Notebook

February 20, 2025

1 1) Introduction

Deep neural networks have obtained impressive results in many applications related to images but encounter significant performance degradation when subject to images that are differ wrt the ones encountered in training due to the domain shift. To address this challenge, Test-Time Adaptation (TTA) has emerged as a technique to improve a model's robustness during inference without retraining on the new domain. The core idea behind TTA is to adapt the model's predictions to each test sample individually or to small sets of samples during inference. Our project is based on Test-time Prompt Tuning that builds upon the principles of TTA and it is designed for vision-language models (VLMs). In particular TPT adapts these VLMs to specific downstream tasks by fine-tuning their prompts (Coop) during inference. The primary objective of our research is to enhance the performance and robustness of Test-Time Prompt Tuning (TPT) by incorporating image patching at test time, inspired by the FILIP methodology. This approach involves subdividing the image into patches and leveraging these subdivisions to gain a deeper understanding of the original image. We aim to explore various strategies based on loss functions to optimize this process and achieve improved outcomes.

2 2) Background information

2.1 CLIP

CLIP (Contrastive Language-Image Pretraining) learns rich knowledge from diverse datasets, enabling zero-shot generalization. It maps images z_i and text t_i embeddings into a shared space using a contrastive loss:

$$\mathcal{L}_{\text{CLIP}} = -\frac{1}{N}\sum_{i=1}^{N}\log\frac{\exp(\cos(\mathbf{z}_i,\mathbf{t}_i)/\tau)}{\sum_j \exp(\cos(\mathbf{z}_i,\mathbf{t}_j)/\tau)},$$

where τ is a temperature parameter.

For our project we used the CLIP **RN50** architecture for the visual encoder.

2.2 COOP

CoOp (Context Optimization) reformulates CLIP's text prompts into learnable vectors $\mathbf{P} = [\mathbf{v}_1, \mathbf{v}_2, \dots, \mathbf{v}_M]$, where $\mathbf{v}_i \in \mathbb{R}^d$. The text embedding for a given class is expressed as

$$\mathbf{t} = [\mathbf{v}_1][\mathbf{v}_2]\cdots[\mathbf{v}_M][\text{CLASS}],$$

where [CLASS] is the embedding of the class name. CoOp optimizes $\mathbf{P}$ using the contrastive loss:

$$\mathcal{L}_{\text{CLIP}} = -\frac{1}{N}\sum_{i=1}^{N}\log\frac{\exp(\cos(\mathbf{z}_i, \mathbf{t}_i)/\tau)}{\sum_j \exp(\cos(\mathbf{z}_i, \mathbf{t}_j)/\tau)},$$

where $\mathbf{z}_i$ and $\mathbf{t}_i$ are the image and text embeddings, and τ is a temperature parameter. By **freezing the image and text encoders**, CoOp learns task-specific prompts, improving downstream performance and generalization to unseen classes.

In our project we initialize the prompt with the pre-trained weights that uses **M=4** and trained on **ImageNet** based on CLIP **RN50**.

2.3 TPT

TPT (Test-Time Prompt Tuning) is a technique designed to adapt vision-language models (VLMs) to downstream tasks during inference, without requiring access to training data. VLMs, such as CLIP, are pre-trained on large-scale image-text datasets and exhibit strong zero-shot transfer capabilities. These models typically rely on manually crafted prompts to perform downstream tasks. TPT improves performance by dynamically fine-tuning these prompts during test time, eliminating the need for handcrafted designs. \

Given only a test sample X_{test}, **without the corresponding label**, and p beeing the prompt to optimize. TPT minimize the entropy of the averaged prediction probability distribution to enhance prediction consistency across augmented views $\{A_i(X_{\text{test}})\}$:

$$p^* = \arg\min_p -\sum_{i=1}^{K}\tilde{p}_p(y_i \mid X_{\text{test}})\log\tilde{p}_p(y_i \mid X_{\text{test}}),$$

$$\text{where } \tilde{p}_p(y_i \mid X_{\text{test}}) = \frac{1}{N}\sum_{i=1}^{N} p_p(y_i \mid \mathcal{A}_i(X_{\text{test}})).$$

Here, $\tilde{p}_p(y \mid \mathcal{A}_i(X_{\text{test}}))$ is the vector of class probabilities produced by the model when provided with prompt p and the i-th augmented view of the test image. \ To filter noisy augmentations, TPT employs confidence selection based on entropy thresholds τ, refining predictions by masking low-confidence views:

$$\tilde{p}_p(y \mid X_{\text{test}}) = \frac{1}{\rho N}\sum_{i=1}^{N}\mathbf{1}[H(p_i) \leq \tau]\, p(y \mid A_i(X_{\text{test}})).$$

TPT aligns prompts to retrieve CLIP's knowledge, boosting generalization without labeled data.

$$c = \text{class}[i] = \arg\max_i \tilde{p}_{p^*}(y_i \mid X_{\text{test}})$$

After the training step, the final prediction c for classification is determined by selecting the class corresponding to the maximum output value of the model. For each subsequent test prediction, the **model is reinitialized**.

3 3) Initialization

3.1 Downloads project files

3.1.1 Download all the dependencies

[56]:
```
%%capture

!pip install numpy
!pip install torch
!pip install torchvision
!pip install matplotlib
!pip install pillow
!pip install scikit-image
!pip install ipykernel
!pip install ftfy
!pip install opencv-python
!pip install gdown
!pip install tensorflow
!pip install tensorboard
```

3.1.2 Download the dataset

[57]:
```
!gdown "https://drive.google.com/uc?id=1Fdx3kYLszDdYsBzqhfPYhfKUEXkFZbhW"
```

```
Downloading…
From (original):
https://drive.google.com/uc?id=1Fdx3kYLszDdYsBzqhfPYhfKUEXkFZbhW
From (redirected): https://drive.google.com/uc?id=1Fdx3kYLszDdYsBzqhfPYhfKUEXkFZ
bhW&confirm=t&uuid=a4963deb-f7f7-4e31-8b4b-f4e62854d134
To: /content/imagenet-a.tar
100% 688M/688M [00:15<00:00, 43.5MB/s]
```

[58]:
```
!tar -xf imagenet-a.tar
```

3.1.3 Download the CoOp weights

[59]:
```
!gdown "https://drive.google.com/uc?id=1shomUlN2fbbolKtIKigq0-FDyb_O4rHl"
```

```
Downloading…
From (original):
https://drive.google.com/uc?id=1shomUlN2fbbolKtIKigq0-FDyb_O4rHl
From (redirected): https://drive.google.com/uc?id=1shomUlN2fbbolKtIKigq0-
```

```
FDyb_O4rHl&confirm=t&uuid=339de4e0-1f73-44b7-89dc-ab50f5800e1a
To: /content/model.pth.tar-50
100% 74.8M/74.8M [00:01<00:00, 38.1MB/s]
```

3.1.4 Download our saved result

Since evaluating the entire dataset is expensive and time consuming, we will import some saved result. Nonetheless we will always provide the code commented to obtain the same results.

[60]:
```
!gdown "https://drive.google.com/uc?id=1X9OWqbrMvnOk4TiwH2T2p-y7sWe6vssu"
!tar -xf runs_baseline.tar

!gdown "https://drive.google.com/uc?id=1Ht-wF7CuwsxRISoDyytodz8T_ZmdPnqP"
!tar -xf runs_patch.tar

!gdown "https://drive.google.com/uc?id=1bIc4iTn9FN4KOagtJX4kT2xrRQtzwb5f"
!tar -xf runs_overlap.tar

!gdown "https://drive.google.com/uc?id=1ESbP3rSKXsja1Tt-a7GRBhuXNdg2qKH5"
!tar -xf runs_crossE_overlap.tar
```

```
Downloading…
From: https://drive.google.com/uc?id=1X9OWqbrMvnOk4TiwH2T2p-y7sWe6vssu
To: /content/runs_baseline.tar
100% 19.4M/19.4M [00:00<00:00, 72.7MB/s]
Downloading…
From: https://drive.google.com/uc?id=1Ht-wF7CuwsxRISoDyytodz8T_ZmdPnqP
To: /content/runs_patch.tar
100% 58.3M/58.3M [00:01<00:00, 34.2MB/s]
Downloading…
From: https://drive.google.com/uc?id=1bIc4iTn9FN4KOagtJX4kT2xrRQtzwb5f
To: /content/runs_overlap.tar
100% 83.3M/83.3M [00:02<00:00, 38.8MB/s]
Downloading…
From: https://drive.google.com/uc?id=1ESbP3rSKXsja1Tt-a7GRBhuXNdg2qKH5
To: /content/runs_crossE_overlap.tar
100% 66.8M/66.8M [00:01<00:00, 37.3MB/s]
```

[61]:
```
%load_ext tensorboard

import tensorflow as tf
from tensorflow.keras.callbacks import TensorBoard
```

```
The tensorboard extension is already loaded. To reload it, use:
  %reload_ext tensorboard
```

3.2 Utils

In this section there are some utility function, for example functions to make the visualization nicer and classes to perform and maintain different metrics.

[62]:
```
import matplotlib.pyplot as plt
from math import ceil

import torchvision.transforms.functional as TF

import torch
import torchvision.transforms as transforms
import numpy as np
import math

BICUBIC = transforms.InterpolationMode.BICUBIC


class IdentityTransform(torch.nn.Module):
    def forward(self, x):
        return x


class NormalizeImgTransform(torch.nn.Module):
    def forward(self, x):
        return x.float() / 255.0


class DeNormalizeImgTransform(torch.nn.Module):
    def forward(self, x):
        return (x.clamp(0, 1) * 255.0).to(torch.uint8)


def _convert_image_to_rgb(image):
    return image.convert("RGB")


# AugMix Transforms
def get_base_transform():
    return transforms.Compose(
        [
            transforms.Resize(224, interpolation=BICUBIC),
            transforms.CenterCrop(224),
            _convert_image_to_rgb,
            transforms.PILToTensor(),
        ]
    )
```

```
def get_normalizer():
    return transforms.Compose(
        [
            NormalizeImgTransform(),
            transforms.Normalize(
                mean=[0.48145466, 0.4578275, 0.40821073],
                std=[0.26862954, 0.26130258, 0.27577711],
            ),
        ]
    )


def get_unnormalizer():
    mean = torch.tensor([0.48145466, 0.4578275, 0.40821073])
    std = torch.tensor([0.26862954, 0.26130258, 0.27577711])
    return transforms.Compose(
        [
            transforms.Normalize(mean=(-mean / std), std=(1.0 / std)),
            DeNormalizeImgTransform(),
        ]
    )


def get_clip_preprocess():
    base_transform = get_base_transform()
    normalizer = get_normalizer()
    return transforms.Compose(base_transform.transforms + normalizer.transforms)


def get_preaugment():
    return transforms.Compose(
        [
            transforms.RandomResizedCrop(224),
            transforms.RandomHorizontalFlip(),
            transforms.PILToTensor(),
        ]
    )

clip_preprocess = get_clip_preprocess()
normalizer = get_normalizer()
unnormalizer = get_unnormalizer()

def show_imgs(images, images_per_row=4, title=None):
    """
    Display a list of PIL images in a grid using Matplotlib, with the first␣
 ↪image
```

```
    highlighted in its own row, and the rest shown in a grid.

    Args:
        images (list): List of PIL.Image objects to display.
        images_per_row (int): Number of images to display per row for the grid.␣
↪Default is 4.
        title (str): Optional title to display above the image grid.

    Returns:
        None
    """
    if not images:
        raise ValueError("The images list is empty.")

    # Separate the first image
    first_image = images[0]
    first_image = TF.to_pil_image(unnormalizer(first_image))
    other_images = images[1:]

    # Calculate grid dimensions for the other images
    num_images = len(other_images)
    num_rows = ceil(num_images / images_per_row)

    # Create the figure
    fig, axes = plt.subplots(num_rows + 1, images_per_row,␣
↪figsize=(images_per_row * 2, (num_rows + 1) * 2))
    axes = axes.reshape(-1, images_per_row)  # Reshape to handle rows and␣
↪columns

    # Add title if provided
    if title:
        fig.suptitle(title, fontsize=16, y=0.98)

    # Highlight the first image in the first row
    first_row_axes = axes[0]
    for ax in first_row_axes:
        ax.axis("off")  # Turn off all axes in the first row
    first_row_axes[0].imshow(first_image)
    first_row_axes[0].axis("on")  # Turn on the axis for the first image

    # Display the remaining images in a grid
    for ax, img in zip(axes[1:].flatten(), other_images):
        img = TF.to_pil_image(unnormalizer(img))
        ax.imshow(img)
        ax.axis("off")  # Turn off axis for grid images

    # Hide any extra subplots
```

```
    for ax in axes[1:].flatten()[num_images:]:
        ax.axis("off")

    plt.tight_layout()
    plt.subplots_adjust(top=0.9)  # Adjust layout to fit title
    plt.show()
```

[63]:
```
import random
import numpy as np
import torch
import torch.nn.functional as F
import json


def set_random_seed(seed):
    random.seed(seed)
    np.random.seed(seed)
    torch.manual_seed(seed)
    torch.cuda.manual_seed_all(seed)


class MetricsTracker:
    def __init__(self, args):
        self.samples = 0.0
        self.cumulative_changed = 0.0

        self.corrects_base = 0.0
        self.corrects_tpt = 0.0

        self.top5_corrects_base = 0.0
        self.top5_corrects_tpt = 0.0

        self.cumulative_crossEntropy_base = 0.0
        self.cumulative_crossEntropy_tpt = 0.0

        self.crossEntropy = torch.nn.CrossEntropyLoss()

        self.sample_statistics = [
            {
                "target": target,
                "tpt_improved_samples": [],
                "tpt_worsened_samples": [],
                "n_samples": 0,
            }
            for target in range(args.nclasses)
        ]
```

```
    def update(
        self, i, view_img, output_base, output_tpt, target, loss_value, writer,␣
↪args
    ):
        writer.add_scalar("Loss/value", loss_value, i)
        target_class = target.item()

        pred_base_conf, pred_base_class = torch.softmax(output_base, dim=1).
↪max(1)
        pred_base_probs = torch.softmax(output_base, dim=1)
        base_crossEntropy = self.crossEntropy(output_base, target)

        pred_tpt_conf, pred_tpt_class = torch.softmax(output_tpt, dim=1).max(1)
        pred_tpt_probs = torch.softmax(output_tpt, dim=1)
        tpt_crossEntropy = self.crossEntropy(output_tpt, target)

        _, top5_pred_base = torch.topk(output_base, 5, dim=1)
        _, top5_pred_tpt = torch.topk(output_tpt, 5, dim=1)

        writer.add_scalar("confidence/Base", pred_base_conf, i)
        writer.add_scalar("confidence/TPT_coop", pred_tpt_conf, i)
        writer.add_scalar(
            "confidence/difference(TPT-base)", pred_tpt_conf - pred_base_conf, i
        )

        self.cumulative_crossEntropy_base += base_crossEntropy
        self.cumulative_crossEntropy_tpt += tpt_crossEntropy
        writer.add_scalar("crossEntropy/Base", base_crossEntropy, i)
        writer.add_scalar("crossEntropy/TPT_coop", tpt_crossEntropy, i)
        writer.add_scalar(
            "crossEntropy/difference(base-TPT)", base_crossEntropy -␣
↪tpt_crossEntropy, i
        )

        true_dist = F.one_hot(target, num_classes=args.nclasses).float().
↪to(args.device)
        kl_base = F.kl_div(pred_base_probs.log(), true_dist,␣
↪reduction="batchmean")
        kl_tpt = F.kl_div(pred_tpt_probs.log(), true_dist,␣
↪reduction="batchmean")
        writer.add_scalar("KL/Base_vs_True", kl_base.item(), i)
        writer.add_scalar("KL/TPT_vs_True", kl_tpt.item(), i)
        writer.add_scalar("KL/Base-TPT", kl_base.item() - kl_tpt.item(), i)

        self.corrects_base += pred_base_class.eq(target).sum().item()
        self.corrects_tpt += pred_tpt_class.eq(target).sum().item()
```

```
        self.top5_corrects_base += (
            torch.sum(top5_pred_base == target.view(-1, 1)).float().item()
        )
        self.top5_corrects_tpt += (
            torch.sum(top5_pred_tpt == target.view(-1, 1)).float().item()
        )

        self.samples += 1

        # Calculate current accuracy
        curr_base_acc = self.get_accuracy_base()
        curr_TPTcoop_acc = self.get_accuracy_tpt()
        curr_base_top5_acc = self.get_top5_accuracy_base()
        curr_TPTcoop_top5_acc = self.get_top5_accuracy_tpt()

        writer.add_scalar("AccuracyTOP1/Base", curr_base_acc, i)
        writer.add_scalar("AccuracyTOP1/TPT_coop", curr_TPTcoop_acc, i)
        writer.add_scalar(
            "AccuracyTOP1/difference(TPT-base)", curr_TPTcoop_acc -␣
↪curr_base_acc, i
        )

        writer.add_scalar("AccuracyTOP5/Base_Top5", curr_base_top5_acc, i)
        writer.add_scalar("AccuracyTOP5/TPT_coop_Top5", curr_TPTcoop_top5_acc,␣
↪i)
        writer.add_scalar(
            "AccuracyTOP5/difference_Top5(TPT-base)",
            curr_TPTcoop_top5_acc - curr_base_top5_acc,
            i,
        )

        change = (
            pred_tpt_class.eq(target).sum().item()
            - pred_base_class.eq(target).sum().item()
        )
        writer.add_scalar("Samples/TPTchange", change, i)

        self.cumulative_changed += change
        writer.add_scalar("Samples/TPTcumulative_changed", self.
↪cumulative_changed, i)

        if change > 0:
            if args.save_imgs:
                writer.add_image(
                    f"Improved_Images/img_{i}:{target_class}-{args.
↪classnames[target_class]}",
```

```
                    view_img.squeeze(0),
                    target_class,
                    i,
                )
            self.sample_statistics[target]["tpt_improved_samples"].append(i)
        elif change < 0:
            if args.save_imgs:
                writer.add_image(
                    f"Worsened_Images/img_{i}:{target_class}-{args.
↪classnames[target_class]}",
                    view_img.squeeze(0),
                    target_class,
                    i,
                )
            self.sample_statistics[target]["tpt_worsened_samples"].append(i)

        self.sample_statistics[target]["n_samples"] += 1

    def write_info(self, writer, args):
        global_stat = {
            "tpt_improved_samples": [],
            "tpt_worsened_samples": [],
            "n_samples": 0,
        }
        for target in range(args.nclasses):
            ntpt_improved = len(self.
↪sample_statistics[target]["tpt_improved_samples"])
            ntpt_worsened = len(self.
↪sample_statistics[target]["tpt_worsened_samples"])
            self.sample_statistics[target]["ntpt_improved"] = ntpt_improved
            self.sample_statistics[target]["ntpt_worsened"] = ntpt_worsened

            writer.add_scalar("Samples-PerClass-info/NImproved", ntpt_improved,␣
↪target)
            writer.add_scalar("Samples-PerClass-info/NWorsened", ntpt_worsened,␣
↪target)
            writer.add_scalar(
                "Samples-PerClass/Changed", ntpt_improved - ntpt_worsened,␣
↪target
            )
            writer.add_scalar(
                "Samples-PerClass-info/Nsamples",
                self.sample_statistics[target]["n_samples"],
                target,
            )
            writer.add_scalar(
```

```
                "Samples-PerClass-info/ImprovedRatio",
                ntpt_improved / self.sample_statistics[target]["n_samples"]
                if self.sample_statistics[target]["n_samples"] != 0
                else 0,
                target,
            )
            writer.add_scalar(
                "Samples-PerClass-info/WorsenedRatio",
                ntpt_worsened / self.sample_statistics[target]["n_samples"]
                if self.sample_statistics[target]["n_samples"] != 0
                else 0,
                target,
            )

            global_stat["tpt_improved_samples"] += self.
↪sample_statistics[target][
                "tpt_improved_samples"
            ]
            global_stat["tpt_worsened_samples"] += self.
↪sample_statistics[target][
                "tpt_worsened_samples"
            ]
            global_stat["n_samples"] += self.
↪sample_statistics[target]["n_samples"]

        global_stat["ntpt_improved"] = len(global_stat["tpt_improved_samples"])
        global_stat["ntpt_worsened"] = len(global_stat["tpt_worsened_samples"])
        writer.add_scalar("Samples-Global/NImproved",␣
↪global_stat["ntpt_improved"])
        writer.add_scalar("Samples-Global/NWorsened",␣
↪global_stat["ntpt_worsened"])
        writer.add_scalar(
            "Samples-Global/ImprovedRatio",
            global_stat["ntpt_improved"] / global_stat["n_samples"]
            if global_stat["n_samples"] != 0
            else 0,
        )
        writer.add_scalar(
            "Samples-Global/WorsenedRatio",
            global_stat["ntpt_worsened"] / global_stat["n_samples"]
            if global_stat["n_samples"] != 0
            else 0,
        )

        writer.add_text(
            "statistics/perclass", json.dumps(self.sample_statistics, indent=4)
        )
```

```
        writer.add_text("statistics/global", json.dumps(global_stat, indent=4))

        final_accuracy_base = self.get_accuracy_base()
        final_accuracy_tpt = self.get_accuracy_tpt()
        final_accuracy_base_top5 = self.get_top5_accuracy_base()
        final_accuracy_tpt_top5 = self.get_top5_accuracy_tpt()

        writer.add_scalar("AccuracyTOP1/Final_Base", final_accuracy_base, 0)
        writer.add_scalar("AccuracyTOP1/Final_TPT_coop", final_accuracy_tpt, 0)

        writer.add_scalar("AccuracyTOP5/Final_Base_Top5",␣
↪final_accuracy_base_top5, 0)
        writer.add_scalar(
            "AccuracyTOP5/Final_TPT_coop_Top5", final_accuracy_tpt_top5, 0
        )

        writer.add_scalar(
            "crossEntropy/Final_Base", self.cumulative_crossEntropy_base, 0
        )
        writer.add_scalar(
            "crossEntropy/Final_TPT_coop", self.cumulative_crossEntropy_tpt, 0
        )

    def get_accuracy_base(self):
        return self.corrects_base / self.samples if self.samples > 0 else 0

    def get_accuracy_tpt(self):
        return self.corrects_tpt / self.samples if self.samples > 0 else 0

    def get_top5_accuracy_base(self):
        return self.top5_corrects_base / self.samples if self.samples > 0 else 0

    def get_top5_accuracy_tpt(self):
        return self.top5_corrects_tpt / self.samples if self.samples > 0 else 0
```

[64]:
```
from types import SimpleNamespace

def generate_run_name(args):
    if args.save:
        config_name = f"size={args.reduced_size if args.reduced_size else␣
↪'Full'}_augmenter={args.augmenter}_loss={args.loss}_naug={args.
↪n_aug}_npatch={args.n_patches}_overlap={args.overlap}_augmix={args.
↪augmix}_severity={args.severity}_lr={args.learning_rate}_spall={args.
↪selection_p_all}_sppat={args.selection_p_patch}"
        return f"{args.run_name}_{config_name}" if args.run_name else␣
↪f"{config_name}"
    else:
```

```
        return "tmp"



def parse_loss(args):
    if args.loss == "defaultTPT":
        args.loss = defaultTPT_loss
    elif args.loss == "crossentropy_soft":
        args.loss = crossentropy_soft_loss
    elif args.loss == "crossentropy_hard1":
        args.loss = crossentropy_hard1_loss
    elif args.loss == "crossentropy_hard5":
        args.loss = crossentropy_hard5_loss
    elif args.loss == "patch_loss1":
        args.loss = patch_loss1
    elif args.loss == "patch_loss2":
        args.loss = patch_loss2
    elif args.loss == "patch_loss3":
        args.loss = patch_loss3
    elif args.loss == "patch_loss4":
        args.loss = patch_loss4
    elif args.loss == "patch_loss5":
        args.loss = patch_loss5
    elif args.loss == "patch_loss6":
        args.loss = patch_loss6
    else:
        exit("Loss not valid")


def parse_augmenter(args):
    if args.augmenter == "AugmenterTPT":
        args.augmenter = AugmenterTPT(
            n_aug=args.n_aug, augmix=args.augmix, severity=args.severity
        )
    elif args.augmenter == "PatchAugmenter":
        args.augmenter = PatchAugmenter(
            n_aug=args.n_aug,
            n_patches=args.n_patches,
            overlap=args.overlap,
            augmix=args.augmix,
            severity=args.severity,
        )
    else:
        exit("Augmenter not valid")
```

3.3 Run function and parameters

[65]:
```
import argparse

def default_args():
    parser = argparse.ArgumentParser(description="TPT-deeplearning, coop and␣
 ↪TPT-next")
    parser.add_argument(
        "--imagenet_a_path",
        type=str,
        default="./imagenet-a/",
        help="Path to ImageNet-A dataset",
    )
    parser.add_argument(
        "--coop_weight_path",
        type=str,
        default="./model.pth.tar-50",
        help="Path to pre-trained CoOp weights",
    )
    parser.add_argument("--n_aug", type=int, default=63, help="Number of␣
 ↪augmentations")
    parser.add_argument(
        "--n_patches",
        type=int,
        default=0,
        help="Number of patches for patch augmenter",
    )
    parser.add_argument("--batch_size", type=int, default=1, help="Batch size")
    parser.add_argument("--arch", type=str, default="RN50", help="Model␣
 ↪architecture")
    parser.add_argument(
        "--device",
        type=str,
        default="cuda:0",
        help="Device to use, e.g., 'cuda:0' or 'cpu'",
    )
    parser.add_argument(
        "--learning_rate", type=float, default=5e-3, help="Learning rate"
    )
    parser.add_argument("--n_ctx", type=int, default=4, help="Number of context␣
 ↪tokens")
    parser.add_argument(
        "--ctx_init", type=str, default="", help="Context token initialization"
    )
    parser.add_argument(
        "--class_token_position",
        type=str,
```

```
        default="end",
        help="Class token position ('end' or 'start')",
    )
    parser.add_argument(
        "--csc", action="store_true", help="Enable class-specific context (CSC)"
    )
    parser.add_argument(
        "--run_name", type=str, default="", help="Custom name for TensorBoard␣
↪run"
    )
    parser.add_argument(
        "--augmenter",
        type=str,
        default="AugmenterTPT",
        help="Select the agumenter: AugmenterTPT, PatchAugmenter",
    )
    parser.add_argument(
        "--loss",
        type=str,
        default="defaultTPT",
        help="Select the loss: defaultTPT, patch_loss1, patch_loss2,␣
↪patch_loss3, patch_loss4, patch_loss5",
    )
    parser.add_argument("--augmix", action="store_true", help="Enable augmix")
    parser.add_argument(
        "--no-augmix", action="store_false", dest="augmix", help="Disable␣
↪augmix"
    )
    parser.add_argument("--severity", type=int, default=2, help="Augmix␣
↪severity")
    parser.add_argument("--num_workers", type=int, default=4, help="number of␣
↪workers")
    parser.add_argument(
        "--save", action="store_true", help="Enable save to TensorBoard"
    )
    parser.add_argument(
        "--no-save",
        action="store_false",
        dest="save",
        help="Disable save to TensorBoard",
    )
    parser.add_argument(
        "--reduced_size", type=int, default=None, help="number of data sample"
    )
    parser.add_argument(
        "--dataset_shuffle", action="store_true", help="Shuffle the dataset"
    )
```

```
    parser.add_argument(
        "--no-dataset_shuffle",
        action="store_false",
        dest="dataset_shuffle",
        help="Don't shuffle the dataset",
    )
    parser.add_argument("--save_imgs", action="store_false", help="Enable␣
 ↪saving images")
    parser.add_argument(
        "--no-save_imgs",
        action="store_false",
        dest="save_imgs",
        help="Disable saving images",
    )
    parser.add_argument(
        "--selection_p_all", type=float, default=0.1, help="Learning rate"
    )
    parser.add_argument(
        "--selection_p_patch", type=float, default=0.9, help="Learning rate"
    )
    parser.add_argument(
        "--overlap", type=float, default=0.0, help="patch_augmenter overlap"
    )
    parser.add_argument(
        "--alpha_exponential_weightening",
        type=float,
        default=1.0,
        help="alpha exponential weightening",
    )
    parser.add_argument("--seed", type=int, default=1337, help="seed")
    return SimpleNamespace(**{action.dest: action.default for action in parser.
 ↪_actions if action.dest != 'help'})


def print_args(args):
  print("Config:", json.dumps(vars(args), indent=4))

print_args(default_args())
```

```
Config: {
    "imagenet_a_path": "./imagenet-a/",
    "coop_weight_path": "./model.pth.tar-50",
    "n_aug": 63,
    "n_patches": 0,
    "batch_size": 1,
    "arch": "RN50",
    "device": "cuda:0",
```

```
    "learning_rate": 0.005,
    "n_ctx": 4,
    "ctx_init": "",
    "class_token_position": "end",
    "csc": false,
    "run_name": "",
    "augmenter": "AugmenterTPT",
    "loss": "defaultTPT",
    "augmix": true,
    "severity": 2,
    "num_workers": 4,
    "save": true,
    "reduced_size": null,
    "dataset_shuffle": true,
    "save_imgs": true,
    "selection_p_all": 0.1,
    "selection_p_patch": 0.9,
    "overlap": 0.0,
    "alpha_exponential_weightening": 1.0,
    "seed": 1337
}
```

[66]:
```
def run(args):
    run_name = generate_run_name(args)

    print("Config:", json.dumps(vars(args), indent=4))

    writer = SummaryWriter(log_dir=f"runs/{run_name}")
    writer.add_text("Config", json.dumps(vars(args), indent=4))
    parse_augmenter(args)
    parse_loss(args)

    set_random_seed(args.seed)

    classnames = ImageNetA.classnames
    dataset = ImageNetA(args.imagenet_a_path, transform=args.augmenter)
    args.nclasses = len(classnames)
    args.classnames = classnames
    dataloader = get_dataloader(
        dataset,
        args.batch_size,
        shuffle=args.dataset_shuffle,
        reduced_size=args.reduced_size,
        num_workers=args.num_workers,
    )
    model = get_coop(args.arch, classnames, args.device, args.n_ctx, args.
 ↪ctx_init)
```

```
    print("Use pre-trained soft prompt (CoOp) as initialization")
    pretrained_ctx = torch.load(args.coop_weight_path)["state_dict"]["ctx"]

    with torch.no_grad():
        model.prompt_learner.ctx.copy_(pretrained_ctx)
        model.prompt_learner.ctx_init_state = pretrained_ctx

    for name, param in model.named_parameters():
        if "prompt_learner" not in name:
            param.requires_grad_(False)

    model = model.to(args.device)

    trainable_param = model.prompt_learner.parameters()
    optimizer = torch.optim.AdamW(trainable_param, args.learning_rate)
    optim_state = deepcopy(optimizer.state_dict())
    scaler = torch.cuda.amp.GradScaler(init_scale=1000)

    cudnn.benchmark = True
    model.reset_classnames(classnames, args.arch)

    result = test_time_adapt_eval(
        dataloader, model, optimizer, optim_state, scaler, writer, args.device,␣
↪args
    )

    print(result)
    writer.close()
```

4 4) Dataset and Dataloader

For our project, we chose to focus on the ImageNet-A dataset because it is specifically designed to evaluate model robustness. It contains 7,500 adversarially filtered images from the 200 most challenging classes of ImageNet-1K, deliberately selected to fool standard classifiers like ResNet-50. These characteristics make it a critical benchmark for testing robustness in real-world scenarios.

Additionally, this dataset is used by the original TPT paper so we can check our implementation with the results reported on the paper and also use that as baseline. Furthermore pre-trained weights on the ImageNets dataset exist which provide a strong foundation for transfer learning.

The following code provides the PyTorch implementation of the dataloader for ImageNet-A.

[67]:
```
from torchvision.datasets import ImageFolder
from typing import Any, Tuple
from torch.utils.data import DataLoader, SubsetRandomSampler
import torch
```

```python
class ImageNetA(ImageFolder):
    folder_classname_dict = {
        "n01498041": "stingray",
        "n01531178": "goldfinch",
        "n01534433": "junco",
        "n01558993": "American robin",
        "n01580077": "jay",
        "n01614925": "bald eagle",
        "n01616318": "vulture",
        "n01631663": "newt",
        "n01641577": "American bullfrog",
        "n01669191": "box turtle",
        "n01677366": "green iguana",
        "n01687978": "agama",
        "n01694178": "chameleon",
        "n01698640": "American alligator",
        "n01735189": "garter snake",
        "n01770081": "harvestman",
        "n01770393": "scorpion",
        "n01774750": "tarantula",
        "n01784675": "centipede",
        "n01819313": "sulphur-crested cockatoo",
        "n01820546": "lorikeet",
        "n01833805": "hummingbird",
        "n01843383": "toucan",
        "n01847000": "duck",
        "n01855672": "goose",
        "n01882714": "koala",
        "n01910747": "jellyfish",
        "n01914609": "sea anemone",
        "n01924916": "flatworm",
        "n01944390": "snail",
        "n01985128": "crayfish",
        "n01986214": "hermit crab",
        "n02007558": "flamingo",
        "n02009912": "great egret",
        "n02037110": "oystercatcher",
        "n02051845": "pelican",
        "n02077923": "sea lion",
        "n02085620": "Chihuahua",
        "n02099601": "Golden Retriever",
        "n02106550": "Rottweiler",
        "n02106662": "German Shepherd Dog",
        "n02110958": "pug",
        "n02119022": "red fox",
        "n02123394": "Persian cat",
```

```
        "n02127052": "lynx",
        "n02129165": "lion",
        "n02133161": "American black bear",
        "n02137549": "mongoose",
        "n02165456": "ladybug",
        "n02174001": "rhinoceros beetle",
        "n02177972": "weevil",
        "n02190166": "fly",
        "n02206856": "bee",
        "n02219486": "ant",
        "n02226429": "grasshopper",
        "n02231487": "stick insect",
        "n02233338": "cockroach",
        "n02236044": "mantis",
        "n02259212": "leafhopper",
        "n02268443": "dragonfly",
        "n02279972": "monarch butterfly",
        "n02280649": "small white",
        "n02281787": "gossamer-winged butterfly",
        "n02317335": "starfish",
        "n02325366": "cottontail rabbit",
        "n02346627": "porcupine",
        "n02356798": "fox squirrel",
        "n02361337": "marmot",
        "n02410509": "bison",
        "n02445715": "skunk",
        "n02454379": "armadillo",
        "n02486410": "baboon",
        "n02492035": "white-headed capuchin",
        "n02504458": "African bush elephant",
        "n02655020": "pufferfish",
        "n02669723": "academic gown",
        "n02672831": "accordion",
        "n02676566": "acoustic guitar",
        "n02690373": "airliner",
        "n02701002": "ambulance",
        "n02730930": "apron",
        "n02777292": "balance beam",
        "n02782093": "balloon",
        "n02787622": "banjo",
        "n02793495": "barn",
        "n02797295": "wheelbarrow",
        "n02802426": "basketball",
        "n02814860": "lighthouse",
        "n02815834": "beaker",
        "n02837789": "bikini",
        "n02879718": "bow",
```

```
    "n02883205": "bow tie",
    "n02895154": "breastplate",
    "n02906734": "broom",
    "n02948072": "candle",
    "n02951358": "canoe",
    "n02980441": "castle",
    "n02992211": "cello",
    "n02999410": "chain",
    "n03014705": "chest",
    "n03026506": "Christmas stocking",
    "n03124043": "cowboy boot",
    "n03125729": "cradle",
    "n03187595": "rotary dial telephone",
    "n03196217": "digital clock",
    "n03223299": "doormat",
    "n03250847": "drumstick",
    "n03255030": "dumbbell",
    "n03291819": "envelope",
    "n03325584": "feather boa",
    "n03355925": "flagpole",
    "n03384352": "forklift",
    "n03388043": "fountain",
    "n03417042": "garbage truck",
    "n03443371": "goblet",
    "n03444034": "go-kart",
    "n03445924": "golf cart",
    "n03452741": "grand piano",
    "n03483316": "hair dryer",
    "n03584829": "clothes iron",
    "n03590841": "jack-o'-lantern",
    "n03594945": "jeep",
    "n03617480": "kimono",
    "n03666591": "lighter",
    "n03670208": "limousine",
    "n03717622": "manhole cover",
    "n03720891": "maraca",
    "n03721384": "marimba",
    "n03724870": "mask",
    "n03775071": "mitten",
    "n03788195": "mosque",
    "n03804744": "nail",
    "n03837869": "obelisk",
    "n03840681": "ocarina",
    "n03854065": "organ",
    "n03888257": "parachute",
    "n03891332": "parking meter",
    "n03935335": "piggy bank",
```

```
    "n03982430": "billiard table",
    "n04019541": "hockey puck",
    "n04033901": "quill",
    "n04039381": "racket",
    "n04067472": "reel",
    "n04086273": "revolver",
    "n04099969": "rocking chair",
    "n04118538": "rugby ball",
    "n04131690": "salt shaker",
    "n04133789": "sandal",
    "n04141076": "saxophone",
    "n04146614": "school bus",
    "n04147183": "schooner",
    "n04179913": "sewing machine",
    "n04208210": "shovel",
    "n04235860": "sleeping bag",
    "n04252077": "snowmobile",
    "n04252225": "snowplow",
    "n04254120": "soap dispenser",
    "n04270147": "spatula",
    "n04275548": "spider web",
    "n04310018": "steam locomotive",
    "n04317175": "stethoscope",
    "n04344873": "couch",
    "n04347754": "submarine",
    "n04355338": "sundial",
    "n04366367": "suspension bridge",
    "n04376876": "syringe",
    "n04389033": "tank",
    "n04399382": "teddy bear",
    "n04442312": "toaster",
    "n04456115": "torch",
    "n04482393": "tricycle",
    "n04507155": "umbrella",
    "n04509417": "unicycle",
    "n04532670": "viaduct",
    "n04540053": "volleyball",
    "n04554684": "washing machine",
    "n04562935": "water tower",
    "n04591713": "wine bottle",
    "n04606251": "shipwreck",
    "n07583066": "guacamole",
    "n07695742": "pretzel",
    "n07697313": "cheeseburger",
    "n07697537": "hot dog",
    "n07714990": "broccoli",
    "n07718472": "cucumber",
```

```
        "n07720875": "bell pepper",
        "n07734744": "mushroom",
        "n07749582": "lemon",
        "n07753592": "banana",
        "n07760859": "custard apple",
        "n07768694": "pomegranate",
        "n07831146": "carbonara",
        "n09229709": "bubble",
        "n09246464": "cliff",
        "n09472597": "volcano",
        "n09835506": "baseball player",
        "n11879895": "rapeseed",
        "n12057211": "yellow lady's slipper",
        "n12144580": "corn",
        "n12267677": "acorn",
    }
    classnames = list(folder_classname_dict.values())

    def __init__(self, root, transform=None, target_transform=None):
        super(ImageNetA, self).__init__(
            root, transform=transform, target_transform=target_transform
        )
        folder_class_dict = self.find_classes(root)[1]
        self.class_classname_dict = {
            folder_class_dict[folder]: classname
            for folder, classname in self.folder_classname_dict.items()
            if folder in folder_class_dict
        }
        assert set(self.classnames) == set(
            self.class_classname_dict.values()
        ), "Dataset ImageNetA is not valid"

    def __getitem__(self, index: int) -> Tuple[Any, Any]:
        img, target = super(ImageNetA, self).__getitem__(index)
        return img, target


def get_dataloader(dataset, batch_size, shuffle=False, reduced_size=None,␣
↪num_workers=4):
    if reduced_size is not None:
        indices = torch.randperm(len(dataset))[:reduced_size]
        sampler = SubsetRandomSampler(indices.tolist())
        return DataLoader(dataset, batch_size=batch_size, sampler=sampler,␣
↪num_workers=num_workers)
    else:
        return DataLoader(dataset, batch_size=batch_size, shuffle=shuffle,␣
↪num_workers=num_workers)
```

We can now extract an image from the dataset and visualize it. For example:

[68]:
```
dt = ImageNetA("./imagenet-a/")
image, label = dt[50]
class_label = ImageNetA.classnames[label]
print("the class is:", class_label)
image
```

```
the class is: stingray
```

[68]:

5 5) The model, CLIP (TPT-paper version)

To have a coherent baseline with the one of the TPT paper we decided to adopt the customized clip model used on their implementation: https://github.com/azshue/TPT.

In particular we used the **RN50** version with **n_ctx=4**, which is the number of trainable token in the prompt.

We also notice that in their implementation they used the AdamW optimizer with GradScaler which enable Automatic Mixed Precision (AMP) for faster training.

5.1 Text tokenizer

[69]:
```
!gdown "https://drive.google.com/uc?id=1tD55BFUxGVMd7U5dFyYCHNLIS1Xe2nv8"
```

```
Downloading…
From: https://drive.google.com/uc?id=1tD55BFUxGVMd7U5dFyYCHNLIS1Xe2nv8
To: /content/bpe_simple_vocab_16e6.txt.gz
   0% 0.00/1.36M [00:00<?, ?B/s] 100% 1.36M/1.36M [00:00<00:00, 12.8MB/s] 100%
1.36M/1.36M [00:00<00:00, 12.7MB/s]
```

[70]:
```
import gzip
import html
import os
from functools import lru_cache

import ftfy
import regex as re


@lru_cache()
def default_bpe():
    return os.path.join(os.getcwd(), "bpe_simple_vocab_16e6.txt.gz")
    #return os.path.join(os.path.dirname(os.path.abspath(__file__)),␣
 ↪"bpe_simple_vocab_16e6.txt.gz")


@lru_cache()
def bytes_to_unicode():
    """
    Returns list of utf-8 byte and a corresponding list of unicode strings.
    The reversible bpe codes work on unicode strings.
    This means you need a large # of unicode characters in your vocab if you␣
 ↪want to avoid UNKs.
    When you're at something like a 10B token dataset you end up needing around␣
 ↪5K for decent coverage.
    This is a signficant percentage of your normal, say, 32K bpe vocab.
    To avoid that, we want lookup tables between utf-8 bytes and unicode␣
 ↪strings.
    And avoids mapping to whitespace/control characters the bpe code barfs on.
    """
    bs = list(range(ord("!"), ord("~")+1))+list(range(ord("¡"),␣
 ↪ord("¬")+1))+list(range(ord("®"), ord("ÿ")+1))
```

```
    cs = bs[:]
    n = 0
    for b in range(2**8):
        if b not in bs:
            bs.append(b)
            cs.append(2**8+n)
            n += 1
    cs = [chr(n) for n in cs]
    return dict(zip(bs, cs))


def get_pairs(word):
    """Return set of symbol pairs in a word.
    Word is represented as tuple of symbols (symbols being variable-length␣
 ↪strings).
    """
    pairs = set()
    prev_char = word[0]
    for char in word[1:]:
        pairs.add((prev_char, char))
        prev_char = char
    return pairs


def basic_clean(text):
    text = ftfy.fix_text(text)
    text = html.unescape(html.unescape(text))
    return text.strip()


def whitespace_clean(text):
    text = re.sub(r'\s+', ' ', text)
    text = text.strip()
    return text


class SimpleTokenizer(object):
    def __init__(self, bpe_path: str = default_bpe()):
        self.byte_encoder = bytes_to_unicode()
        self.byte_decoder = {v: k for k, v in self.byte_encoder.items()}
        merges = gzip.open(bpe_path).read().decode("utf-8").split('\n')
        merges = merges[1:49152-256-2+1]
        merges = [tuple(merge.split()) for merge in merges]
        vocab = list(bytes_to_unicode().values())
        vocab = vocab + [v+'</w>' for v in vocab]
        for merge in merges:
            vocab.append(''.join(merge))
```

```
        vocab.extend(['<|startoftext|>', '<|endoftext|>'])
        self.encoder = dict(zip(vocab, range(len(vocab))))
        self.decoder = {v: k for k, v in self.encoder.items()}
        self.bpe_ranks = dict(zip(merges, range(len(merges))))
        self.cache = {'<|startoftext|>': '<|startoftext|>', '<|endoftext|>':␣
↪'<|endoftext|>'}
        self.pat = re.
↪compile(r"""<\|startoftext\|>|<\|endoftext\|>|'s|'t|'re|'ve|'m|'ll|'d|[\p{L}]+|[\p{N}]|[^\s
↪re.IGNORECASE)

    def bpe(self, token):
        if token in self.cache:
            return self.cache[token]
        word = tuple(token[:-1]) + ( token[-1] + '</w>',)
        pairs = get_pairs(word)

        if not pairs:
            return token+'</w>'

        while True:
            bigram = min(pairs, key = lambda pair: self.bpe_ranks.get(pair,␣
↪float('inf')))
            if bigram not in self.bpe_ranks:
                break
            first, second = bigram
            new_word = []
            i = 0
            while i < len(word):
                try:
                    j = word.index(first, i)
                    new_word.extend(word[i:j])
                    i = j
                except:
                    new_word.extend(word[i:])
                    break

                if word[i] == first and i < len(word)-1 and word[i+1] == second:
                    new_word.append(first+second)
                    i += 2
                else:
                    new_word.append(word[i])
                    i += 1
            new_word = tuple(new_word)
            word = new_word
            if len(word) == 1:
                break
            else:
```

```
                pairs = get_pairs(word)
        word = ' '.join(word)
        self.cache[token] = word
        return word

    def encode(self, text):
        bpe_tokens = []
        text = whitespace_clean(basic_clean(text)).lower()
        for token in re.findall(self.pat, text):
            token = ''.join(self.byte_encoder[b] for b in token.encode('utf-8'))
            bpe_tokens.extend(self.encoder[bpe_token] for bpe_token in self.
↪bpe(token).split(' '))
        return bpe_tokens

    def decode(self, tokens):
        text = ''.join([self.decoder[token] for token in tokens])
        text = bytearray([self.byte_decoder[c] for c in text]).decode('utf-8',␣
↪errors="replace").replace('</w>', ' ')
        return text
```

5.2 Model

[71]:
```
from collections import OrderedDict
from typing import Tuple, Union

import numpy as np
import torch
import torch.nn.functional as F
from torch import nn


class Bottleneck(nn.Module):
    expansion = 4

    def __init__(self, inplanes, planes, stride=1):
        super().__init__()

        # all conv layers have stride 1. an avgpool is performed after the␣
↪second convolution when stride > 1
        self.conv1 = nn.Conv2d(inplanes, planes, 1, bias=False)
        self.bn1 = nn.BatchNorm2d(planes)
        self.relu1 = nn.ReLU(inplace=True)

        self.conv2 = nn.Conv2d(planes, planes, 3, padding=1, bias=False)
        self.bn2 = nn.BatchNorm2d(planes)
        self.relu2 = nn.ReLU(inplace=True)
```

```
        self.avgpool = nn.AvgPool2d(stride) if stride > 1 else nn.Identity()

        self.conv3 = nn.Conv2d(planes, planes * self.expansion, 1, bias=False)
        self.bn3 = nn.BatchNorm2d(planes * self.expansion)
        self.relu3 = nn.ReLU(inplace=True)

        self.downsample = None
        self.stride = stride

        if stride > 1 or inplanes != planes * Bottleneck.expansion:
            # downsampling layer is prepended with an avgpool, and the␣
 ↪subsequent convolution has stride 1
            self.downsample = nn.Sequential(OrderedDict([
                ("-1", nn.AvgPool2d(stride)),
                ("0", nn.Conv2d(inplanes, planes * self.expansion, 1, stride=1,␣
 ↪bias=False)),
                ("1", nn.BatchNorm2d(planes * self.expansion))
            ]))

    def forward(self, x: torch.Tensor):
        identity = x

        out = self.relu1(self.bn1(self.conv1(x)))
        out = self.relu2(self.bn2(self.conv2(out)))
        out = self.avgpool(out)
        out = self.bn3(self.conv3(out))

        if self.downsample is not None:
            identity = self.downsample(x)

        out += identity
        out = self.relu3(out)
        return out


class AttentionPool2d(nn.Module):
    def __init__(self, spacial_dim: int, embed_dim: int, num_heads: int,␣
 ↪output_dim: int = None):
        super().__init__()
        self.positional_embedding = nn.Parameter(torch.randn(spacial_dim ** 2 +␣
 ↪1, embed_dim) / embed_dim ** 0.5)
        self.k_proj = nn.Linear(embed_dim, embed_dim)
        self.q_proj = nn.Linear(embed_dim, embed_dim)
        self.v_proj = nn.Linear(embed_dim, embed_dim)
        self.c_proj = nn.Linear(embed_dim, output_dim or embed_dim)
        self.num_heads = num_heads
```

```
    def forward(self, x):
        x = x.flatten(start_dim=2).permute(2, 0, 1)  # NCHW -> (HW)NC
        x = torch.cat([x.mean(dim=0, keepdim=True), x], dim=0)  # (HW+1)NC
        x = x + self.positional_embedding[:, None, :].to(x.dtype)  # (HW+1)NC
        x, _ = F.multi_head_attention_forward(
            query=x[:1], key=x, value=x,
            embed_dim_to_check=x.shape[-1],
            num_heads=self.num_heads,
            q_proj_weight=self.q_proj.weight,
            k_proj_weight=self.k_proj.weight,
            v_proj_weight=self.v_proj.weight,
            in_proj_weight=None,
            in_proj_bias=torch.cat([self.q_proj.bias, self.k_proj.bias, self.
 ↪v_proj.bias]),
            bias_k=None,
            bias_v=None,
            add_zero_attn=False,
            dropout_p=0,
            out_proj_weight=self.c_proj.weight,
            out_proj_bias=self.c_proj.bias,
            use_separate_proj_weight=True,
            training=self.training,
            need_weights=False
        )
        return x.squeeze(0)


class ModifiedResNet(nn.Module):
    """
    A ResNet class that is similar to torchvision's but contains the following␣
 ↪changes:
    - There are now 3 "stem" convolutions as opposed to 1, with an average pool␣
 ↪instead of a max pool.
    - Performs anti-aliasing strided convolutions, where an avgpool is␣
 ↪prepended to convolutions with stride > 1
    - The final pooling layer is a QKV attention instead of an average pool
    """

    def __init__(self, layers, output_dim, heads, input_resolution=224,␣
 ↪width=64):
        super().__init__()
        self.output_dim = output_dim
        self.input_resolution = input_resolution

        # the 3-layer stem
        self.conv1 = nn.Conv2d(3, width // 2, kernel_size=3, stride=2,␣
 ↪padding=1, bias=False)
```

```
        self.bn1 = nn.BatchNorm2d(width // 2)
        self.relu1 = nn.ReLU(inplace=True)
        self.conv2 = nn.Conv2d(width // 2, width // 2, kernel_size=3,␣
↪padding=1, bias=False)
        self.bn2 = nn.BatchNorm2d(width // 2)
        self.relu2 = nn.ReLU(inplace=True)
        self.conv3 = nn.Conv2d(width // 2, width, kernel_size=3, padding=1,␣
↪bias=False)
        self.bn3 = nn.BatchNorm2d(width)
        self.relu3 = nn.ReLU(inplace=True)
        self.avgpool = nn.AvgPool2d(2)

        # residual layers
        self._inplanes = width  # this is a *mutable* variable used during␣
↪construction
        self.layer1 = self._make_layer(width, layers[0])
        self.layer2 = self._make_layer(width * 2, layers[1], stride=2)
        self.layer3 = self._make_layer(width * 4, layers[2], stride=2)
        self.layer4 = self._make_layer(width * 8, layers[3], stride=2)

        embed_dim = width * 32  # the ResNet feature dimension
        self.attnpool = AttentionPool2d(input_resolution // 32, embed_dim,␣
↪heads, output_dim)

    def _make_layer(self, planes, blocks, stride=1):
        layers = [Bottleneck(self._inplanes, planes, stride)]

        self._inplanes = planes * Bottleneck.expansion
        for _ in range(1, blocks):
            layers.append(Bottleneck(self._inplanes, planes))

        return nn.Sequential(*layers)

    def forward(self, x):
        def stem(x):
            x = self.relu1(self.bn1(self.conv1(x)))
            x = self.relu2(self.bn2(self.conv2(x)))
            x = self.relu3(self.bn3(self.conv3(x)))
            x = self.avgpool(x)
            return x

        x = x.type(self.conv1.weight.dtype)
        x = stem(x)
        x = self.layer1(x)
        x = self.layer2(x)
        x = self.layer3(x)
        x = self.layer4(x)
```

```
        x = self.attnpool(x)

        return x


class LayerNorm(nn.LayerNorm):
    """Subclass torch's LayerNorm to handle fp16."""

    def forward(self, x: torch.Tensor):
        orig_type = x.dtype
        ret = super().forward(x.type(torch.float32))
        return ret.type(orig_type)


class QuickGELU(nn.Module):
    def forward(self, x: torch.Tensor):
        return x * torch.sigmoid(1.702 * x)


class ResidualAttentionBlock(nn.Module):
    def __init__(self, d_model: int, n_head: int, attn_mask: torch.Tensor =␣
 ↪None):
        super().__init__()

        self.attn = nn.MultiheadAttention(d_model, n_head)
        self.ln_1 = LayerNorm(d_model)
        self.mlp = nn.Sequential(OrderedDict([
            ("c_fc", nn.Linear(d_model, d_model * 4)),
            ("gelu", QuickGELU()),
            ("c_proj", nn.Linear(d_model * 4, d_model))
        ]))
        self.ln_2 = LayerNorm(d_model)
        self.attn_mask = attn_mask

    def attention(self, x: torch.Tensor):
        self.attn_mask = self.attn_mask.to(dtype=x.dtype, device=x.device) if␣
 ↪self.attn_mask is not None else None
        return self.attn(x, x, x, need_weights=False, attn_mask=self.
 ↪attn_mask)[0]

    def forward(self, x: torch.Tensor):
        x = x + self.attention(self.ln_1(x))
        x = x + self.mlp(self.ln_2(x))
        return x


class Transformer(nn.Module):
```

```
    def __init__(self, width: int, layers: int, heads: int, attn_mask: torch.
↪Tensor = None):
        super().__init__()
        self.width = width
        self.layers = layers
        self.resblocks = nn.Sequential(*[ResidualAttentionBlock(width, heads,␣
↪attn_mask) for _ in range(layers)])

    def forward(self, x: torch.Tensor):
        return self.resblocks(x)


class VisionTransformer(nn.Module):
    def __init__(self, input_resolution: int, patch_size: int, width: int,␣
↪layers: int, heads: int, output_dim: int):
        super().__init__()
        self.input_resolution = input_resolution
        self.output_dim = output_dim
        self.conv1 = nn.Conv2d(in_channels=3, out_channels=width,␣
↪kernel_size=patch_size, stride=patch_size, bias=False)

        scale = width ** -0.5
        self.class_embedding = nn.Parameter(scale * torch.randn(width))
        self.positional_embedding = nn.Parameter(scale * torch.
↪randn((input_resolution // patch_size) ** 2 + 1, width))
        self.ln_pre = LayerNorm(width)

        self.transformer = Transformer(width, layers, heads)

        self.ln_post = LayerNorm(width)
        self.proj = nn.Parameter(scale * torch.randn(width, output_dim))

    def forward(self, x: torch.Tensor):
        x = self.conv1(x)  # shape = [*, width, grid, grid]
        x = x.reshape(x.shape[0], x.shape[1], -1)  # shape = [*, width, grid **␣
↪2]
        x = x.permute(0, 2, 1)  # shape = [*, grid ** 2, width]
        x = torch.cat([self.class_embedding.to(x.dtype) + torch.zeros(x.
↪shape[0], 1, x.shape[-1], dtype=x.dtype, device=x.device), x], dim=1)  #␣
↪shape = [*, grid ** 2 + 1, width]
        x = x + self.positional_embedding.to(x.dtype)
        x = self.ln_pre(x)

        x = x.permute(1, 0, 2)  # NLD -> LND
        x = self.transformer(x)
        x = x.permute(1, 0, 2)  # LND -> NLD
```

```
        x = self.ln_post(x[:, 0, :])

        if self.proj is not None:
            x = x @ self.proj

        return x


class CLIP(nn.Module):
    def __init__(self,
                 embed_dim: int,
                 # vision
                 image_resolution: int,
                 vision_layers: Union[Tuple[int, int, int, int], int],
                 vision_width: int,
                 vision_patch_size: int,
                 # text
                 context_length: int,
                 vocab_size: int,
                 transformer_width: int,
                 transformer_heads: int,
                 transformer_layers: int
                 ):
        super().__init__()

        self.context_length = context_length

        if isinstance(vision_layers, (tuple, list)):
            vision_heads = vision_width * 32 // 64
            self.visual = ModifiedResNet(
                layers=vision_layers,
                output_dim=embed_dim,
                heads=vision_heads,
                input_resolution=image_resolution,
                width=vision_width
            )
        else:
            vision_heads = vision_width // 64
            self.visual = VisionTransformer(
                input_resolution=image_resolution,
                patch_size=vision_patch_size,
                width=vision_width,
                layers=vision_layers,
                heads=vision_heads,
                output_dim=embed_dim
            )
```

```
        self.transformer = Transformer(
            width=transformer_width,
            layers=transformer_layers,
            heads=transformer_heads,
            attn_mask=self.build_attention_mask()
        )

        self.vocab_size = vocab_size
        self.token_embedding = nn.Embedding(vocab_size, transformer_width)
        self.positional_embedding = nn.Parameter(torch.empty(self.
↪context_length, transformer_width))
        self.ln_final = LayerNorm(transformer_width)

        self.text_projection = nn.Parameter(torch.empty(transformer_width,␣
↪embed_dim))
        self.logit_scale = nn.Parameter(torch.ones([]) * np.log(1 / 0.07))

        self.initialize_parameters()

    def initialize_parameters(self):
        nn.init.normal_(self.token_embedding.weight, std=0.02)
        nn.init.normal_(self.positional_embedding, std=0.01)

        if isinstance(self.visual, ModifiedResNet):
            if self.visual.attnpool is not None:
                std = self.visual.attnpool.c_proj.in_features ** -0.5
                nn.init.normal_(self.visual.attnpool.q_proj.weight, std=std)
                nn.init.normal_(self.visual.attnpool.k_proj.weight, std=std)
                nn.init.normal_(self.visual.attnpool.v_proj.weight, std=std)
                nn.init.normal_(self.visual.attnpool.c_proj.weight, std=std)

            for resnet_block in [self.visual.layer1, self.visual.layer2, self.
↪visual.layer3, self.visual.layer4]:
                for name, param in resnet_block.named_parameters():
                    if name.endswith("bn3.weight"):
                        nn.init.zeros_(param)

        proj_std = (self.transformer.width ** -0.5) * ((2 * self.transformer.
↪layers) ** -0.5)
        attn_std = self.transformer.width ** -0.5
        fc_std = (2 * self.transformer.width) ** -0.5
        for block in self.transformer.resblocks:
            nn.init.normal_(block.attn.in_proj_weight, std=attn_std)
            nn.init.normal_(block.attn.out_proj.weight, std=proj_std)
            nn.init.normal_(block.mlp.c_fc.weight, std=fc_std)
            nn.init.normal_(block.mlp.c_proj.weight, std=proj_std)
```

```
        if self.text_projection is not None:
            nn.init.normal_(self.text_projection, std=self.transformer.width **␣
↪-0.5)

    def build_attention_mask(self):
        # lazily create causal attention mask, with full attention between the␣
↪vision tokens
        # pytorch uses additive attention mask; fill with -inf
        mask = torch.empty(self.context_length, self.context_length)
        mask.fill_(float("-inf"))
        mask.triu_(1)  # zero out the lower diagonal
        return mask

    @property
    def dtype(self):
        return self.visual.conv1.weight.dtype

    def encode_image(self, image):
        return self.visual(image.type(self.dtype))

    def encode_text(self, text):
        x = self.token_embedding(text).type(self.dtype)  # [batch_size, n_ctx,␣
↪d_model]

        x = x + self.positional_embedding.type(self.dtype)
        x = x.permute(1, 0, 2)  # NLD -> LND
        x = self.transformer(x)
        x = x.permute(1, 0, 2)  # LND -> NLD
        x = self.ln_final(x).type(self.dtype)

        # x.shape = [batch_size, n_ctx, transformer.width]
        # take features from the eot embedding (eot_token is the highest number␣
↪in each sequence)
        x = x[torch.arange(x.shape[0]), text.argmax(dim=-1)] @ self.
↪text_projection

        return x

    def forward(self, image, text):
        image_features = self.encode_image(image)
        text_features = self.encode_text(text)

        # normalized features
        image_features = image_features / image_features.norm(dim=1,␣
↪keepdim=True)
```

```
        text_features = text_features / text_features.norm(dim=1, keepdim=True)

        # cosine similarity as logits
        logit_scale = self.logit_scale.exp()
        logits_per_image = logit_scale * image_features @ text_features.t()
        logits_per_text = logits_per_image.t()

        # shape = [global_batch_size, global_batch_size]
        return logits_per_image, logits_per_text


def convert_weights(model: nn.Module):
    """Convert applicable model parameters to fp16"""

    def _convert_weights_to_fp16(l):
        if isinstance(l, (nn.Conv1d, nn.Conv2d, nn.Linear)):
            l.weight.data = l.weight.data.half()
            if l.bias is not None:
                l.bias.data = l.bias.data.half()

        if isinstance(l, nn.MultiheadAttention):
            for attr in [*[f"{s}_proj_weight" for s in ["in", "q", "k", "v"]],␣
 ↪"in_proj_bias", "bias_k", "bias_v"]:
                tensor = getattr(l, attr)
                if tensor is not None:
                    tensor.data = tensor.data.half()

        for name in ["text_projection", "proj"]:
            if hasattr(l, name):
                attr = getattr(l, name)
                if attr is not None:
                    attr.data = attr.data.half()

    model.apply(_convert_weights_to_fp16)


def build_model(state_dict: dict):
    vit = "visual.proj" in state_dict

    if vit:
        vision_width = state_dict["visual.conv1.weight"].shape[0]
        vision_layers = len([k for k in state_dict.keys() if k.
 ↪startswith("visual.") and k.endswith(".attn.in_proj_weight")])
        vision_patch_size = state_dict["visual.conv1.weight"].shape[-1]
        grid_size = round((state_dict["visual.positional_embedding"].shape[0] -␣
 ↪1) ** 0.5)
        image_resolution = vision_patch_size * grid_size
```

```
    else:
        counts: list = [len(set(k.split(".")[2] for k in state_dict if k.
↪startswith(f"visual.layer{b}"))) for b in [1, 2, 3, 4]]
        vision_layers = tuple(counts)
        vision_width = state_dict["visual.layer1.0.conv1.weight"].shape[0]
        output_width = round((state_dict["visual.attnpool.
↪positional_embedding"].shape[0] - 1) ** 0.5)
        vision_patch_size = None
        assert output_width ** 2 + 1 == state_dict["visual.attnpool.
↪positional_embedding"].shape[0]
        image_resolution = output_width * 32

    embed_dim = state_dict["text_projection"].shape[1]
    context_length = state_dict["positional_embedding"].shape[0]
    vocab_size = state_dict["token_embedding.weight"].shape[0]
    transformer_width = state_dict["ln_final.weight"].shape[0]
    transformer_heads = transformer_width // 64
    transformer_layers = len(set(k.split(".")[2] for k in state_dict if k.
↪startswith("transformer.resblocks")))

    model = CLIP(
        embed_dim,
        image_resolution, vision_layers, vision_width, vision_patch_size,
        context_length, vocab_size, transformer_width, transformer_heads,␣
↪transformer_layers
    )

    for key in ["input_resolution", "context_length", "vocab_size"]:
        if key in state_dict:
            del state_dict[key]

    # convert_weights(model)
    model.load_state_dict(state_dict)
    del state_dict
    torch.cuda.empty_cache()
    return model.eval()
```

5.3 Custom Clip

[72]:
```
from typing import Tuple

import torch
import torch.nn as nn

#from .clip import load, tokenize
#from .simple_tokenizer import SimpleTokenizer as _Tokenizer
```

```
_Tokenizer = SimpleTokenizer
_tokenizer = _Tokenizer()

DOWNLOAD_ROOT = ".cache/clip"


class TextEncoder(nn.Module):
    def __init__(self, clip_model):
        super().__init__()
        self.transformer = clip_model.transformer
        self.positional_embedding = clip_model.positional_embedding
        self.ln_final = clip_model.ln_final
        self.text_projection = clip_model.text_projection
        self.dtype = clip_model.dtype

    def forward(self, prompts, tokenized_prompts):
        x = prompts + self.positional_embedding.type(self.dtype)
        x = x.permute(1, 0, 2)  # NLD -> LND
        x = self.transformer(x)
        x = x.permute(1, 0, 2)  # LND -> NLD
        x = self.ln_final(x).type(self.dtype)

        # x.shape = [batch_size, n_ctx, transformer.width]
        # take features from the eot embedding (eot_token is the highest number␣
 ↪in each sequence)
        x = (
            x[torch.arange(x.shape[0]), tokenized_prompts.argmax(dim=-1)]
            @ self.text_projection
        )

        return x


class PromptLearner(nn.Module):
    def __init__(
        self,
        clip_model,
        classnames,
        batch_size=None,
        n_ctx=16,
        ctx_init=None,
        ctx_position="end",
        learned_cls=False,
    ):
        super().__init__()
        n_cls = len(classnames)
        self.learned_cls = learned_cls
```

```
        dtype = clip_model.dtype
        self.dtype = dtype
        self.device = clip_model.visual.conv1.weight.device
        ctx_dim = clip_model.ln_final.weight.shape[0]
        self.ctx_dim = ctx_dim
        self.batch_size = batch_size

        # self.ctx, prompt_prefix = self.reset_prompt(ctx_dim, ctx_init,␣
↪clip_model)

        if ctx_init:
            # use given words to initialize context vectors
            print("Initializing the contect with given words: [{}]".
↪format(ctx_init))
            ctx_init = ctx_init.replace("_", " ")
            if "[CLS]" in ctx_init:
                ctx_list = ctx_init.split(" ")
                split_idx = ctx_list.index("[CLS]")
                ctx_init = ctx_init.replace("[CLS] ", "")
                ctx_position = "middle"
            else:
                split_idx = None
            self.split_idx = split_idx
            n_ctx = len(ctx_init.split(" "))
            prompt = tokenize(ctx_init).to(self.device)
            with torch.no_grad():
                embedding = clip_model.token_embedding(prompt).type(dtype)
            ctx_vectors = embedding[0, 1 : 1 + n_ctx, :]
            prompt_prefix = ctx_init
        else:
            print("Random initialization: initializing a generic context")
            ctx_vectors = torch.empty(n_ctx, ctx_dim, dtype=dtype)
            nn.init.normal_(ctx_vectors, std=0.02)
            prompt_prefix = " ".join(["X"] * n_ctx)

        self.prompt_prefix = prompt_prefix

        print(f'Initial context: "{prompt_prefix}"')
        print(f"Number of context words (tokens): {n_ctx}")

        # batch-wise prompt tuning for test-time adaptation
        if self.batch_size is not None:
            ctx_vectors = ctx_vectors.repeat(batch_size, 1, 1)  # (N, L, D)
        self.ctx_init_state = ctx_vectors.detach().clone()
        self.ctx = nn.Parameter(ctx_vectors)  # to be optimized

        if not self.learned_cls:
```

```
            classnames = [name.replace("_", " ") for name in classnames]
            name_lens = [len(_tokenizer.encode(name)) for name in classnames]
            prompts = [prompt_prefix + " " + name + "." for name in classnames]
        else:
            print("Random initialization: initializing a learnable class token")
            cls_vectors = torch.empty(
                n_cls, 1, ctx_dim, dtype=dtype
            )  # assume each learnable cls_token is only 1 word
            nn.init.normal_(cls_vectors, std=0.02)
            cls_token = "X"
            name_lens = [1 for _ in classnames]
            prompts = [prompt_prefix + " " + cls_token + "." for _ in␣
↪classnames]

            self.cls_init_state = cls_vectors.detach().clone()
            self.cls = nn.Parameter(cls_vectors)  # to be optimized

        tokenized_prompts = torch.cat([tokenize(p) for p in prompts]).to(self.
↪device)
        with torch.no_grad():
            embedding = clip_model.token_embedding(tokenized_prompts).
↪type(dtype)

        # These token vectors will be saved when in save_model(),
        # but they should be ignored in load_model() as we want to use
        # those computed using the current class names
        self.register_buffer("token_prefix", embedding[:, :1, :])  # SOS
        if self.learned_cls:
            self.register_buffer(
                "token_suffix", embedding[:, 1 + n_ctx + 1 :, :]
            )  # ..., EOS
        else:
            self.register_buffer(
                "token_suffix", embedding[:, 1 + n_ctx :, :]
            )  # CLS, EOS

        self.ctx_init = ctx_init
        self.tokenized_prompts = tokenized_prompts  # torch.Tensor
        self.name_lens = name_lens
        self.class_token_position = ctx_position
        self.n_cls = n_cls
        self.n_ctx = n_ctx
        self.classnames = classnames

    def reset(self):
        ctx_vectors = self.ctx_init_state
        self.ctx.copy_(ctx_vectors)  # to be optimized
```

```
        if self.learned_cls:
            cls_vectors = self.cls_init_state
            self.cls.copy_(cls_vectors)

    def reset_classnames(self, classnames, arch):
        self.n_cls = len(classnames)
        if not self.learned_cls:
            classnames = [name.replace("_", " ") for name in classnames]
            name_lens = [len(_tokenizer.encode(name)) for name in classnames]
            prompts = [self.prompt_prefix + " " + name + "." for name in␣
↪classnames]
        else:
            cls_vectors = torch.empty(
                self.n_cls, 1, self.ctx_dim, dtype=self.dtype
            )  # assume each learnable cls_token is only 1 word
            nn.init.normal_(cls_vectors, std=0.02)
            cls_token = "X"
            name_lens = [1 for _ in classnames]
            prompts = [self.prompt_prefix + " " + cls_token + "." for _ in␣
↪classnames]
            # TODO: re-init the cls parameters
            # self.cls = nn.Parameter(cls_vectors) # to be optimized
            self.cls_init_state = cls_vectors.detach().clone()
        tokenized_prompts = torch.cat([tokenize(p) for p in prompts]).to(self.
↪device)

        clip, _, _ = load(arch, device=self.device, download_root=DOWNLOAD_ROOT)

        with torch.no_grad():
            embedding = clip.token_embedding(tokenized_prompts).type(self.dtype)

        self.token_prefix = embedding[:, :1, :]
        self.token_suffix = embedding[:, 1 + self.n_ctx :, :]  # CLS, EOS

        self.name_lens = name_lens
        self.tokenized_prompts = tokenized_prompts
        self.classnames = classnames

    def forward(self, init=None):
        # the init will be used when computing CLIP directional loss
        if init is not None:
            ctx = init
        else:
            ctx = self.ctx
        if ctx.dim() == 2:
            ctx = ctx.unsqueeze(0).expand(self.n_cls, -1, -1)
        elif not ctx.size()[0] == self.n_cls:
```

```
            ctx = ctx.unsqueeze(1).expand(-1, self.n_cls, -1, -1)

        prefix = self.token_prefix
        suffix = self.token_suffix
        if self.batch_size is not None:
            # This way only works for single-gpu setting (could pass batch size␣
↪as an argument for forward())
            prefix = prefix.repeat(self.batch_size, 1, 1, 1)
            suffix = suffix.repeat(self.batch_size, 1, 1, 1)

        if self.learned_cls:
            assert self.class_token_position == "end"
        if self.class_token_position == "end":
            if self.learned_cls:
                cls = self.cls
                prompts = torch.cat(
                    [
                        prefix,  # (n_cls, 1, dim)
                        ctx,  # (n_cls, n_ctx, dim)
                        cls,  # (n_cls, 1, dim)
                        suffix,  # (n_cls, *, dim)
                    ],
                    dim=-2,
                )
            else:
                prompts = torch.cat(
                    [
                        prefix,  # (n_cls, 1, dim)
                        ctx,  # (n_cls, n_ctx, dim)
                        suffix,  # (n_cls, *, dim)
                    ],
                    dim=-2,
                )
        elif self.class_token_position == "middle":
            # TODO: to work with a batch of prompts
            if self.split_idx is not None:
                half_n_ctx = (
                    self.split_idx
                )  # split the ctx at the position of [CLS] in `ctx_init`
            else:
                half_n_ctx = self.n_ctx // 2
            prompts = []
            for i in range(self.n_cls):
                name_len = self.name_lens[i]
                prefix_i = prefix[i : i + 1, :, :]
                class_i = suffix[i : i + 1, :name_len, :]
                suffix_i = suffix[i : i + 1, name_len:, :]
```

```
                ctx_i_half1 = ctx[i : i + 1, :half_n_ctx, :]
                ctx_i_half2 = ctx[i : i + 1, half_n_ctx:, :]
                prompt = torch.cat(
                    [
                        prefix_i,  # (1, 1, dim)
                        ctx_i_half1,  # (1, n_ctx//2, dim)
                        class_i,  # (1, name_len, dim)
                        ctx_i_half2,  # (1, n_ctx//2, dim)
                        suffix_i,  # (1, *, dim)
                    ],
                    dim=1,
                )
                prompts.append(prompt)
            prompts = torch.cat(prompts, dim=0)

        elif self.class_token_position == "front":
            prompts = []
            for i in range(self.n_cls):
                name_len = self.name_lens[i]
                prefix_i = prefix[i : i + 1, :, :]
                class_i = suffix[i : i + 1, :name_len, :]
                suffix_i = suffix[i : i + 1, name_len:, :]
                ctx_i = ctx[i : i + 1, :, :]
                prompt = torch.cat(
                    [
                        prefix_i,  # (1, 1, dim)
                        class_i,  # (1, name_len, dim)
                        ctx_i,  # (1, n_ctx, dim)
                        suffix_i,  # (1, *, dim)
                    ],
                    dim=1,
                )
                prompts.append(prompt)
            prompts = torch.cat(prompts, dim=0)

        else:
            raise ValueError

        return prompts


class ClipTestTimeTuning(nn.Module):
    def __init__(
        self,
        device,
        classnames,
        batch_size,
```

```
        criterion="cosine",
        arch="ViT-L/14",
        n_ctx=16,
        ctx_init=None,
        ctx_position="end",
        learned_cls=False,
    ):
        super(ClipTestTimeTuning, self).__init__()
        clip, _, _ = load(arch, device=device, download_root=DOWNLOAD_ROOT)
        self.image_encoder = clip.visual
        self.text_encoder = TextEncoder(clip)
        self.logit_scale = clip.logit_scale.data
        # prompt tuning
        self.prompt_learner = PromptLearner(
            clip, classnames, batch_size, n_ctx, ctx_init, ctx_position,␣
↪learned_cls
        )
        self.criterion = criterion

    @property
    def dtype(self):
        return self.image_encoder.conv1.weight.dtype

    # restore the initial state of the prompt_learner (tunable prompt)
    def reset(self):
        self.prompt_learner.reset()

    def reset_classnames(self, classnames, arch):
        self.prompt_learner.reset_classnames(classnames, arch)

    def get_text_features(self):
        text_features = []
        prompts = self.prompt_learner()
        tokenized_prompts = self.prompt_learner.tokenized_prompts
        t_features = self.text_encoder(prompts, tokenized_prompts)
        text_features.append(t_features / t_features.norm(dim=-1, keepdim=True))
        text_features = torch.stack(text_features, dim=0)

        return torch.mean(text_features, dim=0)

    def inference(self, image):
        with torch.no_grad():
            image_features = self.image_encoder(image.type(self.dtype))

        text_features = self.get_text_features()
        image_features = image_features / image_features.norm(dim=-1,␣
↪keepdim=True)
```

```
        logit_scale = self.logit_scale.exp()
        logits = logit_scale * image_features @ text_features.t()

        return logits

    def forward(self, input):
        if isinstance(input, Tuple):
            view_0, view_1, view_2 = input
            return self.contrast_prompt_tuning(view_0, view_1, view_2)
        elif len(input.size()) == 2:
            return self.directional_prompt_tuning(input)
        else:
            return self.inference(input)


def get_coop(clip_arch, classnames, device, n_ctx, ctx_init, learned_cls=False):
    model = ClipTestTimeTuning(
        device,
        classnames,
        None,
        arch=clip_arch,
        n_ctx=n_ctx,
        ctx_init=ctx_init,
        learned_cls=learned_cls,
    )

    return model
```

5.4 CLIP

[73]:
```
import hashlib
import os
import urllib
import warnings
from typing import Any, Union, List
from pkg_resources import packaging

import torch
from PIL import Image
from torchvision.transforms import Compose, Resize, CenterCrop, ToTensor,␣
 ↪Normalize
from tqdm import tqdm

#from .model import build_model
#from .simple_tokenizer import SimpleTokenizer as _Tokenizer
```

```
_Tokenizer = SimpleTokenizer

try:
    from torchvision.transforms import InterpolationMode
    BICUBIC = InterpolationMode.BICUBIC
except ImportError:
    BICUBIC = Image.BICUBIC


if packaging.version.parse(torch.__version__) < packaging.version.parse("1.7.
 ↪1"):
    warnings.warn("PyTorch version 1.7.1 or higher is recommended")


__all__ = ["available_models", "load", "tokenize"]
_tokenizer = _Tokenizer()

_MODELS = {
    "RN50": "https://openaipublic.azureedge.net/clip/models/
 ↪afeb0e10f9e5a86da6080e35cf09123aca3b358a0c3e3b6c78a7b63bc04b6762/RN50.pt",
    "RN101": "https://openaipublic.azureedge.net/clip/models/
 ↪8fa8567bab74a42d41c5915025a8e4538c3bdbe8804a470a72f30b0d94fab599/RN101.pt",
    "RN50x4": "https://openaipublic.azureedge.net/clip/models/
 ↪7e526bd135e493cef0776de27d5f42653e6b4c8bf9e0f653bb11773263205fdd/RN50x4.pt",
    "RN50x16": "https://openaipublic.azureedge.net/clip/models/
 ↪52378b407f34354e150460fe41077663dd5b39c54cd0bfd2b27167a4a06ec9aa/RN50x16.pt",
    "RN50x64": "https://openaipublic.azureedge.net/clip/models/
 ↪be1cfb55d75a9666199fb2206c106743da0f6468c9d327f3e0d0a543a9919d9c/RN50x64.pt",
    "ViT-B/32": "https://openaipublic.azureedge.net/clip/models/
 ↪40d365715913c9da98579312b702a82c18be219cc2a73407c4526f58eba950af/ViT-B-32.
 ↪pt",
    "ViT-B/16": "https://openaipublic.azureedge.net/clip/models/
 ↪5806e77cd80f8b59890b7e101eabd078d9fb84e6937f9e85e4ecb61988df416f/ViT-B-16.
 ↪pt",
    "ViT-L/14": "https://openaipublic.azureedge.net/clip/models/
 ↪b8cca3fd41ae0c99ba7e8951adf17d267cdb84cd88be6f7c2e0eca1737a03836/ViT-L-14.
 ↪pt",
}


def _download(url: str, root: str):
    os.makedirs(root, exist_ok=True)
    filename = os.path.basename(url)

    expected_sha256 = url.split("/")[-2]
    download_target = os.path.join(root, filename)
```

```
    if os.path.exists(download_target) and not os.path.isfile(download_target):
        raise RuntimeError(f"{download_target} exists and is not a regular␣
↪file")

    if os.path.isfile(download_target):
        if hashlib.sha256(open(download_target, "rb").read()).hexdigest() ==␣
↪expected_sha256:
            return download_target
        else:
            warnings.warn(f"{download_target} exists, but the SHA256 checksum␣
↪does not match; re-downloading the file")

    with urllib.request.urlopen(url) as source, open(download_target, "wb") as␣
↪output:
        with tqdm(total=int(source.info().get("Content-Length")), ncols=80,␣
↪unit='iB', unit_scale=True, unit_divisor=1024) as loop:
            while True:
                buffer = source.read(8192)
                if not buffer:
                    break

                output.write(buffer)
                loop.update(len(buffer))

    if hashlib.sha256(open(download_target, "rb").read()).hexdigest() !=␣
↪expected_sha256:
        raise RuntimeError(f"Model has been downloaded but the SHA256 checksum␣
↪does not not match")

    return download_target


def _convert_image_to_rgb(image):
    return image.convert("RGB")


def _transform(n_px):
    return Compose([
        Resize(n_px, interpolation=BICUBIC),
        CenterCrop(n_px),
        _convert_image_to_rgb,
        ToTensor(),
        Normalize((0.48145466, 0.4578275, 0.40821073), (0.26862954, 0.26130258,␣
↪0.27577711)),
    ])
```

```
def available_models() -> List[str]:
    """Returns the names of available CLIP models"""
    return list(_MODELS.keys())


def load(name: str, device: Union[str, torch.device] = "cuda" if torch.cuda.
 ↪is_available() else "cpu", jit: bool = False, download_root: str = None):
    """Load a CLIP model

    Parameters
    ----------
    name : str
        A model name listed by `clip.available_models()`, or the path to a␣
 ↪model checkpoint containing the state_dict

    device : Union[str, torch.device]
        The device to put the loaded model

    jit : bool
        Whether to load the optimized JIT model or more hackable non-JIT model␣
 ↪(default).

    download_root: str
        path to download the model files; by default, it uses "~/.cache/clip"

    Returns
    -------
    model : torch.nn.Module
        The CLIP model

    preprocess : Callable[[PIL.Image], torch.Tensor]
        A torchvision transform that converts a PIL image into a tensor that␣
 ↪the returned model can take as its input
    """
    if name in _MODELS:
        model_path = _download(_MODELS[name], download_root or os.path.
 ↪expanduser("~/.cache/clip"))
    elif os.path.isfile(name):
        model_path = name
    else:
        raise RuntimeError(f"Model {name} not found; available models =␣
 ↪{available_models()}")

    try:
```

```
        # loading JIT archive
        model = torch.jit.load(model_path, map_location=device if jit else␣
↪"cpu").eval()
        state_dict = None
    except RuntimeError:
        # loading saved state dict
        if jit:
            warnings.warn(f"File {model_path} is not a JIT archive. Loading as␣
↪a state dict instead")
            jit = False
        state_dict = torch.load(model_path, map_location="cpu")

    embed_dim = model.state_dict()["text_projection"].shape[1]
    if not jit:
        model = build_model(state_dict or model.state_dict()).to(device)
        if str(device) == "cpu":
            model.float()
        return model, embed_dim, _transform(model.visual.input_resolution)

    # patch the device names
    device_holder = torch.jit.trace(lambda: torch.ones([]).to(torch.
↪device(device)), example_inputs=[])
    device_node = [n for n in device_holder.graph.findAllNodes("prim::
↪Constant") if "Device" in repr(n)][-1]

    def patch_device(module):
        try:
            graphs = [module.graph] if hasattr(module, "graph") else []
        except RuntimeError:
            graphs = []

        if hasattr(module, "forward1"):
            graphs.append(module.forward1.graph)

        for graph in graphs:
            for node in graph.findAllNodes("prim::Constant"):
                if "value" in node.attributeNames() and str(node["value"]).
↪startswith("cuda"):
                    node.copyAttributes(device_node)

    model.apply(patch_device)
    patch_device(model.encode_image)
    patch_device(model.encode_text)

    # patch dtype to float32 on CPU
    if str(device) == "cpu":
```

```
        float_holder = torch.jit.trace(lambda: torch.ones([]).float(),␣
↪example_inputs=[])
        float_input = list(float_holder.graph.findNode("aten::to").inputs())[1]
        float_node = float_input.node()

        def patch_float(module):
            try:
                graphs = [module.graph] if hasattr(module, "graph") else []
            except RuntimeError:
                graphs = []

            if hasattr(module, "forward1"):
                graphs.append(module.forward1.graph)

            for graph in graphs:
                for node in graph.findAllNodes("aten::to"):
                    inputs = list(node.inputs())
                    for i in [1, 2]:  # dtype can be the second or third␣
↪argument to aten::to()
                        if inputs[i].node()["value"] == 5:
                            inputs[i].node().copyAttributes(float_node)

        model.apply(patch_float)
        patch_float(model.encode_image)
        patch_float(model.encode_text)

        model.float()

    return model, embed_dim, _transform(model.input_resolution.item())


def tokenize(texts: Union[str, List[str]], context_length: int = 77, truncate:␣
↪bool = False) -> torch.LongTensor:
    """
    Returns the tokenized representation of given input string(s)

    Parameters
    ----------
    texts : Union[str, List[str]]
        An input string or a list of input strings to tokenize

    context_length : int
        The context length to use; all CLIP models use 77 as the context length

    truncate: bool
        Whether to truncate the text in case its encoding is longer than the␣
↪context length
```

```
    Returns
    -------
    A two-dimensional tensor containing the resulting tokens, shape = [number of input strings, context_length]
    """
    if isinstance(texts, str):
        texts = [texts]

    sot_token = _tokenizer.encoder["<|startoftext|>"]
    eot_token = _tokenizer.encoder["<|endoftext|>"]
    all_tokens = [[sot_token] + _tokenizer.encode(text) + [eot_token] for text in texts]
    result = torch.zeros(len(all_tokens), context_length, dtype=torch.long)

    for i, tokens in enumerate(all_tokens):
        if len(tokens) > context_length:
            if truncate:
                tokens = tokens[:context_length]
                tokens[-1] = eot_token
            else:
                raise RuntimeError(f"Input {texts[i]} is too long for context length {context_length}")
        result[i, :len(tokens)] = torch.tensor(tokens)

    return result
```

6 6) TPT-core

To enhance code clarity and minimize the likelihood of errors, we decided to abstract the TPT-like evaluation process and parameterize both the loss function and the augmentations. This approach ensures the code is generic and adaptable for various experiments involving different loss functions and augmentations. By doing so, these parameters can be easily adjusted for each experiment.

Additionally, to monitor various metrics, we implemented a dedicated class which can be found in 3-Utils.

Below is the core implementation of TPT:

[74]:
```
import torch.backends.cudnn as cudnn
import torch.nn.functional as F
import torch
import numpy as np
from tqdm import tqdm
from copy import deepcopy
from torch.utils.tensorboard import SummaryWriter
import argparse
```

```
import json
import sys


def test_time_tuning(model, inputs, optimizer, scaler, args, tta_step=1):
    loss_value = 0.0
    for _ in range(tta_step):
        with torch.cuda.amp.autocast():
            output = model(inputs)
            loss = args.loss(output, args)
            loss_value = loss.item()

        optimizer.zero_grad()
        # compute gradient and do SGD step
        scaler.scale(loss).backward()
        # Unscales the gradients of optimizer's assigned params in-place
        scaler.step(optimizer)
        scaler.update()

    return loss_value


def test_time_adapt_eval(
    dataloader, model, optimizer, optim_state, scaler, writer, device, args
):
    metrics = MetricsTracker(args)

    model.eval()
    with torch.no_grad():
        model.reset()

    print("Test Time Evaluation")

    progress_bar = tqdm(enumerate(dataloader), total=len(dataloader))
    for i, (imgs, target) in progress_bar:
        view_img = imgs[0]
        images = torch.cat(imgs[1:], dim=0).to(device)  # don't consider view␣
 ↪image
        orig_img = imgs[1].to(device)
        target = target.to(device)

        with torch.no_grad():
            model.reset()
        optimizer.load_state_dict(optim_state)

        with torch.no_grad():
            with torch.cuda.amp.autocast():
```

```
                output_base = model(orig_img)

        loss_value = test_time_tuning(model, images, optimizer, scaler, args)

        with torch.no_grad():
            with torch.cuda.amp.autocast():
                output_tpt = model(orig_img)

        metrics.update(
            i, view_img, output_base, output_tpt, target, loss_value, writer,␣
↪args
        )

        progress_bar.set_postfix(
            {
                "Base Acc": f"{metrics.get_accuracy_base():.2f}%",
                "TPT Acc": f"{metrics.get_accuracy_tpt():.2f}%",
            }
        )

    metrics.write_info(writer, args)

    return metrics.get_accuracy_tpt()
```

7 7) Baseline

As first step we decided to reproduce the baseline: the standard version of TPT. To do this we need to implement the described augmenter and loss.

7.1 AugmenterTPT

The `AugmenterTPT` class is used to handle the various processes required to transform images from their original dimensions to the format needed by the model. Specifically, it performs the following steps: 1. **CLIP Preprocessing**: It applies preprocessing tailored for CLIP, which includes resizing, cropping, and normalizing the image to fit the expected input format. 2. **Normalization**: After preprocessing, the image is normalized with predefined mean and standard deviation values specific to the model. 3. **Preaugmentation**: The class applies random resizing crop and horizontal flipping to introduce variation in the data. 4. **AugMix (Optional)**: AugMix is an optional flag that, when enabled, applies more complex augmentations, combining multiple transformations with a specified severity.

[75]:
```
## applies clip_preprocess to the original images and
## random resized and horizontal flip + augmix for the rest of the images
class AugmenterTPT(object):
    def __init__(self, n_aug=2, augmix=False, severity=1):
        self.n_aug = n_aug
        self.clip_preprocess = get_clip_preprocess()
```

```
        self.normalize = get_normalizer()
        self.preaugment = get_preaugment()
        self.augmix = (
            transforms.AugMix(severity=severity) if augmix else␣
↪IdentityTransform()
        )

    def __call__(self, x):
        image = self.clip_preprocess(x)
        augmented = [
            self.normalize(self.augmix(self.preaugment(x))) for _ in range(self.
↪n_aug)
        ]
        return [TF.to_tensor(x)] + [image] + augmented
```

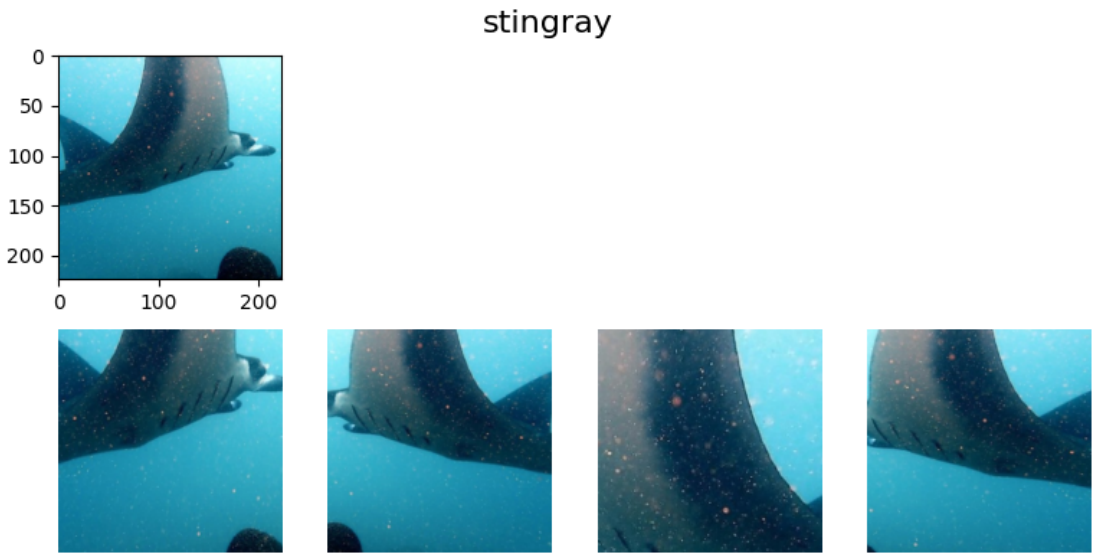
7.2 Augmentation Comparison

In the following examples, we can see two different types of augmentations used for this experiment. The first set of images shows the original image with 4 augmentations, including only random resized crops and horizontal flips. The second set incorporates AugMix, which combines multiple transformations with varying severity, adding more complex and diverse variations to the images.

In particolar for the baseline we use the Augmentation **without** the AugMix as we will describe in section 7-Results.

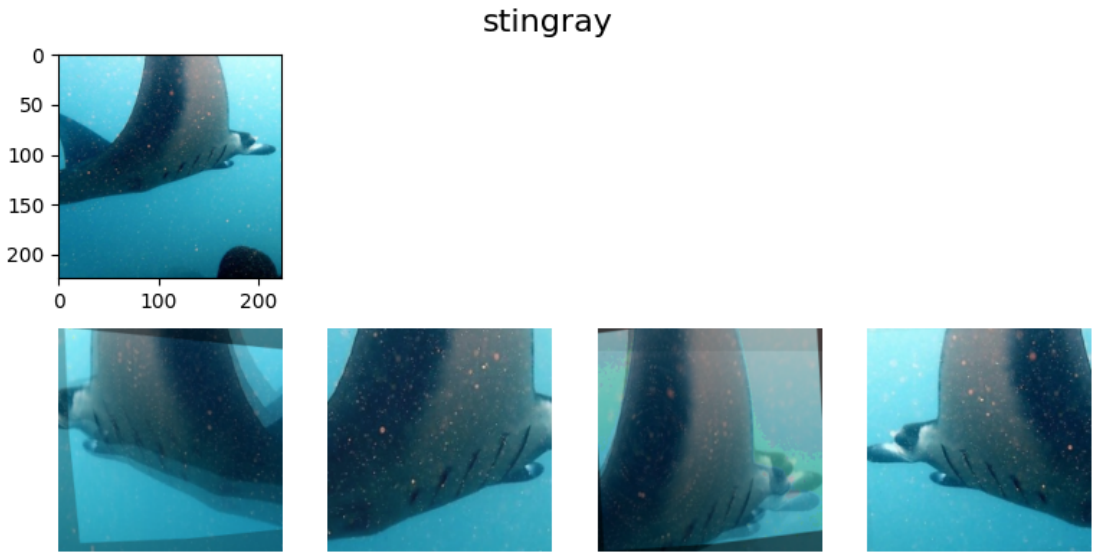
AugmenterTPT - without AugMix

[76]:
```
imgaug = AugmenterTPT(n_aug=4)(image)
show_imgs(imgaug[1:], title=class_label)
```



####AugmenterTPT - with AugMix

[77]:
```
imgaug = AugmenterTPT(n_aug=4, augmix=True, severity=4)(image)
show_imgs(imgaug[1:], title=class_label)
```



7.3 Default TPT loss

This code implements the loss function for Test-Time Prompt Tuning (TPT) as described in the paper. It minimizes the entropy of predictions across multiple augmented views of a test image, selecting confident samples based on their entropy. By averaging the predictions from these augmented views, the loss encourages consistent predictions, improving the model's zero-shot performance without requiring labeled data.

[78]:
```
import torch
import numpy as np
import torch.nn.functional as F


def select_confident_samples(logits, top):
    batch_entropy = -(logits.softmax(1) * logits.log_softmax(1)).sum(1)
    idx = torch.argsort(batch_entropy, descending=False)[
        : int(batch_entropy.size()[0] * top)
    ]
    return logits[idx]


def avg_entropy(outputs):
    logits = outputs - outputs.logsumexp(
        dim=-1, keepdim=True
```

```
    )  # logits = outputs.log_softmax(dim=1) [N, 1000]
    avg_logits = logits.logsumexp(dim=0) - np.log(
        logits.shape[0]
    )  # avg_logits = logits.mean(0) [1, 1000]
    min_real = torch.finfo(avg_logits.dtype).min
    avg_logits = torch.clamp(avg_logits, min=min_real)
    return -(avg_logits * torch.exp(avg_logits)).sum(dim=-1)


def defaultTPT_loss(output, args):
    output = select_confident_samples(output, args.selection_p_all)
    return avg_entropy(output)
```

7.4 Run

[79]:
```
args = default_args()
args.save_imgs = False
print_args(args)
```

```
Config: {
    "imagenet_a_path": "./imagenet-a/",
    "coop_weight_path": "./model.pth.tar-50",
    "n_aug": 63,
    "n_patches": 0,
    "batch_size": 1,
    "arch": "RN50",
    "device": "cuda:0",
    "learning_rate": 0.005,
    "n_ctx": 4,
    "ctx_init": "",
    "class_token_position": "end",
    "csc": false,
    "run_name": "",
    "augmenter": "AugmenterTPT",
    "loss": "defaultTPT",
    "augmix": true,
    "severity": 2,
    "num_workers": 4,
    "save": true,
    "reduced_size": null,
    "dataset_shuffle": true,
    "save_imgs": false,
    "selection_p_all": 0.1,
    "selection_p_patch": 0.9,
    "overlap": 0.0,
    "alpha_exponential_weightening": 1.0,
    "seed": 1337
}
```

[80]:
```
#run(args) ## uncomment to run the baseline
```

7.5 Results

Since the original TPT paper does not explicitly state whether AugMix was utilized and if so which was the optimal value of the severity parameter, we conducted two separate experiments to evaluate its potential impact.

Since the evaluation is quite expensive we tracked different metrics from the start to facilitate future analysis, some important one are:

We refer as Base the Zero-Shot model and TPT as the model after the tpt trainig is done.

- AccuracyTOP1/AccuracyTOP5
- Base: cumulative Zero-Shot accuracy
- Final_Base: final scalar result of Zero-Shot accuracy
- TPT_coop: cumulative tpt accuracy
- Final_TPT_coop: final scalar result of tpt accuracy
- difference(TPT-base): accuracy difference between tpt and Zero-Shot, for visualization pourse
- KL: Kullback–Leibler divergence
- Loss: loss magnitude to empirically check the training signal.
- Samples:
- Samples/TPTchange: for each sample \ (1): if tpt is correct and Zero-Shot is not \ (0): both wrong or both correct \ (-1): if zerZero-Shotoshot is correct and tpt is not \
- Samples/TPTcumulative_changed: cumulative change.
- Samples-Global:
  - Samples-Global/ImprovedRatio: Number of examples that TPT correctly classified but Zero-Shot misclassified. (%)
  - Samples-Global/NWorsened: Number of examples that TPT misclassified but Zero-Shot correctly classified. (%)
- Samples-PerClass: same as previous but for each class
- confidence:
- confidence/Base: confidence of the correct class by tpt
- confidence/TPT_coop: confidence of the correct class by Zero-Shot
- crossEntropy

Since we tracked multiple runs we decided to set as name the parameters of that particular run. In this case for example the augmix is different.

[81]:
```
%tensorboard --logdir runs_baseline/
```

```
<IPython.core.display.Javascript object>
```

7.6 Considerations

We have present again the TOP1 accuracy in a table for improved readability.

Model	TOP1Accuracy	Samples-Global Improved	Samples_Global Worsened
Zero Shot	0.2314	/	/
Tpt without AugMix	0.2936	661	196
Tpt with AugMix (severity=2)	0.2752	554	223

In this table, we compare the performance of the Zero-Shot model with the TPT model, both with and without augmentations. The results indicate that the TPT model without AugMix achieves the best accuracy. This is further evident from the Samples Globally Improved, where it has more improved samples compared to the TPT model with AugMix, and fewer worsened samples.

The accuracy results without AugMix are consistent with those reported in the original paper (link), confirming the correctness of our implementation. Consequently, we will use the TPT model without AugMix as the baseline to evaluate differences relative to our approach.

8 8) Our Approach

8.1 Main Idea

In this work, we extend the original Test-Time Prompt Tuning (TPT) method by introducing a novel strategy: instead of applying augmentations to the entire image, we apply them to smaller patches within the image. Specifically, we divide the image into multiple patches and apply augmentations to each patch separately. This enables the model to capture localized features and focus on specific regions of the image. Such an approach is particularly beneficial when the object of interest occupies only a small portion of the image and is surrounded by a large background.

The motivation behind this modification stems from the characteristics of many images in the Imagenet-A dataset, where objects are often very small compared to the surrounding background. In these cases, the model may struggle to focus on the object due to the overwhelming amount of background information. Traditional image-wide augmentations may not effectively highlight the small object and can lead to the model missing important details. By subdividing the image into patches and applying augmentations to each of them independently, we allow the model to focus on more localized areas, thus improving its ability to detect the small, relevant object.

8.2 Patches and PatchAugmenter

In particular, in our implementation, we decided to split the image into smaller patches with and without overlap: * The method without overlap divides the image into distinct, non-intersecting patches, each representing a separate region of the image. * In contrast, the method with overlap allows adjacent patches to share some content, providing a more continuous view of the image and helping the model capture context that spans across patch boundaries.

For each of these patches, multiple augmentations are applied to introduce variation. Additionally, we include the original image along with its augmentations, allowing the model to compare the patches with the original image during loss computation. This strategy helps the model leverage both localized and global features for more robust learning.

8.3 8.1) Initial Experimentation without Overlap

We first analyze the model's performance using the method without overlap to see how it reacts to isolated regions of the image. This provides insight into how well the model focuses on individual patches in isolation. Later, we introduce the overlap technique to allow the model to capture more context across patch boundaries, helping improve its understanding of relationships between neighboring regions and providing a more holistic view of the image.

[82]:
```
def crop_patches_with_overlap(image, n, base_transform, overlap=0.0):
    if n == 0:
        return []

    n = math.floor(math.sqrt(n))
    width, height = image.size
    part_width = width // n
    part_height = height // n

    # Calculate the overlap in pixels
    overlap_width = int(part_width * overlap)
    overlap_height = int(part_height * overlap)

    images = []

    for i in range(n):
        for j in range(n):
            left = max(j * part_width - overlap_width, 0)
            upper = max(i * part_height - overlap_height, 0)
            right = min((j + 1) * part_width + overlap_width, width)
            lower = min((i + 1) * part_height + overlap_height, height)

            cropped_image = image.crop((left, upper, right, lower))
            images.append(base_transform(cropped_image))

    return images


## n_patches is the total number of patches
class PatchAugmenter(object):
    def __init__(self, n_aug=2, n_patches=16, overlap=0.0, augmix=False,␣
 ↪severity=1):
        self.n_aug = n_aug
        self.n_patches = n_patches
        self.overlap = overlap
```

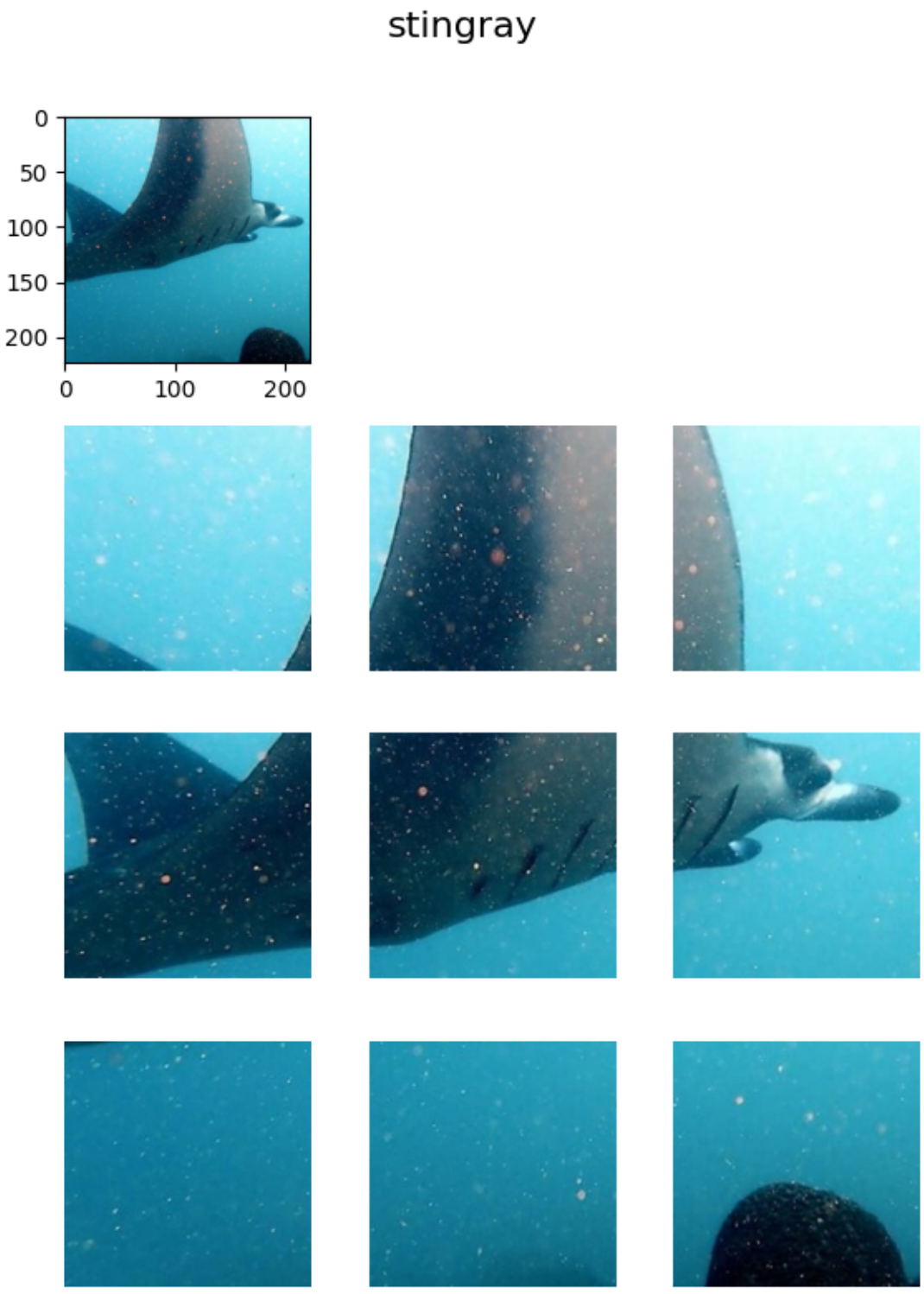
```
        self.clip_preprocess = get_clip_preprocess()
        self.base_transform = get_base_transform()
        self.normalize = get_normalizer()
        self.augmix = (
            transforms.AugMix(severity=severity) if augmix else␣
↪IdentityTransform()
        )
        self.crop_patches = crop_patches_with_overlap

    def __call__(self, x):
        img = self.clip_preprocess(x)
        img_orig_aug = [
            self.normalize(self.augmix(self.base_transform(x)))
            for _ in range(self.n_aug)
        ]

        patches = self.crop_patches(
            x, self.n_patches, self.base_transform, overlap=self.overlap
        )
        patches_augm = [
            augmented_patch
            for patch in patches
            for augmented_patch in [self.normalize(patch)]
            + [self.normalize(self.augmix(patch)) for _ in range(self.n_aug)]
        ]
        return [TF.to_tensor(x)] + [img] + img_orig_aug + patches_augm
```

Here, we display the original image alongside the 9 patches obtained by dividing the image into 9 equal, non-overlapping squares.

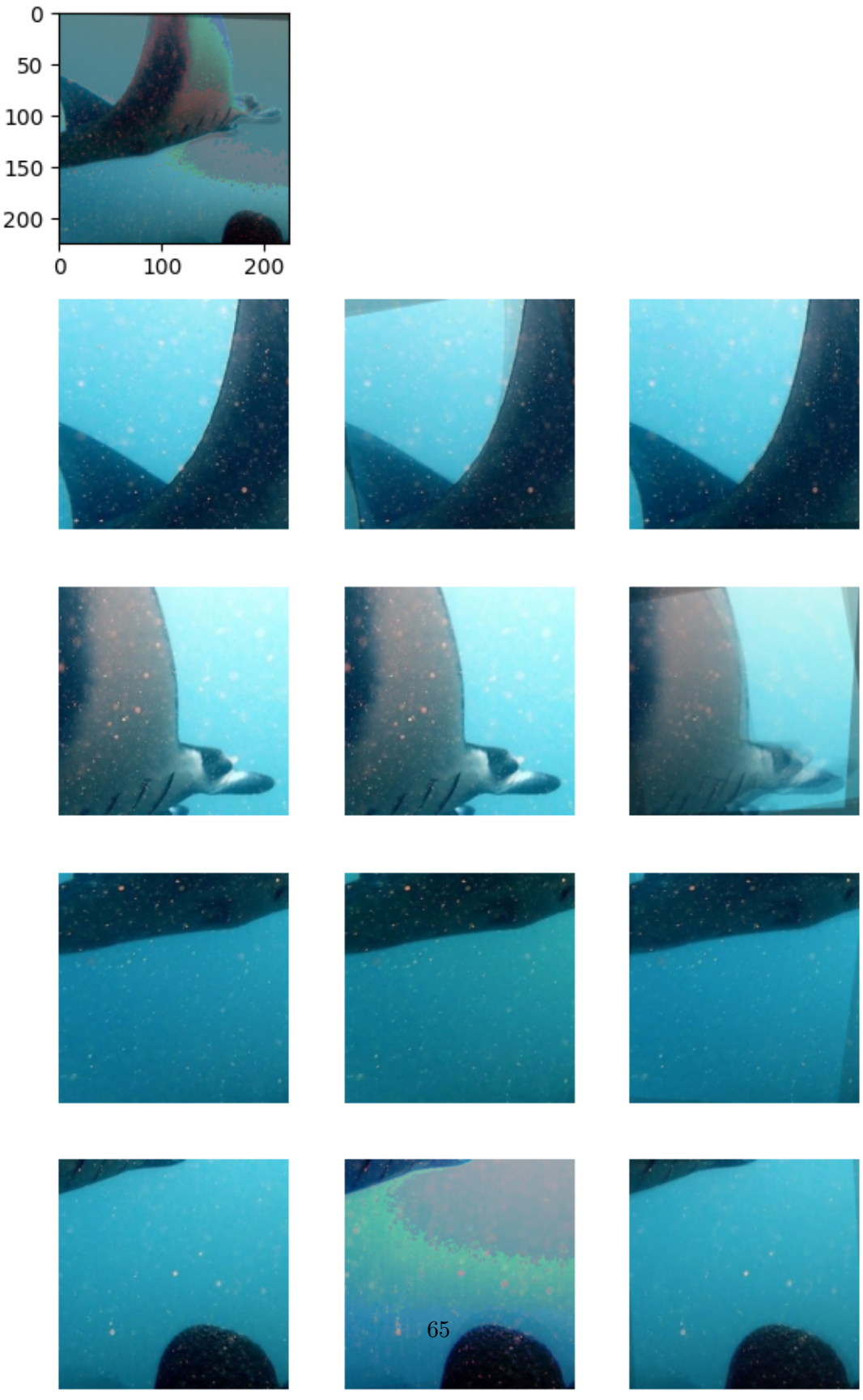
[83]:
```
imgaug = PatchAugmenter(n_patches=9, n_aug=0, overlap=0.0)(image)
show_imgs(imgaug[1:], images_per_row=3, title=class_label)
```



In this image, we display the original image followed by 4 rows of images, each corresponding to one of the 4 patches. In each row, the first image shows the original patch, followed by two augmented

versions of the same patch, with both standard and AugMix transformations at a severity level of 4.

[84]:
```
imgaug = PatchAugmenter(n_patches=4, n_aug=2, overlap=0.0, augmix=True,␣
↪severity=4)(image)
show_imgs(imgaug[3:], images_per_row=3, title=class_label)
```


stingray
0
50
100
150
200
0
100
200

8.4 Patch Losses

In our approach, we combine global and local features to improve model robustness, particularly when dealing with images where the object is small or surrounded by background noise. To achieve this, we design loss functions that leverage both the original image and local patches.

The design of specific loss is a **complex task** and is hard to understand the performance a priori. Minor variations in the design can lead to significant differences in the behavior and effectiveness of the model. For this reason we implemented and tested a variety of losses with slight differences and similarities between them.

8.4.1 Losses are divided into 3 different groups as follows:

Group 1: Ensuring Global-Local Consistency

- patch_loss1: Aligns the class distribution of the original image with a weighted distribution derived from patch predictions, emphasizing confident predictions.
- patch_loss2: Matches the original image's prediction with the most frequent class among the patches, weighted by entropy to account for reliability.

These losses aim to ensure consistency between global (image) and local (patch) predictions, preserving the overall context while capturing fine-grained details.

Group 2: Reliability-Based Weighting with Entropy

- patch_loss3: Computes a global loss by weighting each patch based on the inverse of its entropy, highlighting the most reliable patches.
- patch_loss4: Focuses on the patch with the lowest entropy, considering it the most representative for robust predictions.

These losses prioritize confident patches, reducing the impact of noisy or less reliable patches on the final prediction.

Group 3: Aggregating Probabilities for Robustness

- patch_loss5: Aggregates probabilities across patches, weighting them by the inverse of their entropy to derive a reliable global prediction.
- patch_loss6: Combines patch probabilities using an exponential weighting scheme based on entropy inverse, controlled by the parameter alpha, to dynamically adjust the influence of each patch.

This group focuses on leveraging local patch information effectively to produce robust global predictions, ensuring that patches contribute proportionally to their reliability.

[85]:
```python
def reshape_output_patches(output, args):
    return output.view(-1, args.n_aug + 1, output.shape[-1])


def select_most_frequent_class(logits):
```

```
    probabilities = logits.softmax(dim=-1)
    predicted_classes = probabilities.argmax(dim=2)
    most_frequent_classes = torch.mode(predicted_classes, dim=1).values
    return most_frequent_classes


def weighted_most_frequent_class(entropy, classes):
    weights = 1 / (entropy + 1e-6)
    weighted_counts = {}
    for i, cls in enumerate(classes):
        if cls.item() not in weighted_counts:
            weighted_counts[cls.item()] = 0
        weighted_counts[cls.item()] += weights[i].item()
    most_frequent_class = max(weighted_counts, key=weighted_counts.get)
    return torch.tensor(most_frequent_class).to(entropy.device)


def weighted_class_distribution(entropy, classes, n_classes):
    weights = 1 / (entropy + 1e-6)

    weighted_counts = {}
    for i, cls in enumerate(classes):
        if cls.item() not in weighted_counts:
            weighted_counts[cls.item()] = 0
        weighted_counts[cls.item()] += weights[i].item()

    total_weight = sum(weighted_counts.values())
    distribution = torch.zeros(n_classes, device=entropy.device)

    for cls, weight in weighted_counts.items():
        distribution[cls] = weight / total_weight

    return distribution


def patch_loss1(outputs, args):
    outputs = reshape_output_patches(outputs, args)
    selected_outputs = []

    output_original_image = outputs[0][0]

    patch_entropy = []
    for i in range(0, args.n_patches + 1):
        selected_output = select_confident_samples(outputs[i], args.
 ↪selection_p_patch)
        patch_entropy.append(avg_entropy(selected_output))
```

```
        selected_outputs.append(
            selected_output
        )  # Append each selected output to the list
        """
        for s in selected_output:
            selected_class = torch.softmax(s, dim=0).max(0).indices.item()
            print(selected_class)
        """

    # 1 Cross entropy loss between the original image and the weighted␣
↪probability distribution

    all_selected_output = torch.stack(selected_outputs, dim=0)
    patch_entropy = torch.stack(patch_entropy, dim=0)
    patch_classes = select_most_frequent_class(
        all_selected_output
    )  # Calculate the most frequent class for each patch
    # Calculate the weighted class distribution based on occorrencies in order␣
↪to use it as the target distribution
    target_distribution = weighted_class_distribution(
        patch_entropy[1:], patch_classes[1:], output_original_image.shape[-1]
    )
    # Calculate the cross entropy loss between the original image and the␣
↪target probability distribution
    log_probs = F.log_softmax(output_original_image, dim=-1)
    loss = -(target_distribution * log_probs).sum()

    return loss


def patch_loss2(outputs, args):
    outputs = reshape_output_patches(outputs, args)
    selected_outputs = []

    output_original_image = outputs[0][0]

    patch_entropy = []
    for i in range(0, args.n_patches + 1):
        selected_output = select_confident_samples(outputs[i], args.
↪selection_p_patch)
        patch_entropy.append(avg_entropy(selected_output))

        selected_outputs.append(
            selected_output
        )  # Append each selected output to the list
        """
        for s in selected_output:
```

```
            selected_class = torch.softmax(s, dim=0).max(0).indices.item()
            print(selected_class)
        """
    # 2 Cross entropy loss between the original image and the most frequent␣
↪class

    all_selected_output = torch.stack(selected_outputs, dim=0)
    patch_entropy = torch.stack(patch_entropy, dim=0)
    patch_classes = select_most_frequent_class(
        all_selected_output
    )  # Calculate the most frequent class for each patch
    # Calculate the most frequent class for the patches weighted by their␣
↪entropy
    most_frequent_class = weighted_most_frequent_class(
        patch_entropy[1:], patch_classes[1:]
    )
    # Calculate the cross entropy loss between the original image and the most␣
↪frequent class
    loss = F.cross_entropy(
        output_original_image.unsqueeze(0), most_frequent_class.unsqueeze(0)
    )

    return loss


def patch_loss3(outputs, args):
    outputs = reshape_output_patches(outputs, args)
    selected_outputs = []

    output_original_image = outputs[0][0]

    patch_entropy = []
    for i in range(0, args.n_patches + 1):
        selected_output = select_confident_samples(outputs[i], args.
↪selection_p_patch)
        patch_entropy.append(avg_entropy(selected_output))

        selected_outputs.append(
            selected_output
        )  # Append each selected output to the list
        """
        for s in selected_output:
            selected_class = torch.softmax(s, dim=0).max(0).indices.item()
            print(selected_class)
        """
    # 3 Give a weight to each patch based on its entropy
    patch_entropy = torch.stack(patch_entropy, dim=0)
```

```
    epsilon = 1e-6
    weights = 1 / (patch_entropy + epsilon)
    weights /= weights.sum()  # normalization
    weighted_entropy = (patch_entropy * weights).sum()

    loss = weighted_entropy

    return loss


def patch_loss4(outputs, args):
    outputs = reshape_output_patches(outputs, args)
    selected_outputs = []

    output_original_image = outputs[0][0]

    patch_entropy = []
    for i in range(0, args.n_patches + 1):
        selected_output = select_confident_samples(outputs[i], args.
 ↪selection_p_patch)
        patch_entropy.append(avg_entropy(selected_output))

        selected_outputs.append(
            selected_output
        )  # Append each selected output to the list
        """
        for s in selected_output:
            selected_class = torch.softmax(s, dim=0).max(0).indices.item()
            print(selected_class)
        """
    # 4 Use the loss of the patch with the lowest entropy
    patch_entropy = torch.stack(patch_entropy, dim=0)
    loss = patch_entropy.min()

    return loss


"""
compute the mean logit for each patch
compute the entropy for each patch

output prob is the weighted average of the prob of each patch
weighted by the inverse of the corresponding entropy
"""


def patch_loss5(outputs, args):
```

```
    epsilon = 1e-6

    output_reshaped = reshape_output_patches(outputs, args)

    mean_output_per_patch = output_reshaped.mean(dim=1)

    mean_logprob_per_patch = mean_output_per_patch.log_softmax(dim=1)
    entropy_per_patch = -(
        mean_logprob_per_patch * torch.exp(mean_logprob_per_patch)
    ).sum(dim=-1)

    weighted_logprob_per_patch = mean_logprob_per_patch * (
        1 / (entropy_per_patch.unsqueeze(dim=1) + epsilon)
    )

    logprob_output = weighted_logprob_per_patch.mean(dim=0).log_softmax(dim=0)
    entropy_loss = -(logprob_output * torch.exp(logprob_output)).sum(dim=0)

    return entropy_loss


"""
compute the mean logit for each patch
compute the entropy for each patch

output prob is the weighted average of the prob of each patch
weighted exponentially by the corresponding entropy
"""


def patch_loss6(outputs, args):
    alpha = args.alpha_exponential_weightening

    output_reshaped = reshape_output_patches(outputs, args)

    mean_output_per_patch = output_reshaped.mean(dim=1)

    mean_logprob_per_patch = mean_output_per_patch.log_softmax(dim=1)
    entropy_per_patch = -(
        mean_logprob_per_patch * torch.exp(mean_logprob_per_patch)
    ).sum(dim=-1)

    exp_weights = torch.exp(-alpha * entropy_per_patch).unsqueeze(dim=1)
    weighted_logprob_per_patch = mean_logprob_per_patch * exp_weights

    logprob_output = weighted_logprob_per_patch.mean(dim=0).log_softmax(dim=0)
    entropy_loss = -(logprob_output * torch.exp(logprob_output)).sum(dim=0)
```

```
    return entropy_loss
```

8.5 Run

Here, for example, there is a possible run using the patches and the patch_loss1.

[86]:
```
args = default_args()
args.augmenter = "PatchAugmenter"
args.loss = "patch_loss1"
args.n_aug = 4
args.n_patches = 16
args.save_imgs = False
print_args(args)
```

```
Config: {
    "imagenet_a_path": "./imagenet-a/",
    "coop_weight_path": "./model.pth.tar-50",
    "n_aug": 4,
    "n_patches": 16,
    "batch_size": 1,
    "arch": "RN50",
    "device": "cuda:0",
    "learning_rate": 0.005,
    "n_ctx": 4,
    "ctx_init": "",
    "class_token_position": "end",
    "csc": false,
    "run_name": "",
    "augmenter": "PatchAugmenter",
    "loss": "patch_loss1",
    "augmix": true,
    "severity": 2,
    "num_workers": 4,
    "save": true,
    "reduced_size": null,
    "dataset_shuffle": true,
    "save_imgs": false,
    "selection_p_all": 0.1,
    "selection_p_patch": 0.9,
    "overlap": 0.0,
    "alpha_exponential_weightening": 1.0,
    "seed": 1337
}
```

[87]:
```
#run(args) ## uncomment to run
```

8.6 Results

To ensure a fair comparison with the baseline, we matched computational utilization by using 4 augmentations and 16 patches, resulting in 64 augmented images per original image, consistent with the baseline setup.

The experiment aimed to evaluate the performance of various implemented loss functions. Initially, we conducted parameter tuning on a small dataset partition to optimize configurations. Subsequently, we performed an in-depth analysis of the loss functions.

As before to track the results of our experiments we used tensorboard with the same metrics and setup.

[88]:
```
%tensorboard --logdir runs_patch/
```

```
<IPython.core.display.Javascript object>
```

8.6.1 Consideration

The table below summarizes the performance of various loss functions under the same augmentation settings:

Loss Type	naug	npatch	Accuracy	Samples-Global Improved	Samples-Global Worsened
defaultTPT	4	16	0.2341	235	218
patch_loss1	4	16	0.1857	547	887
patch_loss2	4	16	0.2053	501	694
patch_loss3	4	16	0.2320	202	199
patch_loss4	4	16	0.2375	267	219
patch_loss5	4	16	0.2223	184	256
patch_loss6	4	16	0.2240	183	241

General Observations

- **Overall Performance**: Results under the current augmentation settings are significantly worse compared to the baseline, with accuracies being close to zero-shot levels.

- **Custom Loss Functions**: Even loss functions designed to exploit the patch structure failed to outperform the baseline, indicating limited effectiveness of the proposed approaches in the current setup.

Specific Findings

- **Default TPT and Comparable Losses**: `defaultTPT` loss, along with `patch_loss3` and `patch_loss4` shows slightly more improvements than deteriorations in samples.

- **Trade-Off Between Improvements and Worsenings**: Losses with a high number of sample improvements (`patch_loss1` and `patch_loss2`) also exhibit a proportionally high number of deteriorations, negating the net benefit.

Hypotheses and Next Steps The weak performance of custom losses may stem from two factors:
1. **Weak Signal**: The gradient signal generated by the custom losses might not be strong enough to improve the model in a single training step.
2. **Noisy Optimization**: Loss functions appear noisy, leading to unstable updates.

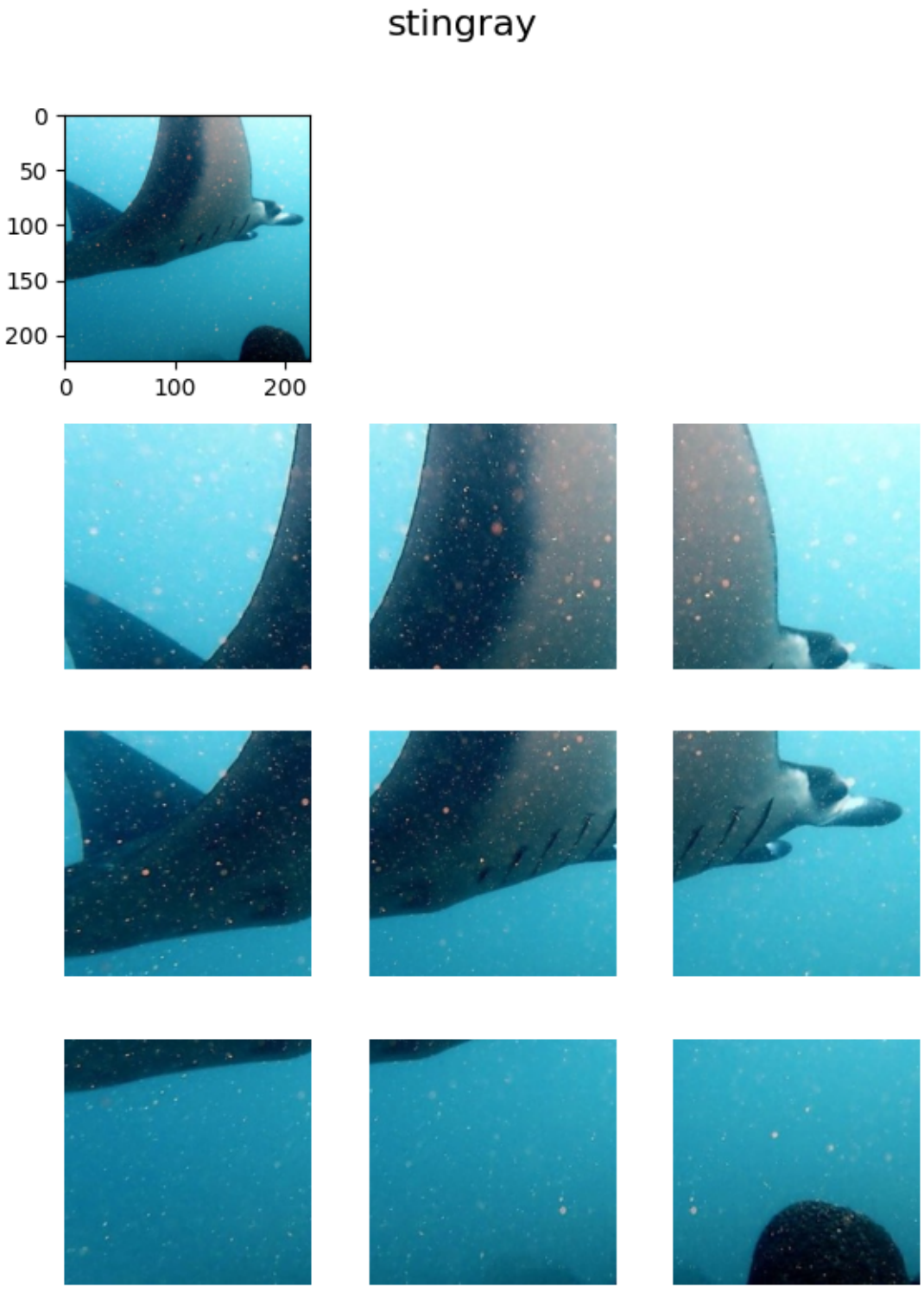
Given that the baseline approach benefits from random crops, where some regions of the image appear multiple times, we propose implementing **patch overlapping** to enhance training signal and better align with the baseline behavior.

9 8.2) Experimentation with Overlap

Patch Overlap involves dividing the image into patches as before but introducing **horizontal** and **vertical** overlap between the patches. Specifically, the overlap is controlled by a parameter in the PatchAugmenter class, which specifies the degree of overlap between adjacent patches.

Here, we show the same images as before but with an overlapping of 25%.

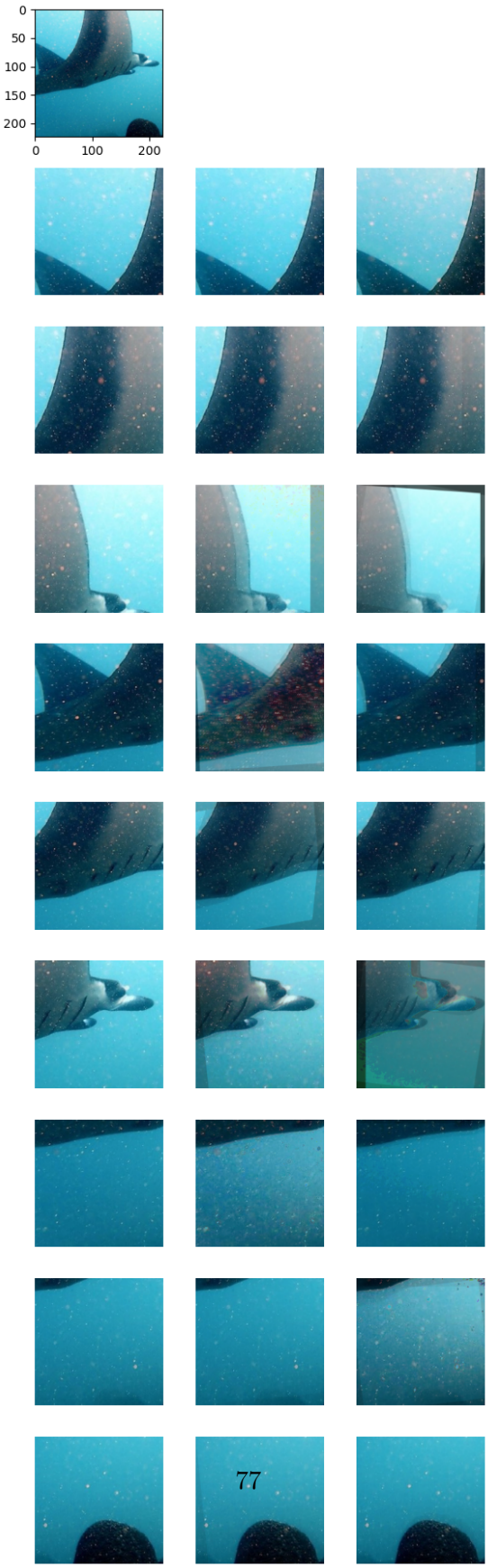
[89]:
```
imgaug = PatchAugmenter(n_patches=9, n_aug=0, overlap=0.25)(image)
show_imgs(imgaug[1:], images_per_row=3, title=class_label)
```



Now we add 2 augmentations for each overlapped patch.

[90]:
```
imgaug = PatchAugmenter(n_patches=9, n_aug=2, overlap=0.25, augmix=True,␣
↪severity=4)(image)
show_imgs(imgaug[3:], images_per_row=3, title=class_label)
```

stingray



9.1 Run

[91]:
```
args = default_args()
args.augmenter = "PatchAugmenter"
args.n_aug = 4
args.n_patches = 16
args.overlap = 0.8
args.save_imgs = False
print_args(args)
```

```
Config: {
    "imagenet_a_path": "./imagenet-a/",
    "coop_weight_path": "./model.pth.tar-50",
    "n_aug": 4,
    "n_patches": 16,
    "batch_size": 1,
    "arch": "RN50",
    "device": "cuda:0",
    "learning_rate": 0.005,
    "n_ctx": 4,
    "ctx_init": "",
    "class_token_position": "end",
    "csc": false,
    "run_name": "",
    "augmenter": "PatchAugmenter",
    "loss": "defaultTPT",
    "augmix": true,
    "severity": 2,
    "num_workers": 4,
    "save": true,
    "reduced_size": null,
    "dataset_shuffle": true,
    "save_imgs": false,
    "selection_p_all": 0.1,
    "selection_p_patch": 0.9,
    "overlap": 0.8,
    "alpha_exponential_weightening": 1.0,
    "seed": 1337
}
```

[92]:
```
#run(args) ## uncomment to run
```

9.2 Results

The objective of this experiment is to assess the impact of varying levels of overlap and compare the results to those obtained without overlap. For consistency, we used the default TPT loss, which

demonstrated one of the best performances when applied to patches (Chapter 8.1).

Since most of the custom-designed losses performed worse than the default TPT loss, and overall, the patch-based losses showed lower performance compared to the baseline TPT, we decided to keep the default loss in order to better understand whether the patching approach could be effective. To isolate the potential issues with patching, we aimed to use as many common parameters as possible to investigate the performance gaps between patch-based and non-patch-based approaches.

[93]:
```
%tensorboard --logdir runs_overlap/
```

```
<IPython.core.display.Javascript object>
```

9.2.1 Consideration

Loss Type	naug	npatch	overlap	Accuracy	Samples-Global Improved	Samples-Global Worsened
defaultTPT	0	64	0.0	0.2243	125	176
defaultTPT	0	64	0.2	0.2324	196	190
defaultTPT	0	64	0.4	0.2387	251	194
defaultTPT	2	32	0.2	0.2451	297	192
defaultTPT	2	32	0.4	0.2635	457	218
defaultTPT	2	32	0.6	0.2773	576	230
defaultTPT	4	16	0.4	0.2707	511	220
defaultTPT	4	16	0.6	0.2841	614	220
defaultTPT	4	16	0.8	0.2895	660	223
defaultTPT	8	8	0.8	0.2644	473	227

The table above summarizes the performance of the default TPT loss with varying levels of patch overlap and augmentation. Some considerations are: - **Impact of Overlap**: Accuracy improves significantly as the overlap between patches increases. This improvement is likely because overlapping patches provide more contextual information by including shared regions of the image. - **Impact of Augmentations**: Performance also improves with a higher number of augmentations. This is likely because the target object appears in more images during the training step. - **Improved vs. Worsened Samples**: The number of samples that worsen compared to zero-shot remains relatively constant across all experiments. However, the number of improved samples increases substantially as the overlap grows.

These results indicate that incorporating overlap between patches helps enhance model performance. This insight will be leveraged in subsequent experiments.

10 8.3) New general Losses

During the implementation of patch overlap, we identified an opportunity to design additional losses within the general framework that do not rely on the patch structure.

The core idea of these losses is to construct a target distribution from the augmented, cropped, or patched images and train the model using the standard cross-entropy loss between the model's output distribution for the original image and the constructed target distribution.

We implemented three variants:
- **soft_loss**: The target distribution is the mean of the top N outputs ranked by the inverse of their entropy.
- **hard1**: The target distribution consists of all zeros, with a single peak of 1 assigned to the class with the highest confidence.
- **hard5**: Similar to **hard1**, but instead of a single peak, the top 5 most confident classes are included and normalized to form a probability distribution.

[94]:
```
def crossentropy_soft_loss(output, args):
    orig_img_output = output[0].softmax(dim=0)

    best_outputs = select_confident_samples(output[1:], args.selection_p_all)
    target_dist = best_outputs.mean(dim=0).softmax(dim=0)

    loss = torch.nn.CrossEntropyLoss()(orig_img_output, target_dist)
    return loss


def crossentropy_hard1_loss(output, args):
    orig_img_output = output[0].softmax(dim=0)

    best_outputs = select_confident_samples(output[1:], args.selection_p_all)
    best_mean = best_outputs.mean(dim=0)

    max_idx = torch.argmax(best_mean)

    target = torch.zeros_like(best_mean)
    target[max_idx] = 1

    loss = torch.nn.CrossEntropyLoss()(orig_img_output, target)
    return loss


def crossentropy_hard5_loss(output, args):
    orig_img_output = output[0].softmax(dim=0)

    best_outputs = select_confident_samples(output[1:], args.selection_p_all)
    best_mean = best_outputs.mean(dim=0)

    topk_values, topk_indices = torch.topk(best_mean, k=5)
    filtered_dist = torch.zeros_like(best_mean)
    filtered_dist[topk_indices] = topk_values
    target = filtered_dist / filtered_dist.sum()

    loss = torch.nn.CrossEntropyLoss()(orig_img_output, target)
    return loss
```

10.1 Run

[95]:
```
args = default_args()
args.augmenter = "PatchAugmenter"
args.loss = "crossentropy_hard1"
args.n_aug = 4
args.n_patches = 16
args.overlap = 0.80
args.save_imgs = False
print_args(args)
```

```
Config: {
    "imagenet_a_path": "./imagenet-a/",
    "coop_weight_path": "./model.pth.tar-50",
    "n_aug": 4,
    "n_patches": 16,
    "batch_size": 1,
    "arch": "RN50",
    "device": "cuda:0",
    "learning_rate": 0.005,
    "n_ctx": 4,
    "ctx_init": "",
    "class_token_position": "end",
    "csc": false,
    "run_name": "",
    "augmenter": "PatchAugmenter",
    "loss": "crossentropy_hard1",
    "augmix": true,
    "severity": 2,
    "num_workers": 4,
    "save": true,
    "reduced_size": null,
    "dataset_shuffle": true,
    "save_imgs": false,
    "selection_p_all": 0.1,
    "selection_p_patch": 0.9,
    "overlap": 0.8,
    "alpha_exponential_weightening": 1.0,
    "seed": 1337
}
```

[96]:
```
#run(args)
```

10.2 Results

Our objective is to identify the best-performing loss function using the original TPT augmenter and then evaluate its effectiveness within our approach (PatchAugmenter). A preliminary evaluation of the patch-based losses from the previous chapter did not yield promising results compared to the newly designed losses. Therefore, we decided to conduct a more in-depth analysis of the latter.

[97]:
```
%tensorboard --logdir runs_crossE_overlap/
```

```
<IPython.core.display.Javascript object>
```

10.2.1 Consideration

First Experiment: Default AugmenterTPT with Custom Losses

Augmenter	Loss Type	naug	npatch	overlap	Accuracy
AugmenterTPT	crossentropy_hard1	63	0	0.0	0.3121
AugmenterTPT	crossentropy_hard5	63	0	0.0	0.2556
AugmenterTPT	crossentropy_soft	63	0	0.0	0.2951
AugmenterTPT	defaultTPT	63	0	0.0	0.2936 (Baseline)

Replacing the default TPT loss with custom losses showed notable improvements: - **CrossEntropy Hard1** achieved the best performance (+1.85% over baseline), leveraging its discrete target distribution focused on the most confident class. - **CrossEntropy Soft** also outperformed the baseline, providing a smoother target distribution, but was less effective than Hard1. - **CrossEntropy Hard5** underperformed, likely due to noise from broader target distributions.

Second Experiment: PatchAugmenter with crossentropy_hard1 Starting from the best performance loss in the default AugmenterTPT we decide to test this loss also in our approach using the patches. The result are shown below:

Augmenter	Loss Type	naug	npatch	overlap	Accuracy
PatchAugmenter	crossentropy_hard1	2	32	0.4	0.2651
PatchAugmenter	crossentropy_hard1	2	32	0.6	0.2876
PatchAugmenter	crossentropy_hard1	4	16	0.4	0.2876
PatchAugmenter	crossentropy_hard1	4	16	0.6	0.3001
PatchAugmenter	crossentropy_hard1	4	16	0.8	0.3032

By combining patch overlap with the **crossentropy_hard1** loss, we achieve an accuracy of 0.3032, surpassing the baseline performance.

11 9) Conclusion and Future Works

In conclusion, this project analyzed the performance of a custom TPT variant that utilizes image patches instead of random cropping. Specifically, we developed a tailored augmenter and various loss functions to achieve this goal. As a first step, we reimplemented the original TPT framework as described in the paper and thoroughly examined its implementation. Since the use of AugMix was not detailed in the paper, we evaluated its impact and found that disabling it led to better performance, which we adopted as our baseline.

We then designed a custom augmenter to generate patches with augmentations and tested it using different loss functions tailored to this approach. However, initial results were not encouraging.

Subsequently, we introduced patch overlap, which improved performance beyond zero-shot understanding and highlighted the trade-off between patch size and augmentation.

Finally, we implemented additional loss functions that construct labels from augmented views and minimize the cross-entropy between the class probabilities of the original images and the generated label probabilities. This approach yielded significant improvements, surpassing the baseline performance.

In details, we now report the most significant result that we have found:

Augmenter	Loss Type	naug	npatch	overlap	Accuracy
AugmenterTPT	defaultTPT	63	0	0.0	0.2936 (Baseline)
AugmenterTPT	crossentropy_hard1	63	0	0.0	0.3121
PatchAugmenter	crossentropy_hard1	4	16	0.8	0.3032

The table summarizes the best results obtained in our experiments alongside the baseline performance. The **crossentropy_hard1** loss improves the performance of TPT + CoOp by 1.85% compared to the default TPT loss. Additionally, our approach, using the PatchAugmenter with this specific loss, achieves a 0.96% improvement over the baseline.

However, the performance of the PatchAugmenter remains lower than that of the AugmenterTPT. To address this, potential improvements could include designing more advanced losses based on Hard1 that also consider patch locations, not just individual patches. Another promising direction would be leveraging architectures like LLaVA to assist in constructing better target distributions, using their outputs to refine the loss formulation.