ДОДАТОК А

Програмна реалізація базової моделі Randomized Smoothing

```
Реалізація базового методу Randomized Smoothing для сертифікованої стійкості.
Автор: Рижова Олександра
Дата: 02.11.2025
"""

import torch
import torch.nn as nn
import torch.nn.functional as F
import numpy as np
from scipy.stats import norm
from typing import Tuple, List, Optional
import time

class BaseRandomizedSmoothing:
    """
    Базовий клас для Randomized Smoothing з гаусівським шумом.
    """

    def __init__(self, base_classifier: nn.Module, sigma: float = 0.25,
                 num_samples: int = 1000, batch_size: int = 256,
                 device: str = 'cuda' if torch.cuda.is_available() else 'cpu'):
        """
        Ініціалізація параметрів згладжування.

        Args:
            base_classifier: Базова модель класифікатора
```

```
            sigma: Стандартне відхилення гаусівського шуму
            num_samples: Кількість шумових вибірок для оцінки
            batch_size: Розмір батча для паралельної обробки
            device: Пристрій для обчислень (cuda/cpu)
        """
        self.base_classifier = base_classifier
        self.sigma = sigma
        self.num_samples = num_samples
        self.batch_size = batch_size
        self.device = device
        self.base_classifier.to(device)
        self.base_classifier.eval()

        # Статистичні константи
        self.alpha = 0.001  # Рівень значущості для довірчого інтервалу
        self.z_alpha = norm.ppf(1 - self.alpha)  # Квантиль нормального розподілу

    def certify(self, x: torch.Tensor, n0: int = 100, n: int = 100000,
                abort_epsilon: float = 0.001) -> Tuple[int, float]:
        """
        Сертифікація одного вхідного зразка.

        Args:
            x: Вхідний зразок [C, H, W]
            n0: Кількість вибірок для початкової перевірки
            n: Максимальна кількість вибірок
            abort_epsilon: Поріг для припинення обчислень

        Returns:
```

```
    predicted_class: Передбачений клас
    radius: Сертифікований радіус стійкості
"""
# Перевірка вхідних даних
if len(x.shape) == 3:
    x = x.unsqueeze(0)  # [1, C, H, W]

# Крок 1: Пошук головного класу з n0 вибірками
counts_estimation = self._sample_noise_predictions(x, n0)
p_a = counts_estimation.max().item() / n0
predicted_class = counts_estimation.argmax().item()

# Якщо ймовірність занадто низька, повертаємо відмову
if p_a < 0.5:
    return predicted_class, 0.0

# Крок 2: Уточнена оцінка ймовірності головного класу
counts_refined = self._sample_noise_predictions(x, n)
n_a = counts_refined[predicted_class].item()
p_a_hat = n_a / n

# Обчислення нижньої межі Clopper-Pearson для p_a
p_a_lower = self._clopper_pearson_lower_bound(n_a, n, self.alpha)

# Перевірка умови для сертифікації
if p_a_lower < 0.5:
    return predicted_class, 0.0

# Крок 3: Пошук другого за ймовірністю класу
counts_refined[predicted_class] = 0
```

```
            second_class = counts_refined.argmax().item()
            n_b = counts_refined[second_class].item()
            p_b_hat = n_b / n

            # Обчислення верхньої межі для p_b
            p_b_upper = self._clopper_pearson_upper_bound(n_b, n, self.alpha)

            # Обчислення сертифікованого радіуса
            if p_a_lower > p_b_upper:
                radius = self.sigma * norm.ppf(p_a_lower) - norm.ppf(p_b_upper)
                radius = max(0.0, radius)
            else:
                radius = 0.0

            return predicted_class, radius

        def predict(self, x: torch.Tensor) -> torch.Tensor:
            """
            Згладжена класифікація для батча зразків.

            Args:
                x: Батч вхідних зразків [B, C, H, W]

            Returns:
                predictions: Прогнози для кожного зразка [B]
            """
            batch_size = x.shape[0]
            predictions = torch.zeros(batch_size, dtype=torch.long,
device=self.device)
```

```
            for i in range(batch_size):
                x_i = x[i:i+1]
                pred_class, _ = self.certify(x_i, n0=100, n=1000)
                predictions[i] = pred_class

            return predictions

        def _sample_noise_predictions(self, x: torch.Tensor, num_samples: int)
-> torch.Tensor:
            """
            Генерація шумових вибірок та підрахунок прогнозів.

            Args:
                x: Вхідний зразок [1, C, H, W]
                num_samples: Кількість шумових вибірок

            Returns:
                counts: Тензор з кількістю прогнозів для кожного класу
            """
            num_classes = 10  # Для CIFAR-10

            # Обчислення кількості батчів
            num_batches = int(np.ceil(num_samples / self.batch_size))
            counts = torch.zeros(num_classes, dtype=torch.long,
device=self.device)

            with torch.no_grad():
                for batch_idx in range(num_batches):
                    # Визначення розміру поточного батча
                    current_batch_size = min(self.batch_size,
```

```
                                num_samples - batch_idx * self.batch_size)

                if current_batch_size <= 0:
                    break

                # Створення батча з однаковими зображеннями
                x_batch = x.repeat(current_batch_size, 1, 1, 1)

                # Додавання гаусівського шуму
                noise = torch.randn_like(x_batch) * self.sigma
                x_noisy = x_batch + noise

                # Класифікація
                outputs = self.base_classifier(x_noisy)
                preds = outputs.argmax(dim=1)

                # Підрахунок прогнозів
                for pred in preds:
                    counts[pred] += 1

            return counts

        @staticmethod
        def _clopper_pearson_lower_bound(k: int, n: int, alpha: float) -> float:
            """
            Обчислення нижньої межі Clopper-Pearson для біноміального розподілу.
            """
            if k == 0:
                return 0.0
```

```
        if k == n:
            return (alpha) ** (1/n)

        from scipy.stats import beta
        return beta.ppf(alpha, k, n - k + 1)

    @staticmethod
    def _clopper_pearson_upper_bound(k: int, n: int, alpha: float) -> float:
        """
        Обчислення верхньої межі Clopper-Pearson для біноміального розподілу.
        """
        if k == 0:
            return 1 - (alpha) ** (1/n)
        if k == n:
            return 1.0

        from scipy.stats import beta
        return beta.ppf(1 - alpha, k + 1, n - k)

    def evaluate_certified_accuracy(self, test_loader, radii: List[float]) -> List[float]:
        """
        Оцінка сертифікованої точності для різних радіусів.

        Args:
            test_loader: DataLoader тестового датасету
            radii: Список радіусів для оцінки

        Returns:
```

```
    certified_accuracies: Список сертифікованих точностей
"""
certified_accuracies = []
total_samples = 0
correct_counts = {r: 0 for r in radii}

self.base_classifier.eval()

with torch.no_grad():
    for batch_idx, (images, labels) in enumerate(test_loader):
        images, labels = images.to(self.device), labels.to(self.device)
        batch_size = images.shape[0]

        for i in range(batch_size):
            x = images[i:i+1]
            y_true = labels[i].item()

            # Сертифікація зразка
            pred_class, certified_radius = self.certify(x)

            # Перевірка для кожного радіуса
            for radius in radii:
                if pred_class == y_true and certified_radius >= radius:
                    correct_counts[radius] += 1

            total_samples += 1

            # Прогрес
            if (i + batch_idx * batch_size) % 100 == 0:
                print(f"Оброблено {i + batch_idx * batch_size} зразків")
```

```
        # Обчислення сертифікованої точності
        for radius in radii:
            accuracy = correct_counts[radius] / total_samples
            certified_accuracies.append(accuracy)
            print(f"Сертифікована точність при radius={radius}: {accuracy:.4f}")

        return certified_accuracies
```

ДОДАТОК Б

Реалізація адаптивного Randomized Smoothing з нейромережним адаптером

```
"""
Реалізація Adaptive Randomized Smoothing з нейромережним шумовим адаптером.
Автор: Рижова Олександра
Дата: 10.11.2025
"""

import torch
import torch.nn as nn
import torch.nn.functional as F
from typing import Tuple, Optional, List
import numpy as np

class NoiseAdapter(nn.Module):
    """
    Нейромережний адаптер для прогнозування параметрів шуму.
    Приймає латентне представлення та виводить параметри шуму.
    """

    def __init__(self, input_dim: int = 512, hidden_dims: List[int] = [256, 128],
                 output_dim: int = 1, dropout_rate: float = 0.2):
        """
        Ініціалізація архітектури адаптера.

        Args:
            input_dim: Розмірність вхідного латентного вектора
```

```
        hidden_dims: Список розмірностей прихованих шарів
        output_dim: Розмірність вихідних параметрів шуму
        dropout_rate: Рівень dropout для регуляризації
    """
    super(NoiseAdapter, self).__init__()

    # Створення шару нормалізації для стабілізації навчання
    self.normalize = nn.LayerNorm(input_dim)

    # Побудова прихованих шарів
    layers = []
    prev_dim = input_dim

    for hidden_dim in hidden_dims:
        layers.extend([
            nn.Linear(prev_dim, hidden_dim),
            nn.BatchNorm1d(hidden_dim),
            nn.ReLU(inplace=True),
            nn.Dropout(dropout_rate)
        ])
        prev_dim = hidden_dim

    # Вихідний шар
    layers.append(nn.Linear(prev_dim, output_dim))
    layers.append(nn.Softplus())  # Гарантує позитивність виходу

    self.mlp = nn.Sequential(*layers)

def forward(self, z: torch.Tensor) -> torch.Tensor:
    """
```

```
        Прямий прохід адаптера.

        Args:
            z: Латентне представлення [batch_size, input_dim]

        Returns:
            sigma: Прогнозовані параметри шуму [batch_size, output_dim]
        """
        z_norm = self.normalize(z)
        sigma = self.mlp(z_norm)
        return sigma

class AdaptiveRandomizedSmoothing(nn.Module):
    """
    Adaptive Randomized Smoothing з нейромережним адаптером.
    """

    def __init__(self, base_classifier: nn.Module, adapter: NoiseAdapter,
                 sigma_min: float = 0.1, sigma_max: float = 1.0,
                 num_samples: int = 1000, device: str = 'cuda'):
        """
        Ініціалізація адаптивного згладжування.

        Args:
            base_classifier: Базова модель класифікатора
            adapter: Шумовий адаптер
            sigma_min: Мінімальне значення шуму
            sigma_max: Максимальне значення шуму
            num_samples: Кількість шумових вибірок
            device: Пристрій для обчислень
```

```
    """
    super(AdaptiveRandomizedSmoothing, self).__init__()

    self.base_classifier = base_classifier
    self.adapter = adapter
    self.sigma_min = sigma_min
    self.sigma_max = sigma_max
    self.num_samples = num_samples
    self.device = device

    # Перенесення моделей на пристрій
    self.base_classifier.to(device)
    self.adapter.to(device)

    # Режим оцінки за замовчуванням
    self.eval()

def forward(self, x: torch.Tensor) -> torch.Tensor:
    """
    Прямий прохід для тренування.

    Args:
        x: Вхідні зображення [batch_size, C, H, W]

    Returns:
        logits: Логіти для кожного класу [batch_size, num_classes]
    """
    batch_size = x.shape[0]

    # Отримання латентного представлення
```

```
        with torch.no_grad():
            # Для ResNet отримуємо вихід перед останнім шаром
            z = self._get_latent_representation(x)

        # Прогнозування параметрів шуму
        sigma_pred = self.adapter(z)

        # Обмеження значень sigma
        sigma = torch.clamp(sigma_pred, self.sigma_min, self.sigma_max)

        # Генерація шуму та згладжена класифікація
        logits_sum = torch.zeros(batch_size, 10, device=self.device)  # Для
CIFAR-10

        for _ in range(self.num_samples):
            # Додавання адаптивного шуму
            noise = torch.randn_like(x)
            # Масштабування шуму для кожного зразка окремо
            noise_scaled = noise * sigma.view(-1, 1, 1, 1)
            x_noisy = x + noise_scaled

            # Класифікація
            logits = self.base_classifier(x_noisy)
            logits_sum += F.softmax(logits, dim=1)

        # Усереднення логітів
        avg_logits = logits_sum / self.num_samples

        return avg_logits
```

```
    def _get_latent_representation(self, x: torch.Tensor) -> torch.Tensor:
        """
        Отримання латентного представлення з базового класифікатора.
        Для ResNet отримуємо вихід перед останнім повнозв'язаним шаром.

        Args:
            x: Вхідні зображення

        Returns:
            z: Латентні представлення [batch_size, latent_dim]
        """
        # Хук для отримання проміжного представлення
        features = None

        def hook_fn(module, input, output):
            nonlocal features
            features = output

        # Реєстрація хука на відповідному шарі
        if hasattr(self.base_classifier, 'layer4'):
            # Для стандартного ResNet
            handle = self.base_classifier.layer4.register_forward_hook(hook_fn)
        elif hasattr(self.base_classifier, 'features'):
            # Для інших архітектур
            handle = self.base_classifier.features.register_forward_hook(hook_fn)
        else:
            # Запасний варіант: використання глобального average pooling
```

```
            features = self.base_classifier(x)
            features = features.view(features.size(0), -1)
            return features

        # Виконання прямого проходу
        _ = self.base_classifier(x)

        # Видалення хука
        handle.remove()

        # Обробка отриманих ознак
        if features is not None:
            features = F.adaptive_avg_pool2d(features, (1, 1))
            features = features.view(features.size(0), -1)

        return features

    def certify_adaptive(self, x: torch.Tensor, n0: int = 100, n: int = 10000)
-> Tuple[int, float]:
        """
        Адаптивна сертифікація з динамічним σ(x).

        Args:
            x: Вхідний зразок [1, C, H, W]
            n0: Початкова кількість вибірок
            n: Повна кількість вибірок

        Returns:
            predicted_class: Передбачений клас
            radius: Сертифікований радіус
```

```
        """
        # Отримання адаптивного σ для даного зразка
        with torch.no_grad():
            z = self._get_latent_representation(x)
            sigma_pred = self.adapter(z)
            sigma = torch.clamp(sigma_pred, self.sigma_min, self.sigma_max).item()

        # Використання базової сертифікації з адаптивним σ
        base_rs = BaseRandomizedSmoothing(
            base_classifier=self.base_classifier,
            sigma=sigma,
            num_samples=self.num_samples,
            device=self.device
        )

        return base_rs.certify(x, n0, n)

    def evaluate_adaptive(self, test_loader, radii: List[float]) -> Tuple[List[float], List[float]]:
        """
        Оцінка адаптивного згладжування.

        Args:
            test_loader: DataLoader тестового датасету
            radii: Список радіусів для оцінки

        Returns:
            clean_accuracies: Чисті точності
            certified_accuracies: Сертифіковані точності
```

```
"""
self.eval()

clean_correct = 0
certified_correct = {r: 0 for r in radii}
total_samples = 0

with torch.no_grad():
    for batch_idx, (images, labels) in enumerate(test_loader):
        images, labels = images.to(self.device), labels.to(self.device)
        batch_size = images.shape[0]

        # Чисті прогнози (без шуму)
        clean_outputs = self.base_classifier(images)
        clean_preds = clean_outputs.argmax(dim=1)
        clean_correct += (clean_preds == labels).sum().item()

        # Адаптивна сертифікація для кожного зразка
        for i in range(batch_size):
            x = images[i:i+1]
            y_true = labels[i].item()

            pred_class, radius = self.certify_adaptive(x)

            # Перевірка для кожного радіуса
            for r in radii:
                if pred_class == y_true and radius >= r:
                    certified_correct[r] += 1

            total_samples += 1
```

```
                # Прогрес
                if batch_idx % 10 == 0:
                    print(f"Batch {batch_idx}: оброблено {batch_idx * batch_size} зразків")

            # Обчислення точностей
            clean_accuracy = clean_correct / total_samples
            certified_accuracies = [certified_correct[r] / total_samples for r in radii]

            print(f"Чиста точність: {clean_accuracy:.4f}")
            for r, acc in zip(radii, certified_accuracies):
                print(f"Сертифікована точність при radius={r}: {acc:.4f}")

            return clean_accuracy, certified_accuracies
```

ДОДАТОК В

Реалізація структурованих шумових моделей

```
"""
Приклад використання запропонованих методів для навчання та оцінки.
Автор: Рижова Олександра
Дата: 16.11.2025
"""

import torch
import torch.nn as nn
import torch.optim as optim
from torchvision import datasets, transforms, models
from torch.utils.data import DataLoader
import matplotlib.pyplot as plt
import numpy as np
from tqdm import tqdm

from base_randomized_smoothing import BaseRandomizedSmoothing
from adaptive_randomized_smoothing import AdaptiveRandomizedSmoothing, NoiseAdapter
from structured_noise_models import StructuredNoiseRS

def load_cifar10_data(batch_size: int = 128):
    """
    Завантаження та підготовка даних CIFAR-10.

    Args:
        batch_size: Розмір батча
```

```
    Returns:
        train_loader, test_loader: DataLoader для тренувальних та тестових
даних
    """
    # Трансформації для даних
    train_transform = transforms.Compose([
        transforms.RandomHorizontalFlip(),
        transforms.RandomCrop(32, padding=4),
        transforms.ToTensor(),
        transforms.Normalize((0.4914, 0.4822, 0.4465), (0.2023, 0.1994,
0.2010)),
    ])

    test_transform = transforms.Compose([
        transforms.ToTensor(),
        transforms.Normalize((0.4914, 0.4822, 0.4465), (0.2023, 0.1994,
0.2010)),
    ])

    # Завантаження датасетів
    train_dataset = datasets.CIFAR10(root='./data', train=True,
                                     download=True, transform=train_transform)
    test_dataset = datasets.CIFAR10(root='./data', train=False,
                                    download=True, transform=test_transform)

    # Створення DataLoader
    train_loader = DataLoader(train_dataset, batch_size=batch_size,
                              shuffle=True, num_workers=2)
    test_loader = DataLoader(test_dataset, batch_size=batch_size,
```

```
                shuffle=False, num_workers=2)

        return train_loader, test_loader

    def train_base_classifier(model, train_loader, test_loader, epochs: int =
100):
        """
        Навчання базового класифікатора.

        Args:
            model: Модель для навчання
            train_loader: DataLoader тренувальних даних
            test_loader: DataLoader тестових даних
            epochs: Кількість епох

        Returns:
            model: Навчена модель
        """
        device = torch.device('cuda' if torch.cuda.is_available() else 'cpu')
        model.to(device)

        criterion = nn.CrossEntropyLoss()
        optimizer = optim.SGD(model.parameters(), lr=0.1,
                              momentum=0.9, weight_decay=5e-4)
        scheduler = optim.lr_scheduler.CosineAnnealingLR(optimizer,
T_max=epochs)

        best_acc = 0.0

        for epoch in range(epochs):
```

```
# Тренування
model.train()
train_loss = 0.0
train_correct = 0
train_total = 0

pbar = tqdm(train_loader, desc=f'Epoch {epoch+1}/{epochs}')
for batch_idx, (inputs, targets) in enumerate(pbar):
    inputs, targets = inputs.to(device), targets.to(device)

    optimizer.zero_grad()
    outputs = model(inputs)
    loss = criterion(outputs, targets)
    loss.backward()
    optimizer.step()

    train_loss += loss.item()
    _, predicted = outputs.max(1)
    train_total += targets.size(0)
    train_correct += predicted.eq(targets).sum().item()

    pbar.set_postfix({
        'Loss': f'{train_loss/(batch_idx+1):.4f}',
        'Acc': f'{100.*train_correct/train_total:.2f}%'
    })

# Тестування
model.eval()
test_correct = 0
test_total = 0
```

```
            with torch.no_grad():
                for inputs, targets in test_loader:
                    inputs, targets = inputs.to(device), targets.to(device)
                    outputs = model(inputs)
                    _, predicted = outputs.max(1)
                    test_total += targets.size(0)
                    test_correct += predicted.eq(targets).sum().item()

            test_acc = 100. * test_correct / test_total

            print(f'Epoch {epoch+1}: Test Accuracy = {test_acc:.2f}%')

            # Збереження найкращої моделі
            if test_acc > best_acc:
                best_acc = test_acc
                torch.save(model.state_dict(), 'best_model.pth')

            scheduler.step()

        print(f'Найкраща точність: {best_acc:.2f}%')

        # Завантаження найкращої моделі
        model.load_state_dict(torch.load('best_model.pth'))

        return model

    def train_noise_adapter(base_model, adapter, train_loader, test_loader,
epochs: int = 50):
        """
```

```
    Навчання шумового адаптера.

    Args:
        base_model: Базова модель класифікатора
        adapter: Шумовий адаптер
        train_loader: DataLoader тренувальних даних
        test_loader: DataLoader тестових даних
        epochs: Кількість епох

    Returns:
        adapter: Навчений адаптер
    """
    device = torch.device('cuda' if torch.cuda.is_available() else 'cpu')
    base_model.to(device)
    adapter.to(device)

    # Заморожування базової моделі
    for param in base_model.parameters():
        param.requires_grad = False

    # Комбінована функція втрат
    classification_criterion = nn.CrossEntropyLoss()
    radius_criterion = nn.MSELoss()  # Для максимізації радіуса

    optimizer = optim.Adam(adapter.parameters(), lr=1e-3)
    scheduler = optim.lr_scheduler.StepLR(optimizer, step_size=20,
gamma=0.1)

    for epoch in range(epochs):
        adapter.train()
```

```
        train_loss = 0.0

        pbar = tqdm(train_loader, desc=f'Training Adapter Epoch {epoch+1}/{epochs}')
        for batch_idx, (inputs, targets) in enumerate(pbar):
            inputs, targets = inputs.to(device), targets.to(device)

            optimizer.zero_grad()

            # Отримання латентних представлень
            with torch.no_grad():
                # Отримання ознак з базової моделі
                features = base_model(inputs)
                features = features.view(features.size(0), -1)

            # Прогнозування параметрів шуму
            sigma_pred = adapter(features)

            # Генерація адаптивного шуму
            noise = torch.randn_like(inputs) * sigma_pred.view(-1, 1, 1, 1)
            inputs_noisy = inputs + noise

            # Класифікація зашумлених даних
            outputs = base_model(inputs_noisy)

            # Обчислення втрат
            loss_classification = classification_criterion(outputs, targets)

            # Додатковий штраф для максимізації сертифікованого радіуса
            # Мета: σ повинно бути достатнім, але не надмірним
```

```
        target_sigma = torch.ones_like(sigma_pred) * 0.3
        loss_radius = radius_criterion(sigma_pred, target_sigma)

        # Сумарна втрата
        loss = loss_classification + 0.1 * loss_radius

        loss.backward()
        optimizer.step()

        train_loss += loss.item()
        pbar.set_postfix({'Loss': f'{train_loss/(batch_idx+1):.4f}'})

    # Оцінка адаптера
    adapter.eval()
    test_correct = 0
    test_total = 0

    with torch.no_grad():
        for inputs, targets in test_loader:
            inputs, targets = inputs.to(device), targets.to(device)

            # Отримання ознак
            features = base_model(inputs)
            features = features.view(features.size(0), -1)

            # Прогнозування σ
            sigma_pred = adapter(features)
            sigma_pred = torch.clamp(sigma_pred, 0.1, 1.0)

            # Адаптивне згладжування (середнє за 10 вибірок)
```

```
                outputs_sum = torch.zeros(inputs.size(0), 10, device=device)

                for _ in range(10):
                    noise = torch.randn_like(inputs) * sigma_pred.view(-1, 1, 1, 1)
                    outputs = base_model(inputs + noise)
                    outputs_sum += torch.softmax(outputs, dim=1)

                avg_outputs = outputs_sum / 10
                _, predicted = avg_outputs.max(1)

                test_total += targets.size(0)
                test_correct += predicted.eq(targets).sum().item()

        test_acc = 100. * test_correct / test_total
        print(f'Epoch {epoch+1}: Adapter Test Accuracy = {test_acc:.2f}%')

        scheduler.step()

    return adapter

def main():
    """Головна функція для запуску експериментів."""
    print("=" * 60)
    print("ЕКСПЕРИМЕНТИ З RANDOMIZED SMOOTHING")
    print("=" * 60)

    # Завантаження даних
    print("\n1. Завантаження даних CIFAR-10...")
    train_loader, test_loader = load_cifar10_data(batch_size=128)
```

```
        # Ініціалізація моделі
        print("\n2. Ініціалізація моделі ResNet-32...")
        base_model = models.resnet32(num_classes=10)

        # Тренування базової моделі (якщо потрібно)
        print("\n3. Тренування базового класифікатора...")
        # Закоментуйте, якщо використовуєте попередньо навчену модель
        # base_model = train_base_classifier(base_model, train_loader,
test_loader, epochs=100)

        # Завантаження попередньо навченої моделі
        try:

base_model.load_state_dict(torch.load('pretrained_resnet32_cifar10.pth'))
            print("   Завантажено попередньо навчену модель")
        except:
            print("   Попередньо навченої моделі не знайдено")

        # Експеримент 1: Базове Randomized Smoothing
        print("\n" + "=" * 60)
        print("ЕКСПЕРИМЕНТ 1: Базове Randomized Smoothing")
        print("=" * 60)

        sigma_values = [0.1, 0.25, 0.5, 0.75, 1.0]
        radii_to_test = [0.0, 0.25, 0.5, 0.75, 1.0]

        results_basic = {}

        for sigma in sigma_values:
            print(f"\nТестування з σ = {sigma}")
```

```
    base_rs = BaseRandomizedSmoothing(
        base_classifier=base_model,
        sigma=sigma,
        num_samples=1000,
        device='cuda' if torch.cuda.is_available() else 'cpu'
    )

    # Оцінка сертифікованої точності
    certified_acc = base_rs.evaluate_certified_accuracy(
        test_loader,
        radii=radii_to_test
    )

    results_basic[sigma] = certified_acc

# Експеримент 2: Адаптивне Randomized Smoothing
print("\n" + "=" * 60)
print("ЕКСПЕРИМЕНТ 2: Адаптивне Randomized Smoothing")
print("=" * 60)

# Створення та навчання адаптера
print("\nСтворення шумового адаптера...")
adapter = NoiseAdapter(
    input_dim=512,  # Для ResNet-32
    hidden_dims=[256, 128],
    output_dim=1,
    dropout_rate=0.2
)

print("Навчання адаптера...")
```

```
adapter = train_noise_adapter(base_model, adapter, train_loader, test_loader, epochs=30)

# Створення адаптивної моделі
print("\nІніціалізація Adaptive Randomized Smoothing...")
adaptive_rs = AdaptiveRandomizedSmoothing(
    base_classifier=base_model,
    adapter=adapter,
    sigma_min=0.1,
    sigma_max=1.0,
    num_samples=1000,
    device='cuda'
)

# Оцінка адаптивної моделі
print("\nОцінка адаптивної моделі...")
clean_acc, certified_acc_adaptive = adaptive_rs.evaluate_adaptive(
    test_loader,
    radii=radii_to_test
)

# Експеримент 3: Структуровані шумові моделі
print("\n" + "=" * 60)
print("ЕКСПЕРИМЕНТ 3: Структуровані шумові моделі")
print("=" * 60)

noise_types = ['gaussian', 'correlated', 'sparse', 'lowrank']
results_structured = {}

for noise_type in noise_types:
```

```
print(f"\nТестування з {noise_type} шумом")

structured_rs = StructuredNoiseRS(
    base_classifier=base_model,
    noise_type=noise_type,
    sigma=0.25,
    num_samples=1000,
    device='cuda'
)

# Оцінка для обмеженої кількості зразків
certified_acc_structured = []

for radius in radii_to_test:
    correct = 0
    total = 0

    # Обмежуємо кількість зразків для швидкості
    for i, (images, labels) in enumerate(test_loader):
        if i >= 10:  # 10 батчів ≈ 1280 зразків
            break

        images, labels = images.to('cuda'), labels.to('cuda')

        for j in range(images.size(0)):
            x = images[j:j+1]
            y_true = labels[j].item()

            pred_class, cert_radius = structured_rs.certify_structured(x)
```

```
                    if pred_class == y_true and cert_radius >= radius:
                        correct += 1

                    total += 1

                accuracy = correct / total if total > 0 else 0.0
                certified_acc_structured.append(accuracy)

                print(f"  Radius {radius}: {accuracy:.4f}")

            results_structured[noise_type] = certified_acc_structured

        # Візуалізація результатів
        print("\n" + "=" * 60)
        print("ВІЗУАЛІЗАЦІЯ РЕЗУЛЬТАТІВ")
        print("=" * 60)

        plot_results(results_basic, results_structured, certified_acc_adaptive,
radii_to_test)

        print("\nЕксперименти завершено успішно!")

        return {
            'basic': results_basic,
            'adaptive': certified_acc_adaptive,
            'structured': results_structured,
            'clean_accuracy': clean_acc
        }

    def plot_results(basic_results, structured_results, adaptive_results, radii):
```

```
        """
        Побудова графіків результатів.

        Args:
            basic_results: Результати базового RS
            structured_results: Результати структурованого RS
            adaptive_results: Результати адаптивного RS
            radii: Список радіусів
        """
        fig, axes = plt.subplots(1, 3, figsize=(18, 6))

        # Графік 1: Базове RS з різними σ
        ax1 = axes[0]
        for sigma, accuracies in basic_results.items():
            ax1.plot(radii, accuracies, marker='o', label=f'σ={sigma}')
        ax1.set_xlabel('Радіус ε')
        ax1.set_ylabel('Сертифікована точність')
        ax1.set_title('Базове Randomized Smoothing')
        ax1.grid(True, alpha=0.3)
        ax1.legend()
        ax1.set_ylim([0, 1])

        # Графік 2: Адаптивне RS
        ax2 = axes[1]
        ax2.plot(radii, adaptive_results, marker='s', color='red', linewidth=3,
label='A-RS з адаптером')

        # Додавання базового RS з σ=0.25 для порівняння
        if 0.25 in basic_results:
            ax2.plot(radii, basic_results[0.25], marker='o', color='blue',
```

```
                linestyle='--', label='Базовий RS (σ=0.25)')

        ax2.set_xlabel('Радіус ε')
        ax2.set_ylabel('Сертифікована точність')
        ax2.set_title('Адаптивне Randomized Smoothing')
        ax2.grid(True, alpha=0.3)
        ax2.legend()
        ax2.set_ylim([0, 1])

        # Графік 3: Структуровані моделі
        ax3 = axes[2]
        colors = ['blue', 'green', 'red', 'purple']

        for idx, (noise_type, accuracies) in
enumerate(structured_results.items()):
            ax3.plot(radii, accuracies, marker='^', color=colors[idx],
                label=noise_type.capitalize())

        ax3.set_xlabel('Радіус ε')
        ax3.set_ylabel('Сертифікована точність')
        ax3.set_title('Структуровані шумові моделі')
        ax3.grid(True, alpha=0.3)
        ax3.legend()
        ax3.set_ylim([0, 1])

        plt.tight_layout()
        plt.savefig('rs_experiment_results.png', dpi=300, bbox_inches='tight')
        plt.show()

    if __name__ == "__main__":
```

```
# Запуск головної функції
results = main()

# Збереження результатів
import pickle
with open('experiment_results.pkl', 'wb') as f:
    pickle.dump(results, f)

print("\nРезультати збережено у файлах:")
print("  - rs_experiment_results.png (графіки)")
print("  - experiment_results.pkl (дані)")
```

ДОДАТОК Г

Конфігураційний файл та утіліти

```
"""
Утилітарні функції для роботи з Randomized Smoothing.
Автор: Рижова Олександра
Дата: 24.11.2025
"""

import torch
import numpy as np
import random
from typing import List, Tuple, Dict, Any
import matplotlib.pyplot as plt
from scipy.stats import beta, norm

def set_random_seed(seed: int = 42):
    """
    Встановлення seed для відтворюваності результатів.

    Args:
        seed: Значення seed
    """
    random.seed(seed)
    np.random.seed(seed)
    torch.manual_seed(seed)
    torch.cuda.manual_seed(seed)
    torch.cuda.manual_seed_all(seed)
    torch.backends.cudnn.deterministic = True
    torch.backends.cudnn.benchmark = False
```

```
    def compute_certified_radius(p_a: float, p_b: float, sigma: float,
                                 alpha: float = 0.001) -> float:
        """
        Обчислення сертифікованого радіуса за формулою RS.

        Args:
            p_a: Ймовірність головного класу
            p_b: Ймовірність другого класу
            sigma: Стандартне відхилення шуму
            alpha: Рівень значущості

        Returns:
            radius: Сертифікований радіус
        """
        if p_a <= p_b or p_a < 0.5:
            return 0.0

        # Обчислення з урахуванням статистичної невизначеності
        z_alpha = norm.ppf(1 - alpha)
        margin = p_a - p_b

        # Консервативна оцінка
        radius = sigma * norm.ppf(p_a - z_alpha * np.sqrt(p_a * (1 - p_a) / 1000))
- \
                 sigma * norm.ppf(p_b + z_alpha * np.sqrt(p_b * (1 - p_b) / 1000))

        return max(0.0, radius)

    def visualize_noise_comparison(original_img: torch.Tensor,
```

```
                    noise_types: List[str],
                    sigma: float = 0.25):
    """
    Візуалізація порівняння різних типів шуму.

    Args:
        original_img: Оригінальне зображення
        noise_types: Список типів шуму
        sigma: Інтенсивність шуму
    """
    from structured_noise_models import StructuredNoiseRS

    num_types = len(noise_types)
    fig, axes = plt.subplots(1, num_types + 1, figsize=(4 * (num_types + 1),
4))

    # Оригінальне зображення
    axes[0].imshow(original_img.permute(1, 2, 0).cpu().numpy())
    axes[0].set_title('Оригінал')
    axes[0].axis('off')

    # Різні типи шуму
    for idx, noise_type in enumerate(noise_types):
        # Створення шумової моделі
        dummy_model = torch.nn.Sequential()  # Заглушка
        noise_rs = StructuredNoiseRS(
            base_classifier=dummy_model,
            noise_type=noise_type,
            sigma=sigma,
            num_samples=1
```

```
            )

            # Генерація шуму
            img_tensor = original_img.unsqueeze(0)
            noise = noise_rs.generate_structured_noise(img_tensor)
            noisy_img = img_tensor + noise

            # Відображення
            axes[idx + 1].imshow(noisy_img[0].permute(1, 2,
0).cpu().detach().numpy())
            axes[idx + 1].set_title(f'{noise_type.capitalize()} шум')
            axes[idx + 1].axis('off')

        plt.tight_layout()
        plt.savefig('noise_comparison.png', dpi=300, bbox_inches='tight')
        plt.show()

    def plot_certification_curve(radii: List[float], accuracies: List[float],
                                 model_name: str = "Model", save_path: str = None):
        """
        Побудова кривої сертифікованої точності.

        Args:
            radii: Список радіусів
            accuracies: Список точностей
            model_name: Назва моделі
            save_path: Шлях для збереження графіка
        """
        plt.figure(figsize=(8, 6))
        plt.plot(radii, accuracies, 'b-o', linewidth=2, markersize=8)
```

```
        plt.fill_between(radii, accuracies, alpha=0.3)

        plt.xlabel('Радіус ε (L₂ норма)', fontsize=12)
        plt.ylabel('Сертифікована точність', fontsize=12)
        plt.title(f'Крива сертифікованої точності: {model_name}',
fontsize=14)
        plt.grid(True, alpha=0.3)
        plt.xlim([0, max(radii)])
        plt.ylim([0, 1])

        # Додавання анотацій
        for i, (r, a) in enumerate(zip(radii, accuracies)):
            plt.annotate(f'{a:.3f}', xy=(r, a), xytext=(5, 5),
                         textcoords='offset points', fontsize=10)

        if save_path:
            plt.savefig(save_path, dpi=300, bbox_inches='tight')

        plt.show()

    def compute_average_certified_radius(radii: List[float],
                                         accuracies: List[float]) -> float:
        """
        Обчислення середнього сертифікованого радіуса (ACR).

        Args:
            radii: Список радіусів
            accuracies: Список точностей

        Returns:
```

```
        acr: Середній сертифікований радіус
    """
    # Обчислення площі під кривою (AUC)
    auc = np.trapz(accuracies, radii)

    # Нормалізація для отримання середнього радіуса
    acr = auc / (max(radii) - min(radii)) if len(radii) > 1 else 0.0

    return acr

def print_results_summary(results: Dict[str, Any]):
    """
    Виведення зведеної таблиці результатів.

    Args:
        results: Словник з результатами
    """
    print("\n" + "=" * 70)
    print("ЗВЕДЕНА ТАБЛИЦЯ РЕЗУЛЬТАТІВ")
    print("=" * 70)

    if 'basic' in results:
        print("\n1. Базове Randomized Smoothing:")
        print("-" * 40)
        for sigma, accuracies in results['basic'].items():
            acr = compute_average_certified_radius(
                list(range(len(accuracies))),
                accuracies
            )
            print(f"  σ = {sigma}: ACR = {acr:.4f}")
```

```
        if 'adaptive' in results:
            print("\n2. Адаптивне Randomized Smoothing (A-RS):")
            print("-" * 40)
            adaptive_acc = results['adaptive']
            adaptive_acr = compute_average_certified_radius(
                list(range(len(adaptive_acc))),
                adaptive_acc
            )
            print(f"  ACR = {adaptive_acr:.4f}")

            # Порівняння з базовим σ=0.25
            if 'basic' in results and 0.25 in results['basic']:
                basic_acc = results['basic'][0.25]
                basic_acr = compute_average_certified_radius(
                    list(range(len(basic_acc))),
                    basic_acc
                )
                improvement = ((adaptive_acr / basic_acr) - 1) * 100
                print(f"  Покращення відносно базового RS (σ=0.25): {improvement:.2f}%")

        if 'structured' in results:
            print("\n3. Структуровані шумові моделі:")
            print("-" * 40)
            for noise_type, accuracies in results['structured'].items():
                acr = compute_average_certified_radius(
                    list(range(len(accuracies))),
                    accuracies
                )
```

```
            print(f"  {noise_type.capitalize()}: ACR = {acr:.4f}")

    if 'clean_accuracy' in results:
        print("\n4. Чиста точність (без атаки):")
        print("-" * 40)
        print(f"  {results['clean_accuracy']:.4f}")

    print("=" * 70 + "\n")
```

ДОДАТОК Д

Відомість кваліфікаційної роботи

Позначення	Найменування	Дод. відомості
	Текстові документи	
1.	Пояснювальна записка	116 с.
	Інші документи	
2.	Презентаційні матеріали	__ плакатів

.	.	Прізвище та ініціали.	Підп.	Дата			
Розробив		Рижова О.Ю.			Покращення сертифікованої стійкості Randomized Smoothing засобами геометрії розподілів, адаптивних збурень та структурованих моделей шуму	Шифр групи	Код напр./спец.
Перевірив		Шевченко О.Ю.				ДСм-24-1	122
Н.контр.		Малєєва І.А.				ХНУРЕ кафедра ШІ	
Затв.		Чала Л.Е.					