Hidden layer 1-2: The same structure as the previous layer is used to further extract intermediate feature representations.

Second hidden layer to output layer: The last layer performs a linear mapping to the output space.

This basic network structure not only ensures computational efficiency, but also provides a clear hierarchical division for the insertion of subsequent incentive modules. Specifically, the output of each hidden layer in the model is regarded as the "effort performance" of the "agent" and is directly input into the contract reward evaluation process.

In addition, in order to ensure the versatility and scalability of the modular design, the size parameters of each layer of the model can be flexibly set, and the dropout rate and activation function can also be adjusted according to the specific task to adapt to data sets of different scales and complexities.

B. Incentive Module Design

The core innovation of ICNNOM lies in the "incentive model" mechanism of the middle layer. A contract-based reward module is introduced immediately after the output of each hidden layer to simulate the contract behavior between the agent and the main structure in the real game, so as to encourage the output of the middle layer to match the final model goal. The specific design is as follows:

The shared_proj1 and shared_proj2 modules correspond to the output projections of the first and second hidden layers, respectively. Its function is to transform the output of the middle layer to the same space as the final output to facilitate subsequent consistency evaluation. This process is equivalent to simulating the intermediate layers to "estimate" the final result, and then evaluating the quality of the output through the contract.

Consistency measure (KL divergence): Kullback-Leibler divergence is used to measure the difference between the projected output of each layer and the final prediction. This difference can be thought of as the "target deviation" of the layer. The smaller the difference, the more "conform to the contract" the agent layer is in achieving its main goal.

Effort cost modeling (cross entropy loss): Each layer must also "work hard" for its output. By calculating the cross entropy loss of the actual label, we can simulate the "cost" or "effort" required for the agent to achieve better results.

Combined with the incentive function and budget constraint, the total reward of each layer is the weighted sum of consistency (KL divergence) and effort cost (cross entropy). The sum of incentive losses of all layers is limited by the total budget constraint (e.g., ≤ 3.0) to prevent the agent from over-occupying resources and causing global imbalance.

The design aims to improve the quality of the overall optimization by "reasonably setting the terms of the contract", so that each hidden layer can actively adapt to the final goal to the maximum extent without increasing the complexity of the structure.

C. Overall Loss Function

After integrating all modules, the training goal of ICNNOM is to minimize the total loss function. Total loss = main task cross entropy loss + $\lambda \times$ incentive loss.

Among them, the main task loss is the cross entropy between the final output of the model and the actual label, which is used to ensure basic classification performance. Incentive loss focuses on the compatibility and effort performance of the middle-layer contract, and is the core driving force of multi-layer collaborative optimization. λ (set to 0.4 by default) is an adjustment factor used to balance the importance of the main task and the incentive mechanism.

During the training process, the incentive loss guides the middle layer to gradually converge in a more consistent and collaborative direction, avoiding the occurrence of the "information island" problem, and stimulating the expression ability of each layer, thereby improving the overall performance of the model in the global optimization framework. Experimental results show that the incentive matching structure significantly improves the generalization ability and convergence speed of the model. Pseudocode for ICNNOM Model:

```
    Initialize ICNNOM model:
    Layer1: FullyConnected(input_size=32*32*3, output_size=256) + LeakyReLU + Dropout
        Layer2: FullyConnected(256, 128) + LeakyReLU + Dropout
        Layer3: FullyConnected(128, num_classes)  # num_classes = 10 or 100
        ContractModules:
            SharedProj1: FullyConnected(256, num_classes) + Tanh
            SharedProj2: FullyConnected(128, num_classes) + Tanh
        ContractWeight = [0.6, 0.4]  # Fixed optimal contract ratio
    Define forward pass(input):
        out1 = Layer1(input)
        out2 = Layer2(out1)
        final_output = Layer3(out2)
        return final_output, [out1, out2]
    Define calculate_incentive_rewards(model, layer_outputs, final_output, target):
        final_log_softmax = LogSoftmax(final_output)
        projections = [model.SharedProj1, model.SharedProj2]
        alpha = [log(1 + i + 2) for i in range(len(layer_outputs))]  # Marginal return weights
        total_reward = 0
        For each i in range(len(layer_outputs)):
            proj_output = projections[i](layer_outputs[i])
            layer_softmax = Softmax(proj_output)
            ce_loss = CrossEntropyLoss(proj_output, target)  # Effort cost
            kl_loss = KL_Divergence(final_log_softmax, layer_softmax)  # Consistency penalty
            reward = alpha[i] * (ce_loss + 0.7 * kl_loss)  # Incentive function
            total_reward += reward
```

```
        budget_constraint = Clamp(total_reward, max=3.0)
        return budget_constraint
    Define training_step(model, data_batch, target_batch, optimizer):
        Set model to training mode
        Flatten data_batch to shape [batch_size, 32*32*3]
        optimizer.zero_grad()
        If model is ICNNOM:
            output, layer_outputs = model.forward(data_batch)
            reward_loss = calculate_incentive_rewards(model, layer_outputs, output, target_batch)
            main_loss = CrossEntropyLoss(output, target_batch)
            total_loss = main_loss + 0.4 * reward_loss
        Else:
            output = model.forward(data_batch)
            total_loss = CrossEntropyLoss(output, target_batch)
        Backpropagate total_loss
        optimizer.step()
    Define evaluate_model(model, test_loader):
        Set model to evaluation mode
        Initialize metrics: correct=0, total=0, total_loss=0
        Initialize lists for all_predictions and all_labels
        For each data_batch, target_batch in test_loader:
            Flatten data_batch
            If model is ICNNOM:
                output, _ = model.forward(data_batch)
            Else:
                output = model.forward(data_batch)
            loss = CrossEntropyLoss(output, target_batch)
            total_loss += loss * batch_size
            predicted = ArgMax(output)
            correct += CountMatches(predicted, target_batch)
            total += batch_size
            Append predicted and target_batch to respective lists
        accuracy = 100 * correct / total
        average_loss = total_loss / total
        precision = ComputePrecision(all_labels, all_predictions)
        recall = ComputeRecall(all_labels, all_predictions)
        f1_score = ComputeF1(all_labels, all_predictions)
        Return average_loss, accuracy, precision, recall, f1_score
    Main Experiment Procedure:
        For dataset in [CIFAR-10, CIFAR-100]:
            Load and preprocess dataset (normalize images to 32x32x3)
            Create train_loader and test_loader
            For model_type in [ICNNOM, SimpleNN]:
                Repeat experiment 10 times:
                    Initialize model (ICNNOM or SimpleNN) with appropriate output classes
                    Initialize Adam optimizer with lr=0.001
                    Record start time
                    For epoch in range(1, 21):
                        For each batch in train_loader:
                            training_step(model, batch_data, batch_target, optimizer)
                    Record elapsed time
                    Evaluate model on test_loader
                    Store metrics: loss, accuracy, precision, recall, F1, training time
                Calculate mean and standard deviation of metrics over 10 runs
                            Print summarized results
```

III. Experimental Design

A. Datasets

We conducted experiments on CIFAR-10 and CIFAR-100 datasets. CIFAR-10: 10 categories of images, 60,000 images in total. CIFAR-100: 100 different images, 60,000 images in total. The image size is normalized to 32×32×3.

B. Comparison Models

ICNNOM: This paper proposes an incentive-compatible optimization model. SimpleNN: A traditional three-layer fully connected network without an incentive mechanism.

C. Training Settings

Optimizer : Adam, learning rate 0.001.
Age : 20 years old;

Each experiment was repeated 10 times, and the average and standard deviation were calculated.

Evaluation indicators: loss, accuracy, precision, recall, F1 score, training time. Python code used in Experiment 1:

```
    import torch
    import torch.nn as nn
    import torch.optim as optim
    from torch.utils.data import DataLoader, Subset
    from torchvision import datasets, transforms
    import torch.nn.functional as F
    from sklearn.metrics import precision_score, recall_score, f1_score
    import time
    import numpy as np
    device = torch.device("cuda" if torch.cuda.is_available() else "cpu")
    # Improved ICNNOM model definition, integrating incentive compatibility and optimal contract idea
    class ICNNOM(nn.Module):
        def __init__(self):
            super(ICNNOM, self).__init__()
            self.layer1 = nn.Sequential(
                nn.Linear(32 * 32 * 3, 256),
                nn.LeakyReLU(0.1),
                nn.Dropout(0.3)
            )
            self.layer2 = nn.Sequential(
                nn.Linear(256, 128),
                nn.LeakyReLU(0.1),
                nn.Dropout(0.3)
            )
            self.layer3 = nn.Linear(128, 10)

            # Multiple "contract signing modules"
            self.shared_proj1 = nn.Sequential(nn.Linear(256, 10), nn.Tanh())
            self.shared_proj2 = nn.Sequential(nn.Linear(128, 10), nn.Tanh())
            self.contract_weight = nn.Parameter(torch.tensor([0.6, 0.4]), requires_grad=False) # Optimal contract ratio structure

            for proj in [self.shared_proj1, self.shared_proj2]:
                nn.init.xavier_uniform_(proj[0].weight)
        def forward(self, x):
            out1 = self.layer1(x)
            out2 = self.layer2(out1)
            out3 = self.layer3(out2)
            return out3, [out1, out2]
    # Incentive compatibility and optimal contract: calculate rewards at each layer and simulate effort-return mechanism
    def calculate_incentive_rewards(model, layer_outputs, final_output, target):
        rewards = []
        final_soft = F.log_softmax(final_output, dim=1)
        projections = [model.shared_proj1, model.shared_proj2]
        alpha = [np.log(1 + i + 2) for i in range(len(layer_outputs))] # Steeper marginal return function
        total_reward = 0
        for i, (output, proj, weight) in enumerate(zip(layer_outputs, projections, alpha)):
            proj_output = proj(output) # Simulate the "contract signing" return at this layer
            layer_soft = F.softmax(proj_output, dim=1)
            ce_loss = nn.CrossEntropyLoss()(proj_output, target) # Simulate "effort cost"
            kl_loss = F.kl_div(final_soft, layer_soft, reduction='batchmean') # Contract goal consistency penalty
            # Simulate "agent's return-effort gap": the higher the effort and target alignment, the more incentive
            reward = weight * (ce_loss + 0.7 * kl_loss) # Incentive function
            rewards.append(reward)
            total_reward += reward
```

```
        # Simulate contract budget constraint (total incentive does not exceed a certain level)
        budget_constraint = torch.clamp(total_reward, max=3.0)
        return budget_constraint
    # Simple reference model
    class SimpleNN(nn.Module):
        def __init__(self):
            super(SimpleNN, self).__init__()
            self.layer1 = nn.Linear(32 * 32 * 3, 128)
            self.layer2 = nn.Linear(128, 64)
            self.layer3 = nn.Linear(64, 10)
        def forward(self, x):
            x = F.relu(self.layer1(x))
            x = F.relu(self.layer2(x))
            return self.layer3(x)
    # Training process
    def train(model, train_loader, optimizer):
        model.train()
        for data, target in train_loader:
            data = data.view(-1, 32 * 32 * 3).to(device)
            target = target.to(device)
            optimizer.zero_grad()
            if isinstance(model, ICNNOM):
                output, layer_outputs = model(data)
                reward_loss = calculate_incentive_rewards(model, layer_outputs, output, target)
                main_loss = nn.CrossEntropyLoss()(output, target)
                loss = main_loss + 0.4 * reward_loss  # Slight increase in reward weight
            else:
                output = model(data)
                loss = nn.CrossEntropyLoss()(output, target)
            loss.backward()
            optimizer.step()
    # Evaluation function
    def evaluate(model, test_loader):
        model.eval()
        correct = 0
        total = 0
        total_loss = 0.0
        all_preds = []
        all_labels = []
        criterion = nn.CrossEntropyLoss()
        with torch.no_grad():
            for data, target in test_loader:
                data = data.view(-1, 32 * 32 * 3).to(device)
                target = target.to(device)
                if isinstance(model, ICNNOM):
                    output, _ = model(data)
                else:
                    output = model(data)
                loss = criterion(output, target)
                total_loss += loss.item() * data.size(0)
                pred = output.argmax(dim=1)
                correct += (pred == target).sum().item()
                total += target.size(0)
                all_preds.extend(pred.cpu().numpy())
                all_labels.extend(target.cpu().numpy())
        acc = 100 * correct / total
        avg_loss = total_loss / total
        precision = precision_score(all_labels, all_preds, average='macro', zero_division=0)
        recall = recall_score(all_labels, all_preds, average='macro', zero_division=0)
        f1 = f1_score(all_labels, all_preds, average='macro', zero_division=0)
        return avg_loss, acc, precision, recall, f1
    # CIFAR10 data loading
    transform = transforms.Compose([
        transforms.ToTensor(),
        transforms.Normalize((0.5, 0.5, 0.5), (0.5, 0.5, 0.5))
    ])
    train_data = datasets.CIFAR10(root='./data', train=True, download=True, transform=transform)
    train_subset = Subset(train_data, range(10000))
    train_loader = DataLoader(train_subset, batch_size=64, shuffle=True)
    test_data = datasets.CIFAR10(root='./data', train=False, download=True, transform=transform)
    test_loader = DataLoader(test_data, batch_size=64, shuffle=False)
    # Multiple experiment evaluations
    from collections import defaultdict
    results = defaultdict(list)
    for name in ['ICNNOM', 'SimpleNN']:
        print(f"\n{name} is being evaluated 10 times...")
        for run in range(10):
            model = ICNNOM().to(device) if name == 'ICNNOM' else SimpleNN().to(device)
            optimizer = optim.Adam(model.parameters(), lr=0.001)
            start_time = time.time()
            for epoch in range(1, 21):
                train(model, train_loader, optimizer)
            elapsed = time.time() - start_time
            loss, acc, precision, recall, f1 = evaluate(model, test_loader)
            results[name + '_loss'].append(loss)
            results[name + '_acc'].append(acc)
            results[name + '_precision'].append(precision)
            results[name + '_recall'].append(recall)
            results[name + '_f1'].append(f1)
            results[name + '_time'].append(elapsed)
            # Printing the time used for each run
            print(f"Run {run+1}: Loss={loss:.4f}, Acc={acc:.2f}%, Precision={precision:.4f}, Recall={recall:.4f}, F1={f1:.4f}, Time={elapsed:.2f}s")
    # Average statistics
    print("\n===== Average and standard deviation over 10 runs =====")
    for name in ['ICNNOM', 'SimpleNN']:
        print(f"\n{name}:")
        for metric in ['loss', 'acc', 'precision', 'recall', 'f1', 'time']:
            values = results[name + '_' + metric]
            mean = np.mean(values)
            std = np.std(values)
            print(f"{metric.capitalize():>9}: Mean = {mean:.4f}, Std = {std:.4f}")
    Python code used in Experiment 2:
    import torch
    import torch.nn as nn
    import torch.optim as optim
    from torch.utils.data import DataLoader, Subset
    from torchvision import datasets, transforms
    import torch.nn.functional as F
    from sklearn.metrics import precision_score, recall_score, f1_score
    import time
    import numpy as np
    device = torch.device("cuda" if torch.cuda.is_available() else "cpu")
    # Improved ICNNOM model definition, integrating incentive compatibility and optimal contract idea
    class ICNNOM(nn.Module):
        def __init__(self):
            super(ICNNOM, self).__init__()
            self.layer1 = nn.Sequential(
                nn.Linear(32 * 32 * 3, 256),
                nn.LeakyReLU(0.1),
                nn.Dropout(0.3)
            )
            self.layer2 = nn.Sequential(
                nn.Linear(256, 128),
                nn.LeakyReLU(0.1),
                nn.Dropout(0.3)
            )
            self.layer3 = nn.Linear(128, 100)  # Change to 100 for CIFAR-100
                # Multiple "contract signing modules"
            self.shared_proj1 = nn.Sequential(nn.Linear(256, 100), nn.Tanh())  # Change to 100 for CIFAR-100
            self.shared_proj2 = nn.Sequential(nn.Linear(128, 100), nn.Tanh())  # Change to 100 for CIFAR-100
            self.contract_weight = nn.Parameter(torch.tensor([0.6, 0.4]), requires_grad=False)  # Optimal contract ratio structure
            for proj in [self.shared_proj1, self.shared_proj2]:
                nn.init.xavier_uniform_(proj[0].weight)
        def forward(self, x):
```

```
        out1 = self.layer1(x)
        out2 = self.layer2(out1)
        out3 = self.layer3(out2)
        return out3, [out1, out2]
    # Incentive compatibility and optimal contract: calculate rewards at each layer and simulate effort-return mechanism
    def calculate_incentive_rewards(model, layer_outputs, final_output, target):
        rewards = []
        final_soft = F.log_softmax(final_output, dim=1)
        projections = [model.shared_proj1, model.shared_proj2]
        alpha = [np.log(1 + i + 2) for i in range(len(layer_outputs))]  # Steeper marginal return function
        total_reward = 0
        for i, (output, proj, weight) in enumerate(zip(layer_outputs, projections, alpha)):
            proj_output = proj(output)  # Simulate the "contract signing" return at this layer
            layer_soft = F.softmax(proj_output, dim=1)
            ce_loss = nn.CrossEntropyLoss()(proj_output, target)  # Simulate "effort cost"
            kl_loss = F.kl_div(final_soft, layer_soft, reduction='batchmean')  # Contract goal consistency penalty
            # Simulate "agent's return-effort gap": the higher the effort and target alignment, the more incentive
            reward = weight * (ce_loss + 0.7 * kl_loss) # Incentive function
            rewards.append(reward)
            total_reward += reward
        # Simulate contract budget constraint (total incentive does not exceed a certain level)
        budget_constraint = torch.clamp(total_reward, max=3.0)
        return budget_constraint
    # Simple reference model
    class SimpleNN(nn.Module):
        def __init__(self):
            super(SimpleNN, self).__init__()
            self.layer1 = nn.Linear(32 * 32 * 3, 128)
            self.layer2 = nn.Linear(128, 64)
            self.layer3 = nn.Linear(64, 100)  # Change to 100 for CIFAR-100
        def forward(self, x):
            x = F.relu(self.layer1(x))
            x = F.relu(self.layer2(x))
            return self.layer3(x)
    # Training process
    def train(model, train_loader, optimizer):
        model.train()
        for data, target in train_loader:
            data = data.view(-1, 32 * 32 * 3).to(device)
            target = target.to(device)
            optimizer.zero_grad()
            if isinstance(model, ICNNOM):
                output, layer_outputs = model(data)
                reward_loss = calculate_incentive_rewards(model, layer_outputs, output, target)
                main_loss = nn.CrossEntropyLoss()(output, target)
                loss = main_loss + 0.4 * reward_loss  # Slight increase in reward weight
            else:
                output = model(data)
                loss = nn.CrossEntropyLoss()(output, target)
            loss.backward()
            optimizer.step()
    # Evaluation function
    def evaluate(model, test_loader):
        model.eval()
        correct = 0
        total = 0
        total_loss = 0.0
        all_preds = []
        all_labels = []
        criterion = nn.CrossEntropyLoss()
        with torch.no_grad():
            for data, target in test_loader:
                data = data.view(-1, 32 * 32 * 3).to(device)
                target = target.to(device)
                if isinstance(model, ICNNOM):
                    output, _ = model(data)
                else:
                    output = model(data)
                loss = criterion(output, target)
                total_loss += loss.item() * data.size(0)
                pred = output.argmax(dim=1)
                correct += (pred == target).sum().item()
                total += target.size(0)
                all_preds.extend(pred.cpu().numpy())
                all_labels.extend(target.cpu().numpy())
        acc = 100 * correct / total
        avg_loss = total_loss / total
        precision = precision_score(all_labels, all_preds, average='macro', zero_division=0)
        recall = recall_score(all_labels, all_preds, average='macro', zero_division=0)
        f1 = f1_score(all_labels, all_preds, average='macro', zero_division=0)
        return avg_loss, acc, precision, recall, f1
    # CIFAR-100 data loading
    transform = transforms.Compose([
        transforms.ToTensor(),
        transforms.Normalize((0.5, 0.5, 0.5), (0.5, 0.5, 0.5))
    ])
    train_data = datasets.CIFAR100(root='./data', train=True, download=True, transform=transform)
    train_subset = Subset(train_data, range(10000))
    train_loader = DataLoader(train_subset, batch_size=64, shuffle=True)
    test_data = datasets.CIFAR100(root='./data', train=False, download=True, transform=transform)
    test_loader = DataLoader(test_data, batch_size=64, shuffle=False)
    # Multiple experiment evaluations
    from collections import defaultdict
    results = defaultdict(list)
    for name in ['ICNNOM', 'SimpleNN']:
        print(f"\n{name} is being evaluated 10 times...")
        for run in range(10):
            model = ICNNOM().to(device) if name == 'ICNNOM' else SimpleNN().to(device)
            optimizer = optim.Adam(model.parameters(), lr=0.001)
            start_time = time.time()
            for epoch in range(1, 21):
                train(model, train_loader, optimizer)
            elapsed = time.time() - start_time
            loss, acc, precision, recall, f1 = evaluate(model, test_loader)
            results[name + '_loss'].append(loss)
            results[name + '_acc'].append(acc)
            results[name + '_precision'].append(precision)
            results[name + '_recall'].append(recall)
            results[name + '_f1'].append(f1)
            results[name + '_time'].append(elapsed)
            # Printing the time used for each run
            print(f"Run {run+1}: Loss={loss:.4f}, Acc={acc:.2f}%, Precision={precision:.4f}, Recall={recall:.4f}, F1={f1:.4f}, Time={elapsed:.2f}s")
    # Average statistics
    print("\n===== Average and standard deviation over 10 runs =====")
    for name in ['ICNNOM', 'SimpleNN']:
        print(f"\n{name}:")
        for metric in ['loss', 'acc', 'precision', 'recall', 'f1', 'time']:
            values = results[name + '_' + metric]
            mean = np.mean(values)
            std = np.std(values)
            print(f"{metric.capitalize():>9}: Mean = {mean:.4f}, Std = {std:.4f}")
```

IV. Experimental Results

In this section, we show the performance of the ICNNOM model and the traditional SimpleNN model on several experimental tasks. In order to ensure the reliability of the experimental results, the experiment was repeated 10 times, and the average and standard deviation of each experiment were calculated. The specific results are as follows:

A. Output of the Two Experimental Codes

The following is the output of the Experiment 1 code:

```
===== Average and standard deviation over 10 runs =====
ICNNOM:
    Loss: Mean = 1.5888, Std = 0.0136
     Acc: Mean = 46.6210, Std = 0.3238
Precision: Mean = 0.4676, Std = 0.0041
  Recall: Mean = 0.4662, Std = 0.0032
      F1: Mean = 0.4638, Std = 0.0032
    Time: Mean = 130.0870, Std = 7.6289
SimpleNN:
    Loss: Mean = 2.7077, Std = 0.0514
     Acc: Mean = 43.7330, Std = 0.6423
Precision: Mean = 0.4425, Std = 0.0053
  Recall: Mean = 0.4373, Std = 0.0064
      F1: Mean = 0.4368, Std = 0.0063
    Time: Mean = 91.8828, Std = 2.3384
The following is the output of the Experiment 2 code:
===== Average and standard deviation over 10 runs =====
ICNNOM:
    Loss: Mean = 3.6401, Std = 0.0123
     Acc: Mean = 17.9400, Std = 0.4250
Precision: Mean = 0.1734, Std = 0.0054
  Recall: Mean = 0.1794, Std = 0.0042
      F1: Mean = 0.1669, Std = 0.0039
    Time: Mean = 145.1524, Std = 9.9955
SimpleNN:
    Loss: Mean = 5.0537, Std = 0.0342
     Acc: Mean = 15.1790, Std = 0.3905
Precision: Mean = 0.1544, Std = 0.0056
  Recall: Mean = 0.1518, Std = 0.0039
      F1: Mean = 0.1476, Std = 0.0042
Time: Mean = 89.4728, Std = 7.7531
```

B. Model Performance on Different Datasets

In the following experiments, we studied the performance of the ICNNOM model on various datasets and verified its generalization ability in multiple tasks. In our tests, we used more complex and simpler datasets.

On complex datasets, the performance of the ICNNOM model is more stable, and its accuracy and recall are significantly higher than SimpleNN. On simpler datasets, the performance gap is slightly narrowed, but ICNNOM still maintains a significant advantage. This further proves that the ICNNOM model can promote collaboration between layers through incentive mechanisms and effectively improve overall performance.

V. Results Analysis

The experimental results show that the ICNNOM model is significantly better than SimpleNN in multiple indicators. Let's take a closer look at the key indicators.

A. Loss Function and Model Convergence

The loss value of the ICNNOM model is significantly lower than that of SimpleNN, indicating that the incentive mechanism effectively optimizes the model training process. By introducing the contract reward module, ICNNOM not only enables each layer of the network to continuously adjust to the final goal, but also effectively avoids overfitting of the network. This mechanism stabilizes the ICNNOM loss at a low level and reduces fluctuations during training.

In contrast, SimpleNN lacks an effective inter-layer collaboration mechanism, resulting in higher loss values and slower convergence. This shows that SimpleNN cannot fully utilize inter-layer information during training, resulting in inefficiency.

B. Accuracy and Recall

ICNNOM performs well in both accuracy and recall, but accuracy is particularly prominent. The incentive module effectively reduces the inter-layer target deviation, allowing the network to more accurately judge positive and negative samples. At the same time, the improvement in recall shows that ICNNOM is able to identify as many positive samples as possible while reducing false positives.

SimpleNN performs poorly on both indicators, which is very different from ICNNOM, especially in recall. This shows that it is not powerful enough to handle complex patterns.

C. Time and Efficiency

Due to the introduction of incentive modules and reward mechanisms at each layer, the training time of ICNNOM is significantly longer than that of SimpleNN. This additional computational overhead is intended to improve the accuracy and stability of the model. In practical applications, if the inference time requirement is high, the time and accuracy can be balanced by optimizing the algorithm or reducing the number of layers.

VI. Conclusion

The experiments in this paper show that the ICNNOM model outperforms the traditional SimpleNN model in all performance indicators. By introducing an incentive-compatible mechanism, ICNNOM effectively improves inter-layer collaboration, ensures the consistency of model objectives, and achieves higher accuracy, precision, recall, and F1 values.

The training time of ICNNOM is slightly longer, but the performance improvement is very significant, so the price is acceptable. In future work, we will explore how to further optimize the computational efficiency of this model to reduce training time while maintaining high performance. In addition, the excitation module of ICNNOM can be extended to other types of neural networks, further verifying its wide applicability.

However, the increased training time may affect practical deployment scenarios, especially when computing resources are limited or latency requirements are strict. To address this issue, we explored possible optimization strategies such as model pruning, parameter quantization, and parallel computing. In addition, we plan to refine the future development path by incorporating advanced techniques such as hardware acceleration and efficient model compression. Finally, our ICNNOM framework is not only applicable to feedforward neural networks such as convolutional neural networks (CNNs), recurrent neural networks (RNNs), and Transformer architectures, but is also expected to show broad generalization capabilities across various deep learning models.

References

[1] K. Y. Chan *et al*., "Deep neural networks in the cloud: Review, applications, challenges and research directions," *Neurocomputing*, vol. 545, p. 126327, 2023, https://doi.org/10.1016/j.neucom.2023.126327.

[2] N. J. Cronin, "Using deep neural networks for kinematic analysis: Challenges and opportunities," *Journal of Biomechanics*, vol. 123, p. 110460, 2021, https://doi.org/10.1016/j.jbiomech.2021.110460.

[3] W. Peng, T. Varanka, A. Mostafa, H. Shi and G. Zhao, "Hyperbolic Deep Neural Networks: A Survey," *IEEE Transactions on Pattern Analysis and Machine Intelligence*, vol. 44, no. 12, pp. 10023-10044, 2022, https://doi.org/10.1109/TPAMI.2021.3136921.

[4] A. Halbouni, T. S. Gunawan, M. H. Habaebi, M. Halbouni, M. Kartiwi and R. Ahmad, "CNN-LSTM: Hybrid Deep Neural Network for Network Intrusion Detection System," *IEEE Access*, vol. 10, pp. 99837-99849, 2022, https://doi.org/10.1109/ACCESS.2022.3206425.

[5] S. Nazir, D. M. Dickson, and M. U. Akram, "Survey of explainable artificial intelligence techniques for biomedical imaging with deep neural networks," *Computers in Biology and Medicine*, vol. 156, p. 106668, 2023, https://doi.org/10.1016/j.compbiomed.2023.106668.

[6] X. Wu *et al*., "Sensing prior constraints in deep neural networks for solving exploration geophysical problems," *Proceedings of the National Academy of Sciences of the United States of America*, vol. 120, no. 23, p. e2219573120, 2023, https://doi.org/10.1073/pnas.2219573120.

[7] J. Gawlikowski *et al*., "A survey of uncertainty in deep neural networks," *Artificial Intelligence Review*, vol. 56, pp. 1513-1589, 2023, https://doi.org/10.1007/s10462-023-10562-9.

[8] G. Joshi, R. Walambe and K. Kotecha, "A Review on Explainability in Multimodal Deep Neural Nets," *IEEE Access*, vol. 9, pp. 59800-59821, 2021, https://doi.org/10.1109/ACCESS.2021.3070212.

[9] S. M. Mousavi, G. C. Beroza, T. Mukerji, and M. Rasht-Behesht, "Applications of deep neural networks in exploration seismology: A technical survey," *Geophysics*, vol. 89, no. 1, pp. WA95–WA115, 2024, https://doi.org/10.1190/geo2023-0063.1.

[10] M. A. Abdou, "Literature review: Efficient deep neural networks techniques for medical image analysis," *Neural Computing and Applications*, vol. 34, no. 8, pp. 5791-5812, 2022, https://doi.org/10.1007/s00521-022-06960-9.

[11] M. Li *et al*., "Guest Editorial: Deep Neural Networks for Graphs: Theory, Models, Algorithms, and Applications," *IEEE Transactions on Neural Networks and Learning Systems*, vol. 35, no. 4, pp. 4367-4372, 2024, https://doi.org/10.1109/TNNLS.2024.3371592.

[12] W. Samek, G. Montavon, S. Lapuschkin, C. J. Anders and K. -R. Müller, "Explaining Deep Neural Networks and Beyond: A Review of Methods and Applications," *Proceedings of the IEEE*, vol. 109, no. 3, pp. 247-278, 2021, https://doi.org/10.1109/JPROC.2021.3060483.

[13] G. C. Marinó, A. Petrini, D. Malchiodi, and M. Frasca, "Deep neural networks compression: A comparative survey and choice recommendations," *Neurocomputing*, vol. 520, pp. 152-170, 2023, https://doi.org/10.1016/j.neucom.2022.11.072.

[14] J. Guo, Y. Yang, H. Li, L. Dai, and B. Huang, "A parallel deep neural network for intelligent fault diagnosis of drilling pumps," *Engineering Applications of Artificial Intelligence*, vol. 133, p. 108071, 2024, https://doi.org/10.1016/j.engappai.2024.108071.

[15] M. Ivanovs, R. Kadikis, and K. Ozols, "Perturbation-based methods for explaining deep neural networks: A survey," *Pattern Recognition Letters*, vol. 150, pp. 228-234, 2021, https://doi.org/10.1016/j.patrec.2021.06.030.

[16] Y. Assael *et al*., "Restoring and attributing ancient texts using deep neural networks," *Nature*, vol. 603, no. 7900, pp. 280-283, 2022, https://doi.org/10.1038/s41586-022-04448-z.

[17] B. Rokh, A. Azarpeyvand, and A. Khanteymoori, "A comprehensive survey on model quantization for deep neural networks in image classification," *ACM Transactions on Intelligent Systems and Technology*, vol. 14, no. 6, pp. 1-50, 2023, https://doi.org/10.1145/3623402.

[18] M. A. Elaziz *et al*., "Advanced metaheuristic optimization techniques in applications of deep neural networks: a review," *Neural Computing and Applications*, vol. 33, pp. 14079-14099, 2021, https://doi.org/10.1007/s00521-021-05960-5.

[19] M. Woźniak, J. Siłka, and M. Wieczorek, "Deep neural network correlation learning mechanism for CT brain tumor detection," *Neural Computing and Applications*, vol. 35, no. 20, pp. 14611-14626, 2023, https://doi.org/10.1007/s00521-021-05841-x.

[20] M. Groh *et al*., "Evaluating Deep Neural Networks Trained on Clinical Images in Dermatology with the Fitzpatrick 17k Dataset," *2021 IEEE/CVF Conference on Computer Vision and Pattern Recognition Workshops (CVPRW)*, pp. 1820-1828, 2021, https://doi.org/10.1109/CVPRW53098.2021.00201.

[21] M. Kohler and S. Langer, "On the rate of convergence of fully connected deep neural network regression estimates," *Annals of Statistics*, vol. 49, no. 4, pp. 2231-2249, 2021, https://doi.org/10.1214/20-AOS2034.

[22] C. Liu *et al*., "A programmable diffractive deep neural network based on a digital-coding metasurface array," *Nature Electronics*, vol. 5, no. 2, pp. 113-122, 2022, https://doi.org/10.1038/s41928-022-00719-9.

[23] A. M. Roy, R. Bose, and J. Bhaduri, "A fast accurate fine-grain object detection model based on YOLOv4 deep neural network," *Neural Computing and Applications*, vol. 34, no. 5, pp. 3895-3921, 2022, https://doi.org/10.1007/s00521-021-06651-x.

[24] M. F. Degenhardt *et al*., "Determining structures of RNA conformers using AFM and deep neural networks," *Nature*, vol. 637, no. 8048, pp. 1234-1243, 2025, https://doi.org/10.1038/s41586-024-07559-x.

[25] W. Niu, J. Guan, Y. Wang, G. Agrawal, and B. Ren, "DNNFusion: accelerating deep neural networks execution with advanced operator fusion," *Proceedings of the 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation*, pp. 883-898, 2021, https://doi.org/10.1145/3453483.3454083.

[26] S. Hangaragi and T. Singh, "Face detection and recognition using Face Mesh and deep neural network," *Procedia Computer Science*, vol. 218, pp. 741-749, 2023, https://doi.org/10.1016/j.procs.2023.01.054.

[27] Y. Xie, C. Ding, Y. Li, and K. Wang, "Optimal incentive contract in continuous time with different behavior relationships between agents," *International Review of Financial Analysis*, vol. 86, p. 102521, 2023, https://doi.org/10.1016/j.irfa.2023.102521.

[28] D. Krähmer and R. Strausz, "Unidirectional incentive compatibility," *Journal of Economic Theory*, vol. 228, p. 106051, 2024, https://doi.org/10.1016/j.jet.2025.106051.

[29] H. Guo and N. C. Yannelis, "Incentive compatibility under ambiguity," *Economic Theory*, vol. 73, no. 2, pp. 565-593, 2022, https://doi.org/10.1007/s00199-020-01304-x.

[30] S. Pegoraro, "Incentives and performance with optimal money management contracts," *Journal of Political Economy*, vol. 131, no. 10, pp. 2920-2968, 2023, https://doi.org/10.1086/724576.

[31] Y. Chi and K. S. Tan, "Optimal incentive-compatible insurance with background risk," *ASTIN Bulletin*, vol. 51, no. 2, pp. 661-688, 2021, https://doi.org/10.1017/asb.2021.7.

[32] T. Zhang and Q. Zhu, "On incentive compatibility in dynamic mechanism design with exit option in a Markovian environment," *Dynamic Games and Applications*, vol. 12, no. 2, pp. 701-745, 2022, https://doi.org/10.1007/s13235-021-00388-x.