```python
import pandas as pd
import numpy as np
import os
import matplotlib.pyplot as plt
import seaborn as sns
import matplotlib.patches as patches
from sklearn.metrics import precision_recall_curve,
average_precision_score
from sklearn.preprocessing import LabelEncoder
from torchvision import transforms
from PIL import Image
from tqdm import tqdm
import shutil
from sklearn.metrics import roc_curve, auc
import torch
import torch.nn as nn
import torch.optim as optim
import os
import random
from PIL import Image, ImageEnhance
from tqdm import tqdm
import random
import math
import time
import pandas as pd
from collections import Counter
from torchvision import transforms, models
from torch.utils.data import Dataset, DataLoader, random_split
from sklearn.metrics import confusion_matrix, ConfusionMatrixDisplay

test_base_dir = "/kaggle/input/gtsrb-german-traffic-sign"
test_csv_path = f"{test_base_dir}/Test.csv"
model_path =
"/kaggle/input/theiss_model/pytorch/default/1/mobilenet_v2_traffic_sig
ns.pth"

# Load Test Data
test_df = pd.read_csv(test_csv_path)

# Custom Dataset Class
class CustomDataset(Dataset):
    def __init__(self, dataframe, base_dir, transform=None):
        self.dataframe = dataframe
        self.base_dir = base_dir
        self.transform = transform

    def __len__(self):
        return len(self.dataframe)

    def __getitem__(self, idx):
```

```
        img_path = os.path.join(self.base_dir,
self.dataframe.iloc[idx, -1])
        label = int(self.dataframe.iloc[idx, -2])
        image = Image.open(img_path).convert("RGB")
        if self.transform:
            image = self.transform(image)
        return image, label, img_path


test_dataset = CustomDataset(test_df, test_base_dir)

# Load the Trained Model
device = torch.device("cuda" if torch.cuda.is_available() else "cpu")
model =
models.mobilenet_v2(weights=models.MobileNet_V2_Weights.IMAGENET1K_V1)
model.classifier[1] = nn.Linear(model.last_channel, 43)  # 43 classes
model.load_state_dict(torch.load(model_path))
model = model.to(device)
model.eval()
```

```
Downloading: "https://download.pytorch.org/models/mobilenet_v2-
b0353104.pth" to /root/.cache/torch/hub/checkpoints/mobilenet_v2-
b0353104.pth
100%|██████████| 13.6M/13.6M [00:00<00:00, 73.1MB/s]
<ipython-input-8-bae117fd77ef>:5: FutureWarning: You are using
`torch.load` with `weights_only=False` (the current default value),
which uses the default pickle module implicitly. It is possible to
construct malicious pickle data which will execute arbitrary code
during unpickling (See
https://github.com/pytorch/pytorch/blob/main/SECURITY.md#untrusted-
models for more details). In a future release, the default value for
`weights_only` will be flipped to `True`. This limits the functions
that could be executed during unpickling. Arbitrary objects will no
longer be allowed to be loaded via this mode unless they are
explicitly allowlisted by the user via
`torch.serialization.add_safe_globals`. We recommend you start setting
`weights_only=True` for any use case where you don't have full control
of the loaded file. Please open an issue on GitHub for any issues
related to this experimental feature.
  model.load_state_dict(torch.load(model_path))
```

```
MobileNetV2(
  (features): Sequential(
    (0): Conv2dNormActivation(
      (0): Conv2d(3, 32, kernel_size=(3, 3), stride=(2, 2),
padding=(1, 1), bias=False)
      (1): BatchNorm2d(32, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
      (2): ReLU6(inplace=True)
    )
```

```
    (1): InvertedResidual(
      (conv): Sequential(
        (0): Conv2dNormActivation(
          (0): Conv2d(32, 32, kernel_size=(3, 3), stride=(1, 1),
padding=(1, 1), groups=32, bias=False)
          (1): BatchNorm2d(32, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
          (2): ReLU6(inplace=True)
        )
        (1): Conv2d(32, 16, kernel_size=(1, 1), stride=(1, 1),
bias=False)
        (2): BatchNorm2d(16, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
      )
    )
    (2): InvertedResidual(
      (conv): Sequential(
        (0): Conv2dNormActivation(
          (0): Conv2d(16, 96, kernel_size=(1, 1), stride=(1, 1),
bias=False)
          (1): BatchNorm2d(96, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
          (2): ReLU6(inplace=True)
        )
        (1): Conv2dNormActivation(
          (0): Conv2d(96, 96, kernel_size=(3, 3), stride=(2, 2),
padding=(1, 1), groups=96, bias=False)
          (1): BatchNorm2d(96, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
          (2): ReLU6(inplace=True)
        )
        (2): Conv2d(96, 24, kernel_size=(1, 1), stride=(1, 1),
bias=False)
        (3): BatchNorm2d(24, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
      )
    )
    (3): InvertedResidual(
      (conv): Sequential(
        (0): Conv2dNormActivation(
          (0): Conv2d(24, 144, kernel_size=(1, 1), stride=(1, 1),
bias=False)
          (1): BatchNorm2d(144, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
          (2): ReLU6(inplace=True)
        )
        (1): Conv2dNormActivation(
          (0): Conv2d(144, 144, kernel_size=(3, 3), stride=(1, 1),
padding=(1, 1), groups=144, bias=False)
```

```
          (1): BatchNorm2d(144, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
          (2): ReLU6(inplace=True)
        )
        (2): Conv2d(144, 24, kernel_size=(1, 1), stride=(1, 1),
bias=False)
        (3): BatchNorm2d(24, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
      )
    )
    (4): InvertedResidual(
      (conv): Sequential(
        (0): Conv2dNormActivation(
          (0): Conv2d(24, 144, kernel_size=(1, 1), stride=(1, 1),
bias=False)
          (1): BatchNorm2d(144, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
          (2): ReLU6(inplace=True)
        )
        (1): Conv2dNormActivation(
          (0): Conv2d(144, 144, kernel_size=(3, 3), stride=(2, 2),
padding=(1, 1), groups=144, bias=False)
          (1): BatchNorm2d(144, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
          (2): ReLU6(inplace=True)
        )
        (2): Conv2d(144, 32, kernel_size=(1, 1), stride=(1, 1),
bias=False)
        (3): BatchNorm2d(32, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
      )
    )
    (5): InvertedResidual(
      (conv): Sequential(
        (0): Conv2dNormActivation(
          (0): Conv2d(32, 192, kernel_size=(1, 1), stride=(1, 1),
bias=False)
          (1): BatchNorm2d(192, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
          (2): ReLU6(inplace=True)
        )
        (1): Conv2dNormActivation(
          (0): Conv2d(192, 192, kernel_size=(3, 3), stride=(1, 1),
padding=(1, 1), groups=192, bias=False)
          (1): BatchNorm2d(192, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
          (2): ReLU6(inplace=True)
        )
        (2): Conv2d(192, 32, kernel_size=(1, 1), stride=(1, 1),
```

```
bias=False)
        (3): BatchNorm2d(32, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
      )
    )
    (6): InvertedResidual(
      (conv): Sequential(
        (0): Conv2dNormActivation(
          (0): Conv2d(32, 192, kernel_size=(1, 1), stride=(1, 1),
bias=False)
          (1): BatchNorm2d(192, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
          (2): ReLU6(inplace=True)
        )
        (1): Conv2dNormActivation(
          (0): Conv2d(192, 192, kernel_size=(3, 3), stride=(1, 1),
padding=(1, 1), groups=192, bias=False)
          (1): BatchNorm2d(192, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
          (2): ReLU6(inplace=True)
        )
        (2): Conv2d(192, 32, kernel_size=(1, 1), stride=(1, 1),
bias=False)
        (3): BatchNorm2d(32, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
      )
    )
    (7): InvertedResidual(
      (conv): Sequential(
        (0): Conv2dNormActivation(
          (0): Conv2d(32, 192, kernel_size=(1, 1), stride=(1, 1),
bias=False)
          (1): BatchNorm2d(192, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
          (2): ReLU6(inplace=True)
        )
        (1): Conv2dNormActivation(
          (0): Conv2d(192, 192, kernel_size=(3, 3), stride=(2, 2),
padding=(1, 1), groups=192, bias=False)
          (1): BatchNorm2d(192, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
          (2): ReLU6(inplace=True)
        )
        (2): Conv2d(192, 64, kernel_size=(1, 1), stride=(1, 1),
bias=False)
        (3): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
      )
    )
```

```
    (8): InvertedResidual(
      (conv): Sequential(
        (0): Conv2dNormActivation(
          (0): Conv2d(64, 384, kernel_size=(1, 1), stride=(1, 1),
bias=False)
          (1): BatchNorm2d(384, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
          (2): ReLU6(inplace=True)
        )
        (1): Conv2dNormActivation(
          (0): Conv2d(384, 384, kernel_size=(3, 3), stride=(1, 1),
padding=(1, 1), groups=384, bias=False)
          (1): BatchNorm2d(384, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
          (2): ReLU6(inplace=True)
        )
        (2): Conv2d(384, 64, kernel_size=(1, 1), stride=(1, 1),
bias=False)
        (3): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
      )
    )
    (9): InvertedResidual(
      (conv): Sequential(
        (0): Conv2dNormActivation(
          (0): Conv2d(64, 384, kernel_size=(1, 1), stride=(1, 1),
bias=False)
          (1): BatchNorm2d(384, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
          (2): ReLU6(inplace=True)
        )
        (1): Conv2dNormActivation(
          (0): Conv2d(384, 384, kernel_size=(3, 3), stride=(1, 1),
padding=(1, 1), groups=384, bias=False)
          (1): BatchNorm2d(384, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
          (2): ReLU6(inplace=True)
        )
        (2): Conv2d(384, 64, kernel_size=(1, 1), stride=(1, 1),
bias=False)
        (3): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
      )
    )
    (10): InvertedResidual(
      (conv): Sequential(
        (0): Conv2dNormActivation(
          (0): Conv2d(64, 384, kernel_size=(1, 1), stride=(1, 1),
bias=False)
```

```
          (1): BatchNorm2d(384, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
          (2): ReLU6(inplace=True)
        )
        (1): Conv2dNormActivation(
          (0): Conv2d(384, 384, kernel_size=(3, 3), stride=(1, 1),
padding=(1, 1), groups=384, bias=False)
          (1): BatchNorm2d(384, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
          (2): ReLU6(inplace=True)
        )
        (2): Conv2d(384, 64, kernel_size=(1, 1), stride=(1, 1),
bias=False)
        (3): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
      )
    )
    (11): InvertedResidual(
      (conv): Sequential(
        (0): Conv2dNormActivation(
          (0): Conv2d(64, 384, kernel_size=(1, 1), stride=(1, 1),
bias=False)
          (1): BatchNorm2d(384, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
          (2): ReLU6(inplace=True)
        )
        (1): Conv2dNormActivation(
          (0): Conv2d(384, 384, kernel_size=(3, 3), stride=(1, 1),
padding=(1, 1), groups=384, bias=False)
          (1): BatchNorm2d(384, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
          (2): ReLU6(inplace=True)
        )
        (2): Conv2d(384, 96, kernel_size=(1, 1), stride=(1, 1),
bias=False)
        (3): BatchNorm2d(96, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
      )
    )
    (12): InvertedResidual(
      (conv): Sequential(
        (0): Conv2dNormActivation(
          (0): Conv2d(96, 576, kernel_size=(1, 1), stride=(1, 1),
bias=False)
          (1): BatchNorm2d(576, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
          (2): ReLU6(inplace=True)
        )
        (1): Conv2dNormActivation(
```

```
          (0): Conv2d(576, 576, kernel_size=(3, 3), stride=(1, 1),
padding=(1, 1), groups=576, bias=False)
          (1): BatchNorm2d(576, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
          (2): ReLU6(inplace=True)
        )
        (2): Conv2d(576, 96, kernel_size=(1, 1), stride=(1, 1),
bias=False)
        (3): BatchNorm2d(96, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
      )
    )
    (13): InvertedResidual(
      (conv): Sequential(
        (0): Conv2dNormActivation(
          (0): Conv2d(96, 576, kernel_size=(1, 1), stride=(1, 1),
bias=False)
          (1): BatchNorm2d(576, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
          (2): ReLU6(inplace=True)
        )
        (1): Conv2dNormActivation(
          (0): Conv2d(576, 576, kernel_size=(3, 3), stride=(1, 1),
padding=(1, 1), groups=576, bias=False)
          (1): BatchNorm2d(576, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
          (2): ReLU6(inplace=True)
        )
        (2): Conv2d(576, 96, kernel_size=(1, 1), stride=(1, 1),
bias=False)
        (3): BatchNorm2d(96, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
      )
    )
    (14): InvertedResidual(
      (conv): Sequential(
        (0): Conv2dNormActivation(
          (0): Conv2d(96, 576, kernel_size=(1, 1), stride=(1, 1),
bias=False)
          (1): BatchNorm2d(576, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
          (2): ReLU6(inplace=True)
        )
        (1): Conv2dNormActivation(
          (0): Conv2d(576, 576, kernel_size=(3, 3), stride=(2, 2),
padding=(1, 1), groups=576, bias=False)
          (1): BatchNorm2d(576, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
          (2): ReLU6(inplace=True)
```

```
        )
        (2): Conv2d(576, 160, kernel_size=(1, 1), stride=(1, 1),
bias=False)
        (3): BatchNorm2d(160, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
      )
    )
    (15): InvertedResidual(
      (conv): Sequential(
        (0): Conv2dNormActivation(
          (0): Conv2d(160, 960, kernel_size=(1, 1), stride=(1, 1),
bias=False)
          (1): BatchNorm2d(960, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
          (2): ReLU6(inplace=True)
        )
        (1): Conv2dNormActivation(
          (0): Conv2d(960, 960, kernel_size=(3, 3), stride=(1, 1),
padding=(1, 1), groups=960, bias=False)
          (1): BatchNorm2d(960, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
          (2): ReLU6(inplace=True)
        )
        (2): Conv2d(960, 160, kernel_size=(1, 1), stride=(1, 1),
bias=False)
        (3): BatchNorm2d(160, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
      )
    )
    (16): InvertedResidual(
      (conv): Sequential(
        (0): Conv2dNormActivation(
          (0): Conv2d(160, 960, kernel_size=(1, 1), stride=(1, 1),
bias=False)
          (1): BatchNorm2d(960, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
          (2): ReLU6(inplace=True)
        )
        (1): Conv2dNormActivation(
          (0): Conv2d(960, 960, kernel_size=(3, 3), stride=(1, 1),
padding=(1, 1), groups=960, bias=False)
          (1): BatchNorm2d(960, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
          (2): ReLU6(inplace=True)
        )
        (2): Conv2d(960, 160, kernel_size=(1, 1), stride=(1, 1),
bias=False)
        (3): BatchNorm2d(160, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
```

```
      )
    )
    (17): InvertedResidual(
      (conv): Sequential(
        (0): Conv2dNormActivation(
          (0): Conv2d(160, 960, kernel_size=(1, 1), stride=(1, 1),
bias=False)
          (1): BatchNorm2d(960, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
          (2): ReLU6(inplace=True)
        )
        (1): Conv2dNormActivation(
          (0): Conv2d(960, 960, kernel_size=(3, 3), stride=(1, 1),
padding=(1, 1), groups=960, bias=False)
          (1): BatchNorm2d(960, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
          (2): ReLU6(inplace=True)
        )
        (2): Conv2d(960, 320, kernel_size=(1, 1), stride=(1, 1),
bias=False)
        (3): BatchNorm2d(320, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
      )
    )
    (18): Conv2dNormActivation(
      (0): Conv2d(320, 1280, kernel_size=(1, 1), stride=(1, 1),
bias=False)
      (1): BatchNorm2d(1280, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
      (2): ReLU6(inplace=True)
    )
  )
  (classifier): Sequential(
    (0): Dropout(p=0.2, inplace=False)
    (1): Linear(in_features=1280, out_features=43, bias=True)
  )
)

# Prediction Function
def predict(model, image, device):
    image = image.to(device)
    with torch.no_grad():
        output = model(image.unsqueeze(0))  # Add batch dimension
        probabilities = torch.nn.functional.softmax(output, dim=1)
        confidence, predicted = probabilities.max(1)
        return {"predicted_class": predicted.item(), "confidence":
confidence.item()}

# VANET Classes
class NetworkInterface:
```

```
    def __init__(self, car_id):
        self.car_id = car_id
        self.rx_queue = []

    def transmit(self, message, cars, communication_range,
car_position, road_length):
        for car in cars:
            if car.id != self.car_id:
                distance = self.calculate_distance(car_position,
car.x, road_length)
                if distance <= communication_range:
                    car.network_interface.receive(message)

    def receive(self, message):
        self.rx_queue.append(message)

    def process_reception_queue(self):
        messages = self.rx_queue[:]
        self.rx_queue.clear()
        return messages

    @staticmethod
    def calculate_distance(pos1, pos2, road_length):
        return min(abs(pos1 - pos2), road_length - abs(pos1 - pos2))
```

Brightness change simulation

```
test_transforms = transforms.Compose([
    transforms.Resize((224, 224)),
    transforms.ToTensor(),
    transforms.Normalize(mean=[0.485, 0.456, 0.406], std=[0.229,
0.224, 0.225]),
])

def adjust_brightness(image):
    brightness_factor = random.uniform(0.3, 1.3)

    enhancer = ImageEnhance.Brightness(image)
    img_enhanced = enhancer.enhance(brightness_factor)

    # img_enhanced.save("/kaggle/working/img.png")
    return test_transforms(img_enhanced)

class Car:
    def __init__(self, id, x, velocity):
        self.id = id
        self.x = x
        self.velocity = velocity
        self.received_predictions = []
```

```
        self.network_interface = NetworkInterface(self.id)
        self.own_prediction = {}

    def move(self, road_length):
        self.x = (self.x + self.velocity) % road_length

    def make_prediction(self, image):
        # Convert back to tensor for prediction
        image_tensor = adjust_brightness(image)
        self.own_prediction = predict(model, image_tensor, device)


    def broadcast(self, cars, communication_range, road_length):
        message = {
            "sender_id": self.id,
            "predicted_class": self.own_prediction["predicted_class"],
            "confidence": self.own_prediction["confidence"],
        }
        self.network_interface.transmit(message, cars,
communication_range, self.x, road_length)

    def process_messages(self):
        received = self.network_interface.process_reception_queue()
        self.received_predictions.extend(received)

    def calculate_consensus(self):
        all_predictions = self.received_predictions + [
            {"predicted_class":
self.own_prediction["predicted_class"], "confidence":
self.own_prediction["confidence"]}
        ]
        # Step 1: Count occurrences of each class
        classes = [p["predicted_class"] for p in all_predictions]
        class_counts = Counter(classes)

        # Step 2: Group confidences by class
        confidences = {cls: [] for cls in classes}
        for p in all_predictions:
            confidences[p["predicted_class"]].append(p["confidence"])

        # Step 3: Find the most common class
        max_count = max(class_counts.values())
        candidates = [cls for cls, count in class_counts.items() if
count == max_count]

        # Step 4: Resolve ties by selecting the class with the highest
average confidence
        if len(candidates) > 1:
            avg_confidences = {cls: sum(confidences[cls]) /
len(confidences[cls]) for cls in candidates}
```

```
            most_common_class = max(avg_confidences,
key=avg_confidences.get)
        else:
            most_common_class = candidates[0]

        # Step 5: Calculate average confidence for the selected class
        avg_confidence = sum(confidences[most_common_class]) /
len(confidences[most_common_class])

        return {"predicted_class": most_common_class, "confidence":
avg_confidence}

# Simulation Parameters
num_cars = 10
road_length = 100
communication_range = 20
time_steps = 50
results = []

# Initialize Cars
cars = [Car(id=i, x=random.uniform(0, road_length),
velocity=random.uniform(1, 5)) for i in range(num_cars)]

# Simulation Loop
for idx, (image_tensor, true_label, img_path) in
enumerate(tqdm(test_dataset, desc="Processing Images", total=12630)):
    total_messages = 0
    all_received = False

    # if idx >= 10:
    #     break

    # print(f"Image {idx + 1}/{len(test_dataset)}: {img_path}")

    car_predictions = []
    for car in cars:
        car.received_predictions.clear()
        car.network_interface.rx_queue.clear()
        car.make_prediction(image_tensor)
        car_predictions.append({
            "car_id": car.id,
            "predicted_class": car.own_prediction["predicted_class"],
            "confidence": car.own_prediction["confidence"]
        })

    # # Print each car's prediction for the current image
    # for prediction in car_predictions:
    #     # print(f"Car {prediction['car_id']} predicted class
{prediction['predicted_class']} with confidence
{prediction['confidence']:.2f}")
```

```python
    for t in range(time_steps):
        for car in cars:
            car.move(road_length)
        for car in cars:
            car.broadcast(cars, communication_range, road_length)
        for car in cars:
            car.process_messages()
        if all(len(car.received_predictions) + 1 == len(cars) for car in cars):
            all_received = True
            break

    consensus_results = [car.calculate_consensus() for car in cars]
    most_common_class = Counter([c["predicted_class"] for c in consensus_results]).most_common(1)[0][0]
    avg_confidence = sum(c["confidence"] for c in consensus_results if c["predicted_class"] == most_common_class) / len(
        [c for c in consensus_results if c["predicted_class"] == most_common_class]
    )

    results.append({
        "image_id": idx,
        "image_path": img_path,
        "true_label": true_label,
        "total_messages": sum(len(car.received_predictions) for car in cars),
        "all_received": all_received,
        "time_steps": t + 1,
        "final_predicted_class": most_common_class,
        "final_confidence": avg_confidence,
    })
```

```
Processing Images: 100%|██████████| 12630/12630 [14:47<00:00, 14.23it/s]
```

```python
# Save Results
results_df = pd.DataFrame(results)
results_df.to_csv("vanet_simulation_results.csv", index=False)

# Accuracy Calculation
correct_predictions = results_df["true_label"] == results_df["final_predicted_class"]
accuracy = 100.0 * correct_predictions.sum() / len(correct_predictions)
print(f"System Accuracy: {accuracy:.2f}%")
```

```
System Accuracy: 95.46%
```

So basically the brightness did not make any difference at all. it was expected because it barely affectdd the single-use model as well. so in your thesis, write down why your model is so resistent to lighting changes. it's a great finding.

Motion blur simulation scenario

```
from PIL import Image, ImageFilter

def adjust_motion_blur(image):
    blur_radius = random.uniform(0.5, 3.0)

    img_blurred = image.filter(ImageFilter.GaussianBlur(blur_radius))

    # img_enhanced.save("/kaggle/working/img.png")
    return test_transforms(img_blurred)

class Car:
    def __init__(self, id, x, velocity):
        self.id = id
        self.x = x
        self.velocity = velocity
        self.received_predictions = []
        self.network_interface = NetworkInterface(self.id)
        self.own_prediction = {}

    def move(self, road_length):
        self.x = (self.x + self.velocity) % road_length

    def make_prediction(self, image):
        # Convert back to tensor for prediction
        image_tensor = adjust_motion_blur(image)
        self.own_prediction = predict(model, image_tensor, device)


    def broadcast(self, cars, communication_range, road_length):
        message = {
            "sender_id": self.id,
            "predicted_class": self.own_prediction["predicted_class"],
            "confidence": self.own_prediction["confidence"],
        }
        self.network_interface.transmit(message, cars,
communication_range, self.x, road_length)

    def process_messages(self):
        received = self.network_interface.process_reception_queue()
        self.received_predictions.extend(received)

    def calculate_consensus(self):
        all_predictions = self.received_predictions + [
```

```
            {"predicted_class":
self.own_prediction["predicted_class"], "confidence":
self.own_prediction["confidence"]}
        ]
        # Step 1: Count occurrences of each class
        classes = [p["predicted_class"] for p in all_predictions]
        class_counts = Counter(classes)

        # Step 2: Group confidences by class
        confidences = {cls: [] for cls in classes}
        for p in all_predictions:
            confidences[p["predicted_class"]].append(p["confidence"])

        # Step 3: Find the most common class
        max_count = max(class_counts.values())
        candidates = [cls for cls, count in class_counts.items() if
count == max_count]

        # Step 4: Resolve ties by selecting the class with the highest
average confidence
        if len(candidates) > 1:
            avg_confidences = {cls: sum(confidences[cls]) /
len(confidences[cls]) for cls in candidates}
            most_common_class = max(avg_confidences,
key=avg_confidences.get)
        else:
            most_common_class = candidates[0]

        # Step 5: Calculate average confidence for the selected class
        avg_confidence = sum(confidences[most_common_class]) /
len(confidences[most_common_class])

        return {"predicted_class": most_common_class, "confidence":
avg_confidence}

# Simulation Parameters
num_cars = 10
road_length = 100
communication_range = 20
time_steps = 50
results = []

# Initialize Cars
cars = [Car(id=i, x=random.uniform(0, road_length),
velocity=random.uniform(1, 5)) for i in range(num_cars)]

# Simulation Loop
for idx, (image_tensor, true_label, img_path) in
enumerate(tqdm(test_dataset, desc="Processing Images", total=12630)):
    total_messages = 0
```

```
    all_received = False

    # if idx >= 10:
    #     break

    # print(f"Image {idx + 1}/{len(test_dataset)}: {img_path}")

    car_predictions = []
    for car in cars:
        car.received_predictions.clear()
        car.network_interface.rx_queue.clear()
        car.make_prediction(image_tensor)
        car_predictions.append({
            "car_id": car.id,
            "predicted_class": car.own_prediction["predicted_class"],
            "confidence": car.own_prediction["confidence"]
        })

    # # Print each car's prediction for the current image
    # for prediction in car_predictions:
    #     # print(f"Car {prediction['car_id']} predicted class
{prediction['predicted_class']} with confidence
{prediction['confidence']:.2f}")

    for t in range(time_steps):
        for car in cars:
            car.move(road_length)
        for car in cars:
            car.broadcast(cars, communication_range, road_length)
        for car in cars:
            car.process_messages()
        if all(len(car.received_predictions) + 1 == len(cars) for car
in cars):
            all_received = True
            break

    consensus_results = [car.calculate_consensus() for car in cars]
    most_common_class = Counter([c["predicted_class"] for c in
consensus_results]).most_common(1)[0][0]
    avg_confidence = sum(c["confidence"] for c in consensus_results if
c["predicted_class"] == most_common_class) / len(
        [c for c in consensus_results if c["predicted_class"] ==
most_common_class]
    )

    results.append({
        "image_id": idx,
        "image_path": img_path,
        "true_label": true_label,
        "total_messages": sum(len(car.received_predictions) for car in
```

```
cars),
        "all_received": all_received,
        "time_steps": t + 1,
        "final_predicted_class": most_common_class,
        "final_confidence": avg_confidence,
    })
```

```
Processing Images: 100%|██████████| 12630/12630 [15:55<00:00,
13.22it/s]
```

```
# Save Results
results_df = pd.DataFrame(results)
results_df.to_csv("vanet_simulation_results.csv", index=False)

# Accuracy Calculation
correct_predictions = results_df["true_label"] ==
results_df["final_predicted_class"]
accuracy = 100.0 * correct_predictions.sum() /
len(correct_predictions)
print(f"System Accuracy: {accuracy:.2f}%")
```

```
System Accuracy: 66.37%
```

slight performance boost. write it in your thesis. it's worthwhile

Angle and rotation

```
from PIL import Image

def adjust_rotation(image):
    random_angle = random.uniform(-45, 45)

    rotated_image = image.rotate(random_angle, resample=Image.BICUBIC,
expand=True)

    # img_enhanced.save("/kaggle/working/img.png")
    return test_transforms(rotated_image)

class Car:
    def __init__(self, id, x, velocity):
        self.id = id
        self.x = x
        self.velocity = velocity
        self.received_predictions = []
        self.network_interface = NetworkInterface(self.id)
        self.own_prediction = {}

    def move(self, road_length):
        self.x = (self.x + self.velocity) % road_length
```

```
    def make_prediction(self, image):
        # Convert back to tensor for prediction
        image_tensor = adjust_rotation(image)
        self.own_prediction = predict(model, image_tensor, device)


    def broadcast(self, cars, communication_range, road_length):
        message = {
            "sender_id": self.id,
            "predicted_class": self.own_prediction["predicted_class"],
            "confidence": self.own_prediction["confidence"],
        }
        self.network_interface.transmit(message, cars,
communication_range, self.x, road_length)

    def process_messages(self):
        received = self.network_interface.process_reception_queue()
        self.received_predictions.extend(received)

    def calculate_consensus(self):
        all_predictions = self.received_predictions + [
            {"predicted_class":
self.own_prediction["predicted_class"], "confidence":
self.own_prediction["confidence"]}
        ]
        # Step 1: Count occurrences of each class
        classes = [p["predicted_class"] for p in all_predictions]
        class_counts = Counter(classes)

        # Step 2: Group confidences by class
        confidences = {cls: [] for cls in classes}
        for p in all_predictions:
            confidences[p["predicted_class"]].append(p["confidence"])

        # Step 3: Find the most common class
        max_count = max(class_counts.values())
        candidates = [cls for cls, count in class_counts.items() if
count == max_count]

        # Step 4: Resolve ties by selecting the class with the highest
average confidence
        if len(candidates) > 1:
            avg_confidences = {cls: sum(confidences[cls]) /
len(confidences[cls]) for cls in candidates}
            most_common_class = max(avg_confidences,
key=avg_confidences.get)
        else:
            most_common_class = candidates[0]
```

```
        # Step 5: Calculate average confidence for the selected class
        avg_confidence = sum(confidences[most_common_class]) /
len(confidences[most_common_class])

        return {"predicted_class": most_common_class, "confidence":
avg_confidence}

# Simulation Parameters
num_cars = 10
road_length = 100
communication_range = 20
time_steps = 50
results = []

# Initialize Cars
cars = [Car(id=i, x=random.uniform(0, road_length),
velocity=random.uniform(1, 5)) for i in range(num_cars)]

# Simulation Loop
for idx, (image_tensor, true_label, img_path) in
enumerate(tqdm(test_dataset, desc="Processing Images", total=12630)):
    total_messages = 0
    all_received = False

    # if idx >= 10:
    #     break

    # print(f"Image {idx + 1}/{len(test_dataset)}: {img_path}")

    car_predictions = []
    for car in cars:
        car.received_predictions.clear()
        car.network_interface.rx_queue.clear()
        car.make_prediction(image_tensor)
        car_predictions.append({
            "car_id": car.id,
            "predicted_class": car.own_prediction["predicted_class"],
            "confidence": car.own_prediction["confidence"]
        })

    # # Print each car's prediction for the current image
    # for prediction in car_predictions:
    #     # print(f"Car {prediction['car_id']} predicted class
{prediction['predicted_class']} with confidence
{prediction['confidence']:.2f}")

    for t in range(time_steps):
        for car in cars:
            car.move(road_length)
        for car in cars:
```

```
            car.broadcast(cars, communication_range, road_length)
        for car in cars:
            car.process_messages()
        if all(len(car.received_predictions) + 1 == len(cars) for car
in cars):
            all_received = True
            break

    consensus_results = [car.calculate_consensus() for car in cars]
    most_common_class = Counter([c["predicted_class"] for c in
consensus_results]).most_common(1)[0][0]
    avg_confidence = sum(c["confidence"] for c in consensus_results if
c["predicted_class"] == most_common_class) / len(
        [c for c in consensus_results if c["predicted_class"] ==
most_common_class]
    )

    results.append({
        "image_id": idx,
        "image_path": img_path,
        "true_label": true_label,
        "total_messages": sum(len(car.received_predictions) for car in
cars),
        "all_received": all_received,
        "time_steps": t + 1,
        "final_predicted_class": most_common_class,
        "final_confidence": avg_confidence,
    })
```

```
Processing Images: 100%|██████████| 12630/12630 [15:26<00:00,
13.63it/s]
```

```
# Save Results
results_df = pd.DataFrame(results)
results_df.to_csv("vanet_simulation_results.csv", index=False)

# Accuracy Calculation
correct_predictions = results_df["true_label"] ==
results_df["final_predicted_class"]
accuracy = 100.0 * correct_predictions.sum() /
len(correct_predictions)
print(f"System Accuracy: {accuracy:.2f}%")
```

```
System Accuracy: 77.66%
```

```
results_df[10:20]
```

```
    image_id                                          image_path
true_label  \
10        10  /kaggle/input/gtsrb-german-traffic-sign/Test/0...
12
```

```
11         11  /kaggle/input/gtsrb-german-traffic-sign/Test/0...
7
12         12  /kaggle/input/gtsrb-german-traffic-sign/Test/0...
23
13         13  /kaggle/input/gtsrb-german-traffic-sign/Test/0...
7
14         14  /kaggle/input/gtsrb-german-traffic-sign/Test/0...
4
15         15  /kaggle/input/gtsrb-german-traffic-sign/Test/0...
9
16         16  /kaggle/input/gtsrb-german-traffic-sign/Test/0...
21
17         17  /kaggle/input/gtsrb-german-traffic-sign/Test/0...
20
18         18  /kaggle/input/gtsrb-german-traffic-sign/Test/0...
27
19         19  /kaggle/input/gtsrb-german-traffic-sign/Test/0...
38

    total_messages  all_received  time_steps  final_predicted_class  \
10            1884         False          50                     12
11            2012         False          50                      7
12            1604         False          50                     23
13            1866         False          50                      7
14            1664         False          50                      4
15            1548         False          50                     32
16            2270         False          50                     20
17            1862         False          50                     20
18            2060         False          50                     27
19            1884         False          50                     38

    final_confidence
10          0.769514
11          0.972472
12          0.718060
13          0.994168
14          0.991494
15          0.720205
16          0.235127
17          0.746733
18          0.988319
19          0.874092
```

it's good. let's see how the model itself performs alone.