Homework 5
CSC 277 / 477
End-to-end Deep Learning
Fall 2024

Tianyi Zhou - tzhou25@u.rochester.edu

Nov.19 2024

Deadline: See Blackboard

Instructions

Your homework solution must be typed and prepared in LaTeX. It must be output to PDF format. To use LaTeX, we suggest using `http://overleaf.com`, which is free.

Your submission must cite any references used (including articles, books, code, websites, and personal communications). All solutions must be written in your own words, and you must program the algorithms yourself. **If you do work with others, you must list the people you worked with.** Submit your solutions as a PDF to Blackboard.

Your programs must be written in Python. The relevant code should be in the PDF you turn in. If a problem involves programming, then the code should be shown as part of the solution. One easy way to do this in LaTeX is to use the verbatim environment, i.e., \begin{verbatim} YOUR CODE \end{verbatim}.

Problem 1 - Out-of-Distribution Detection (11 Points)

Out-of-distribution (OOD) detection refers to identifying data points that do not conform to the same distribution as the training data. In this context, a classifier needs to reject a novel input from classes unseen during training rather than assigning it an incorrect label. Two broad categories of methods have been developed to enable a model for OOD detection:

1. **Inference Methods**: These techniques create an explicit acceptance score function to distinguish novel inputs from familiar ones based on the confidence or uncertainty measure associated with a model's predictions.
2. **Regularization Methods**: Regularization approaches modify the feature representations of a model during training. The objective is to enhance the model's ability to discriminate between in-distribution and novel samples.

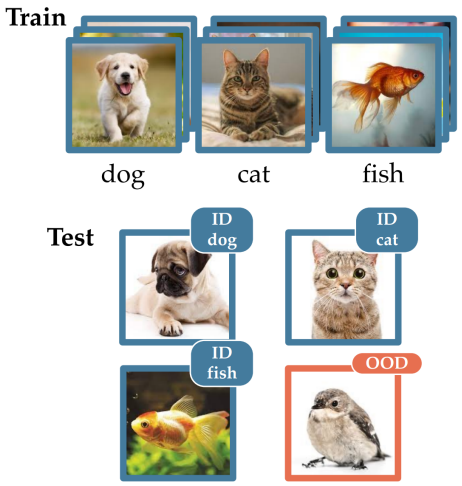

Figure 1: Illustration of OOD task. ID stands for in-distribution.

In this problem, you will leverage the pytorch-ood package to explore various OOD detection methods, which not only implements commonly used methods but also provides readily available pre-trained models and datasets. To get started with this problem, begin by following the package installation instructions and its dependencies as outlined in the provided guide. Please note that upgrading to the latest version of PyTorch may be necessary if you encounter any error messages during package usage.

Part 1: Inference Method

Part 1.1: Maximum Softmax Probability Thresholding (6 Points)

The Maximum Softmax Probability (MSP) Thresholding method represents a straightforward baseline approach for OOD detection. This method is designed to assess the confidence of a model's predictions by analyzing the final output of the model after applying the softmax activation function. The goal is to determine if a given input falls within the known classes or if it should be classified as an OOD instance.

Formally, the MSP score for a given input instance is defined as:

$$-\max_y \sigma_y(f(x)/T)$$

Where σ is the softmax function, T is the optimal temperature, and σ_y indicates the y^{th} value of the resulting probability vector.

Intuitive Understanding: The intuition behind MSP Thresholding is that if the maximum softmax score, a heuristic measure for certainty, is low, the model is uncertain about the most likely class for a given input. This uncertainty implies that the input might not belong to the classes seen during training, making it a potential candidate for an OOD instance.

In this section, you will assess the performance of the MSP Thresholding baseline method on a pre-trained model using the CIFAR-10 training dataset. Follow the steps below to familiarize yourself with the usage of this package and learn how to perform OOD detection:

1. Begin by following the quick start guide
    - Load a WideResNet model pretrained on CIFAR-10 by setting the `pretrained` parameter to `cifar10-pt`.
    - Prepare an MSP detector using the `detector.MaxSoftmax` class with the default temperature settings.
    - Instantiate an `OODMetrics` class.
2. Define the Test Set for OOD Detection:
    - Create a test set for OOD detection. This set combines the CIFAR-10 **test set**, with the resized LSUN dataset. For guidance on obtaining the evaluation transform for a pretrained model, refer to the code tutorial here.
    - You can use the `torch.utils.data.ConcatDataset` function to concatenate the two datasets effectively.

3. Perform OOD Detection on the Test Set:
    - Follow the steps outlined in the quick start guide to perform OOD detection on the test set you've defined.
    - Compute the metrics to evaluate the performance of your OOD detection method.
4. Generate an ROC Curve:
    - While the `pytorch-ood` package may not provide a direct method for generating an ROC curve, you can access the necessary data for creating the curve within the Metric object. (Hint: refer to the implementation of the OODMetrics for details on where the required data is stored.)
    - Utilize libraries such as `sklearn` and `matplotlib` to create the ROC curve based on the retrieved data.
    - Save the data for usage in the next parts.

Deliverable:

1. Provide the labels for data in the in-distribution and OOD datasets, along with the size of each dataset. Present your answer in a LaTeX table.
2. Create a LaTeX table summarizing the logged metrics obtained from the `OODMetrics` class. Use up arrows (↑) or down arrows (↓) to indicate whether a particular metric is considered better when larger or smaller, respectively.
3. Include the ROC curve you generated as part of your deliverable.
4. Briefly interpret the meaning of each metric in a few words. You may refer to Section 4.3 EVALUATION METRICS of this paper.

Answer:

Labels of data:
ID:

```
Label 0: airplane
Label 1: automobile
Label 2: bird
Label 3: cat
Label 4: deer
Label 5: dog
Label 6: frog
Label 7: horse
Label 8: ship
```

```
Label 9: truck
```

OOD:

```
Unique labels in LSUN Resized dataset: {-1}
```

Size of the dataset:

Dataset	Size
ID	10,000
OOD	10,000

Table 1: Size of each dataset

OOD Detection Metrics:

```
AUROC: 0.91% ↑
AUPR-IN: 0.92% ↑
AUPR-OUT: 0.89% ↑
FPR95TPR: 0.30% ↓
```

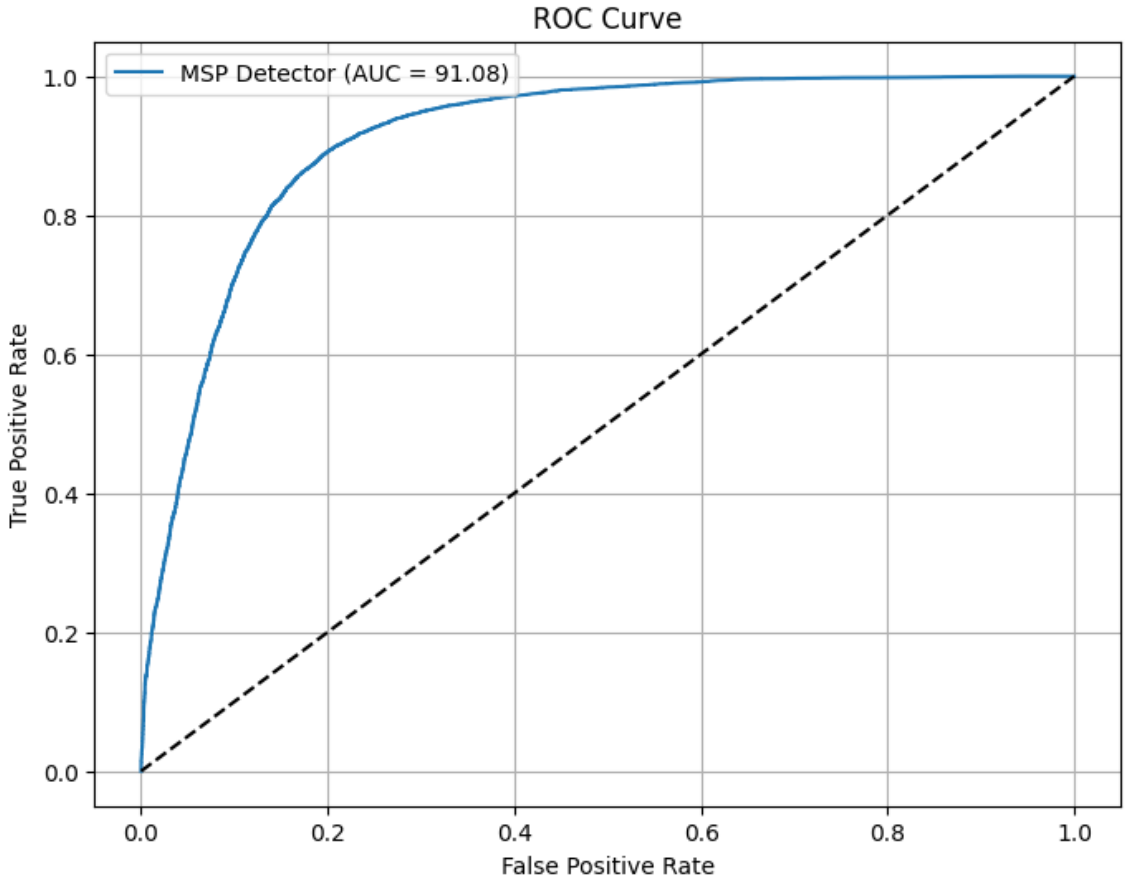

Figure 2: MSP ROC Curve

Interpretation:

1. 91% in **AUROC** suggests MSP has high overall ability to distinguish between ID and OOD data.
2. 92% in **AUPR-IN** means when correctly identifying ID samples, detector maintains high precision and recall
3. 89% in **AUPR-OUT** means when correctly identifying OOD samples, detector, when treating OOD samples as positives, has less precision and recall compared to AUPR-IN

4. 30% **FPR95TPR** suggests that at a 95% threshold of ID samples being correctly identified, 30% of the OOD samples are incorrectly identified.

Part 1.2: ODIN (Out-of-DIstribution detector for Neural networks) (5 Points)

ODIN is a technique designed to enhance the OOD detection capabilities of neural networks, which primarily focuses on modifying the input data during inference to increase the network's sensitivity to OOD samples. It includes two key steps:

1. **Perturbation**: ODIN perturbs the input data by adding a small, carefully chosen perturbation that makes OOD samples more distinguishable from in-distribution samples:
$$\hat{x} = x - \epsilon \text{ sign}(\nabla_x \mathcal{L}(f(x)/T, \hat{y}))$$
where $\hat{y}$ is the predicted class of the network.
2. **Temperature Scaling**: Similar to MSP, ODIN scales the logits of the model with a temperature parameter.

In this part, you will carry out OOD detection using ODIN. Follow the following steps:

1. Begin by reading Section 3 of the ODIN paper. Understand the motivation behind input preprocessing in ODIN and how it relates to adversarial examples.
2. Conduct an experiment by replacing the OOD detection method with ODIN, using its default hyperparameters.
3. Repeat the experiment with ODIN, but this time use the hyperparameters specifically for the CIFAR-10 dataset, which is given here.
4. Create an ROC curve that includes the results from the **three** different settings you've experimented with so far (i.e., one for MSP, two for ODIN). Ensure that the labels and legends on the ROC curve are clear. Set the axis ranges according to Figure 1 in the ODIN paper to illustrate differences effectively.

Deliverables:

1. In a few words, explain how the perturbation process in ODIN relates to adversarial examples.
2. Report the numerical metrics obtained from your ODIN experiments, including results for **both** the default and tuned hyperparameter settings. Give your result in a latex table.
3. Present the ROC curve containing the results of the three experiments.

4. Briefly compare the results of the ODIN experiments with those of the baseline method used earlier in your study. Consider the advantage(s) and disadvantage(s) of ODIN compared to the baseline.

Answer:
The perturbation process in ODIN relates to adversarial examples because it adds small perturbation to the input data that maximize model's confidence (softmax output) between ID and OOD samples.

Numerical metrics:

Detector Method	**AUROC**	**AUPR-IN**	**AUPR-OUT**	**FPR0.95TPR**
MSP	93.2%	92.3%	89.0%	30.4%
ODIN_default	67.5%	72.0%	59.1%	71.1%
ODIN_CIFAR10	94.3%	94.5%	94.1%	27.5%

Table 2: Size of each dataset

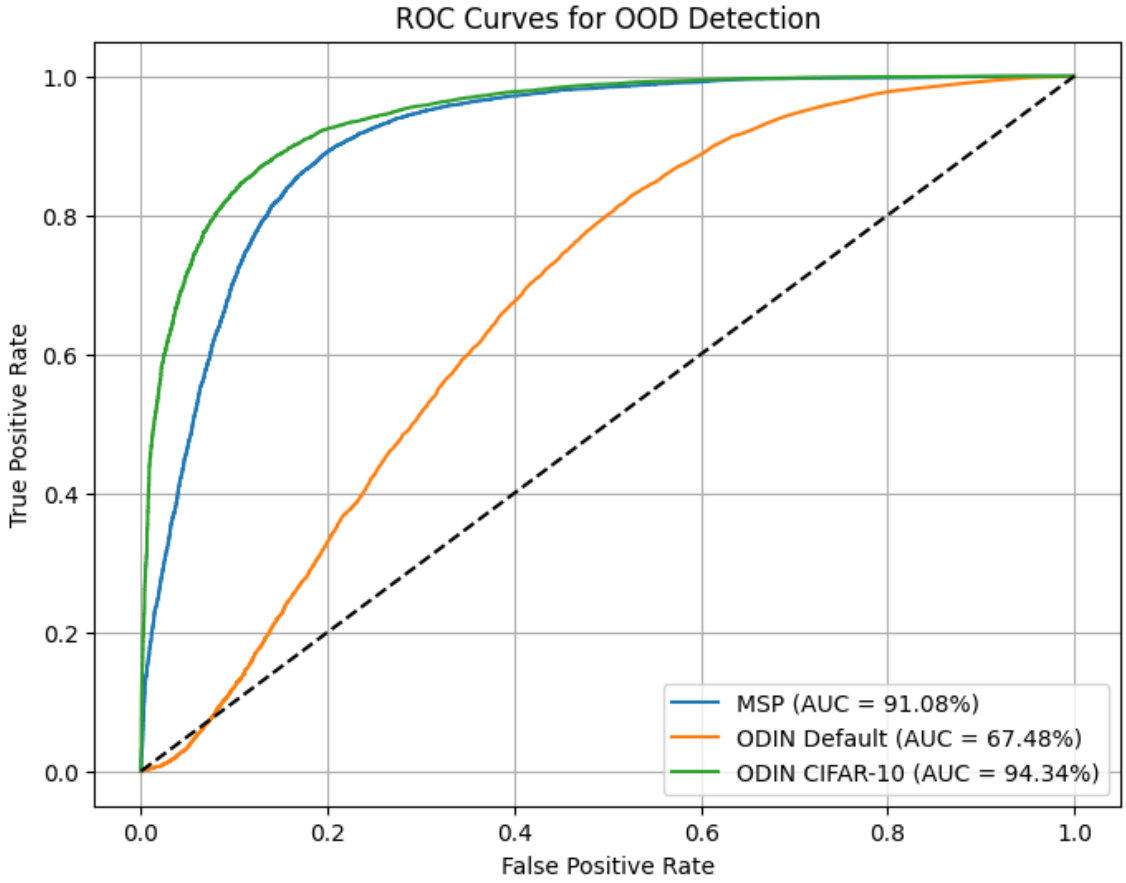

Figure 3: ROC curves of MSP(blue), ODIN with default hyperparameters (orange) and ODIN with CIFAR10 hyperparameters (green)

Problem 2 - Continual Learning (6 Points)

Continual learning focuses on training models that can progressively learn from data that becomes available over time. In an online learning (or streaming learning) setting, each data point is processed only once, and the data often exhibits non-independent and identically distributed (non-iid) characteristics. This poses a significant challenge for deep neural networks, which tend to suffer from **catastrophic forgetting**—the phenomenon where learning new information causes the model to forget previously acquired knowledge. Traditional fine-tuning methods typically fail in this scenario because they overwrite existing representations with new ones.

In this problem, you will compare a fine-tuning baseline with a simple yet effective continual learning algorithm, **Streaming Linear Discriminant Analysis** (SLDA), which has been developed to enable models to learn from new data while retaining past knowledge incrementally. Follow these steps:

1. Read the paper, specifically Section 3, and the SLDA implementation in `slda.py` to understand how SLDA works.
2. Examine the attached code for this problem to understand the experimental setup. The dataset is split into 5 chunks, each containing a distinct set of classes. The model is trained on these chunks one after another and is evaluated on all learned chunks after each session.
3. Execute the experiments by running `main.py` with and without SLDA. The performance will be printed out. Analyze the performance on the recently learned classes and the previously learned classes.

Deliverables:

1. What is the number of **trainable** parameters in the naive fine-tuning baseline? What is the number in Streaming LDA? Explain the differences by analyzing which parts of the model are trainable.
2. Analyze the forgetting you observed in these two conditions (fine-tuning vs. SLDA) by tracking the performance on a specific chunk immediately after learning it and after learning subsequent tasks. Describe and interpret your findings.
3. What happens when the data stream is iid? Present a table showing the average accuracy on the test set under these four conditions: Fine-tuning *vs.* SLDA; iid *vs.* non-iid data. Describe how data distribution introduces catastrophic forgetting. (Hint: 1. You can easily experiment with this condition by changing the `num_split` parameter in `main.py`. 2. Assume each chunk of test data is of the same size, so you can compute the overall test accuracy by averaging the accuracies on each chunk.)

Answer:

1. **Number of trainable parameters:**
   Naive fine-tuning baseline has 11689512 trainable parameters, while Streaming LDA has 0 trainable parameters. The difference is because SLDA uses incremental formulas for weight updates without calculating gradients.
2. **Fine-tuning**:

```
Training on split 0
         Accuracy on split 0: 0.9214285714285714
Training on split 1
         Accuracy on split 0: 0.1
         Accuracy on split 1: 0.9071428571428571
Training on split 2
         Accuracy on split 0: 0.03571428571428571
         Accuracy on split 1: 0.1357142857142857
         Accuracy on split 2: 0.8714285714285714
Training on split 3
         Accuracy on split 0: 0.02857142857142857
         Accuracy on split 1: 0.014285714285714285
         Accuracy on split 2: 0.22857142857142856
         Accuracy on split 3: 0.9714285714285714
Training on split 4
         Accuracy on split 0: 0.0
         Accuracy on split 1: 0.0
         Accuracy on split 2: 0.007142857142857143
         Accuracy on split 3: 0.1357142857142857
         Accuracy on split 4: 0.9640287769784173
```

Figure 4: Training incrementally on four splits using Naive fine-tuning

We can see, for example, in split 2, as soon as the model learned new data from split 2 with accuracy 0.87, it immediately forgets previous learned knowledge: accuracy on split 1 plummets from 0.907 to 0.136.

SLDA:

```
Training on split 0
        Accuracy on split 0: 0.9142857142857143
Training on split 1
        Accuracy on split 0: 0.85
        Accuracy on split 1: 0.9071428571428571
Training on split 2
        Accuracy on split 0: 0.8285714285714286
        Accuracy on split 1: 0.8928571428571429
        Accuracy on split 2: 0.9357142857142857
Training on split 3
        Accuracy on split 0: 0.85
        Accuracy on split 1: 0.8785714285714286
        Accuracy on split 2: 0.9214285714285714
        Accuracy on split 3: 0.9642857142857143
Training on split 4
        Accuracy on split 0: 0.8428571428571429
        Accuracy on split 1: 0.8785714285714286
        Accuracy on split 2: 0.9214285714285714
        Accuracy on split 3: 0.9642857142857143
        Accuracy on split 4: 0.9496402877697842
```

Figure 5: Training incrementally on four splits using SLDA

We can see, for example, in split 2, as the accuracy on split 2 is 0.936, accuracy on split 1 does not drop much: only a slight drop from 0.907 to 0.893. The forgetting is drastically improved by adapting SLDA from naive fine-tuning.

Condition	**Split 1**	**Split 2**	**Split 3**	**Split 4**	**Split 5**	**Average Accuracy**
Fine-tune, IID						88.1%
Fine-tune, Non-IID	0.7%	0.0	0.0	17.9%	95.0%	22.72%
SLDA, IID						91.0%
SLDA, Non-IID	84.3%	87.9%	91.4%	96.4%	95.0%	91%

Table 3: Size of each dataset

3. We can observe that non-iid data distribution introduce catastrophic forgetting in models that rely on due to incrementally learning paradigm. By comparing conventional fine-tune method and SLDA method's result, we can see that fine-tune models are vulnerable to non-iid data while SLDA demonstrated well robustness against non-iid data.

1 Appendix

Listing 1: Code Implementation for Problem 1 Part 1

```
import torch
import torchvision
from torch.utils.data import DataLoader, ConcatDataset
from pytorch_ood.model import WideResNet
from pytorch_ood.detector import MaxSoftmax
from pytorch_ood.utils import OODMetrics
from pytorch_ood.dataset.img import LSUNResize

import matplotlib.pyplot as plt
from sklearn.metrics import roc_curve, auc

import pickle
import numpy as np
from tqdm import tqdm


## Create Neural Network
device = "cuda" if torch.cuda.is_available() else "cpu"
model = WideResNet(num_classes=10, pretrained="cifar10-pt")
model.eval().to(device)
# Create detector
detector = MaxSoftmax(model)
# Instantiate metrics
metrics = OODMetrics()

## Define Test Set
trans = WideResNet.transform_for("cifar10-pt")
norm_std = WideResNet.norm_std_for("cifar10-pt")
# Load CIFAR-10 test set
cifar10_test = torchvision.datasets.CIFAR10(root="./data", train=False,
    download=True, transform=trans)
# Load LSUN test set
lsun_test = LSUNResize(root="./data", download=True, transform=trans)
#%%
# Extract labels
cifar10_test_labels = [label for _, label in cifar10_test]
cifar10_cls_name = cifar10_test.classes
unique_labels = set(cifar10_test_labels)
for label in sorted(unique_labels):
    print(f"Label {label}: {cifar10_cls_name[label]}")

# Attempt to extract labels from the LSUN Resized dataset
lsun_labels = [label for _, label in lsun_test]
unique_lsun_labels = set(lsun_labels)
print("Unique labels in LSUN Resized dataset:", unique_lsun_labels)
#%%
## Perform OOD detection
test_loader = DataLoader(cifar10_test+lsun_test, batch_size=128,
   shuffle=False)
```

```
    all_scores, all_labels = [],[]
    with torch.no_grad():
        for data, labels in tqdm(test_loader, desc="Processing batches"):
            outputs = detector(data.to(device))
            scores = outputs.cpu().numpy()
            all_scores.extend(scores)

            labels = labels.cpu().numpy()
            binary_labels = np.where(labels >= 0, 0, 1)
            all_labels.extend(binary_labels)
            # Update metrics
            metrics.update(outputs, torch.tensor(labels))
    # Print results
    result = metrics.compute()
    print('MSP Result: ', result)
    #%%
    ## Plot ROC curve
    fpr, tpr, _ = roc_curve(all_labels, all_scores)
    roc_auc = auc(fpr, tpr)

    plt.figure(figsize=(8,6))
    plt.plot(fpr, tpr, label="MSP Detector (AUC = {:.2f})".format(roc_auc
        *100))
    plt.plot([0,1],[0,1], "k--")
    plt.xlabel("False Positive Rate")
    plt.ylabel("True Positive Rate")
    plt.title("ROC Curve")
    plt.legend(loc='upper left')
    plt.grid(True)
    plt.show()

    # Save results
    with open("msp_result.pkl", "wb") as f:
        pickle.dump({
            'scores': all_scores,
            'labels': all_labels,
            'metrics': result,
            'fpr': fpr,
            'tpr': tpr,
        },f)

    # Print result:
    print("\nOOD Detection Metrics:")
    for k, v in result.items():
        direction = '   ' if 'AU' in k else '   '
        print(f"{k}: {v:.2f}% {direction}")
```

Listing 2: Code Implementation for Problem 1 Part 2

```
    import torch
```

```
import torchvision
from torch.utils.data import DataLoader, ConcatDataset
from pytorch_ood.model import WideResNet
from pytorch_ood.detector import ODIN, MaxSoftmax
from pytorch_ood.utils import OODMetrics
from pytorch_ood.dataset.img import LSUNResize

import matplotlib.pyplot as plt
from sklearn.metrics import roc_curve, auc

import pickle
import numpy as np
from tqdm import tqdm
#%%
## Create Neural Network
device = "cuda" if torch.cuda.is_available() else "cpu"
model = WideResNet(num_classes=10, pretrained="cifar10-pt")
model.eval().to(device)
# Create detector
detector = ODIN(model)
# Instantiate metrics
metrics = OODMetrics()

## Define Test Set
trans = WideResNet.transform_for("cifar10-pt")
norm_std = WideResNet.norm_std_for("cifar10-pt")
# Load CIFAR-10 test set
cifar10_test = torchvision.datasets.CIFAR10(root="./data", train=False,
    download=True, transform=trans)
# Load LSUN test set
lsun_test = LSUNResize(root="./data", download=True, transform=trans)

test_loader = DataLoader(cifar10_test+lsun_test, batch_size=128,
    shuffle=False)
#%%
def ood_detection(detector, model_name):
    metrics = OODMetrics()
    all_scores, all_labels = [], []
    with torch.no_grad():
        for data, labels in tqdm(test_loader, desc=f'Processing batches
            with {model_name}'):
            data = data.to(device)
            labels = labels.numpy()
            binary_labels = np.where(labels >= 0, 0, 1)

            # For detectors which needs gradients
            data.requires_grad = True
            outputs = detector(data)
            data.requires_grad = False
            scores = outputs.cpu().numpy()
```

```
            all_scores.extend(scores)
            all_labels.extend(binary_labels)

            metrics.update(outputs, torch.tensor(labels))
    result = metrics.compute()
    print(f"{model_name} OOD Detection Results: {result}")
    return all_labels, all_scores, result
#%%
## Compute metrics
msp_detector = MaxSoftmax(model)
odin_default_detector = ODIN(model)
odin_cifar10_detector = ODIN(model, eps=0.002, norm_std=norm_std)

# Perform OOD detection for each detector
print("Performing OOD detection with MSP...")
msp_labels, msp_scores, msp_result = ood_detection(msp_detector, "MSP")

print("\nPerforming OOD detection with ODIN (Default Hyperparameters)
    ...")
odin_default_labels, odin_default_scores, odin_default_result =
    ood_detection(odin_default_detector, "ODIN Default")

print("\nPerforming OOD detection with ODIN (CIFAR-10 Hyperparameters)
    ...")
odin_cifar10_labels, odin_cifar10_scores, odin_cifar10_result =
    ood_detection(odin_cifar10_detector, "ODIN CIFAR-10")
#%%
## Plot curves
plt.figure(figsize=(8, 6))
# MSP ROC Curve
fpr_msp, tpr_msp, _ = roc_curve(msp_labels, msp_scores)
roc_auc_msp = auc(fpr_msp, tpr_msp)
plt.plot(fpr_msp, tpr_msp, label="MSP (AUC = {:.2f}%)".format(
    roc_auc_msp * 100))

# ODIN Default ROC Curve
fpr_odin_def, tpr_odin_def, _ = roc_curve(odin_default_labels,
    odin_default_scores)
roc_auc_odin_def = auc(fpr_odin_def, tpr_odin_def)
plt.plot(fpr_odin_def, tpr_odin_def, label="ODIN Default (AUC = {:.2f
    }%)".format(roc_auc_odin_def * 100))

# ODIN CIFAR-10 ROC Curve
fpr_odin_c10, tpr_odin_c10, _ = roc_curve(odin_cifar10_labels,
    odin_cifar10_scores)
roc_auc_odin_c10 = auc(fpr_odin_c10, tpr_odin_c10)
plt.plot(fpr_odin_c10, tpr_odin_c10, label="ODIN CIFAR-10 (AUC = {:.2f
    }%)".format(roc_auc_odin_c10 * 100))
```

```
    # Plot settings
    plt.plot([0, 1], [0, 1], "k--")  # Diagonal line for random guess
    plt.xlabel("False Positive Rate")
    plt.ylabel("True Positive Rate")
    plt.title("ROC Curves for OOD Detection")
    plt.legend(loc='lower right')
    plt.grid(True)
    plt.show()
```

Listing 3: Code Implementation for Problem 2 trainable params count

```
    if __name__ == '__main__':
    args = get_args()
    device = torch.device('cuda' if torch.cuda.is_available() else 'cpu')
    train_data_stream, test_data_stream, num_class = create_online_dataset(
        args.label_file, args.img_dir, num_split=args.num_split)

    if not args.use_slda:
        model = resnet18(weights=ResNet18_Weights.DEFAULT)
        print(f'Number of trainable parameters: {count_trainable_params(
            model)}')
        model.fc = nn.Linear(model.fc.in_features, num_class)
        model.to(device)
        online_training(model, train_data_stream, test_data_stream, device,
             batch_size=args.batch_size)

    else:
        slda = StreamingLDA(512, num_class, device=device)
        print(f'Number of trainable parameters: {count_trainable_params(
            slda)}')
        feature_extractor = resnet18(weights=ResNet18_Weights.DEFAULT)
        feature_extraction_wrapper = ModelWrapper(feature_extractor.to(
            device), ['layer4.1'], return_single=True).eval()
        online_training_slda(slda,feature_extraction_wrapper,
            train_data_stream, test_data_stream, device, batch_size=args.
            batch_size)
```