DADA_TCHAKONTE_Projet1

February 3, 2025

1 ENSTA MI201 2024

1.1 `Rendu du Projet 1: Attaque adversaire en segmentation (Pol Labarbarie, Adrien Chan-Hon-Tong)`

1. **Membres du groupe:**
   - Dada Simeu Cedric Darel
   - Tchakonte Tchiegue Cedrick
2. *Ce notebook explique comment réaliser une attaque sur des modèles de segmentation et fait également une évaluation des performances et de la transférabilité*

Open in colab

1.2 Introduction

Les attaques adversaires représentent une menace sérieuse pour les modèles de vision par ordinateur, en particulier pour les tâches de segmentation d'images. Ces attaques consistent à ajouter un bruit imperceptible à une image pour induire le modèle en erreur, tout en laissant l'image apparemment inchangée pour un observateur humain. Ce projet explore les mécanismes des attaques adversaires sur des modèles de segmentation pré-entraînés, en utilisant le dataset MS-COCO et des modèles issus de la bibliothèque torchvision.

1.3 Contexte et Objectifs

1.3.1 Contexte

La segmentation d'images est une tâche fondamentale en vision par ordinateur, où l'objectif est d'attribuer une étiquette à chaque pixel d'une image. Les modèles de segmentation, tels que DeepLabV3 ou FCN, sont largement utilisés dans des applications critiques comme la conduite autonome ou la médecine. Cependant, ces modèles sont vulnérables aux attaques adversaires, où de petites perturbations peuvent entraîner des prédictions erronées.

1.3.2 Objectifs

Les objectifs de ce projet sont les suivants :

- **Q1** : Implémenter une attaque non ciblée (untargeted attack) sur un modèle de segmentation.
- **Q2** : Implémenter une attaque ciblée (targeted attack) pour forcer le modèle à prédire une classe spécifique.
- **Q3** : Étudier l'impact de la norme de l'attaque sur la performance du modèle.

- **Q4** : Évaluer la transferabilité de l'attaque entre différents modèles.
- **Q5** : Explorer l'efficacité d'une attaque apprise sur un ensemble de réseaux.

1.4 Implémentation

1.4.1 Question 1 : Attaque non Ciblée contre un réseau (Untargeted)

1.4.2 Principe Clé

Perturber les features intermédiaires pour déstabiliser la segmentation.

1.4.3 Détails

- **Perturbation des caractéristiques originales** :
  - la ligne net.classifier = torch.nn.Identity() nous permet de modifier la dernière couche du modèle pour la remplacer par l'identité, ce qui nous permet de récupérer les features intermédiaires du mini-batch initial img via `f_0 = net(normalize(img))["out"]`
  - La ligne `f_0 = f_0[:, torch.randperm(f_0.size(1)), :, :]` mélange aléatoirement les canaux des features originales.
  - Cela crée une "cible aléatoire" dans l'espace des features, brisant leur cohérence spatiale.
- **Optimisation de la perturbation** :
  - La perte MSE entre les features perturbées (f) et les features mélangées (f_0) est minimisée.
  - Effet : Le modèle ne peut plus interpréter correctement les features, ce qui corrompt la segmentation.
- **Contraintes** :
  - La perturbation est limitée à $\pm 8/255$ (norme $L\infty$) pour rester imperceptible.
- **Pourquoi ça marche** : En forçant les features à ressembler à une version désorganisée d'elles-mêmes, on induit des erreurs de segmentation sans cible spécifique.

[ ]:
```
import os
import torch
import torchvision
import matplotlib.pyplot as plt

# Déterminer le device disponible
#device = torch.device("cuda" if torch.cuda.is_available() else "cpu")
device = torch.device("cpu")
print(f"Using device: {device}")

# Télécharger les données si nécessaire
if not os.path.isfile("coco_sample.pth"):
    os.system("wget https://httpmail.onera.fr/21/
 ↪050a4e5c4611d260c1b8035b5dc8617e01A12h/coco_sample.pth")

# Initialiser le modèle sur le device
W = torchvision.models.segmentation.DeepLabV3_ResNet50_Weights.
 ↪COCO_WITH_VOC_LABELS_V1
```

```
net = torchvision.models.segmentation.deeplabv3_resnet50(weights=W).eval().
 ↪to(device)

# Remplacer le classifieur
net.classifier = torch.nn.Identity()

# Normalisation
normalize = torchvision.transforms.Normalize([0.485, 0.456, 0.406], [0.229, 0.
 ↪224, 0.225])

# Charger les images et les déplacer sur le device
img = torch.load("coco_sample.pth", map_location=device)[3:7]  # Charger␣
 ↪directement sur le device
img = torch.nn.functional.interpolate(img, size=(220, 220), mode="bilinear").
 ↪to(device)

# Calcul des caractéristiques cibles
with torch.no_grad():
    f_0 = net(normalize(img))["out"]
    f_0 = f_0[:, torch.randperm(f_0.size(1)), :, :].to(device)  # permet de␣
 ↪S'assurer du device

# Initialiser la perturbation sur le device
attaque = torch.nn.Parameter(torch.zeros(img.shape, device=device))
optimizer = torch.optim.SGD([attaque], lr=0.005) # SGD car on veut une␣
 ↪perturbation "légère"

# Boucle d'attaque
for i in range(10):
    optimizer.zero_grad()

    # Tous les calculs se font sur le device
    x_perturbed = normalize(img + attaque)
    f = net(x_perturbed)["out"]

    loss = ((f - f_0) ** 2).sum()
    print(f"Iter {i}, Loss: {loss.item():.4f}")

    loss.backward()
    optimizer.step()

    # Application des contraintes
    with torch.no_grad():
        attaque.clamp_(-8/255, 8/255)
        attaque.data = torch.clamp(attaque.data, -img, 1 - img)
```

```
Using device: cpu
```

```
<ipython-input-1-985c4a4fda9b>:26: FutureWarning: You are using `torch.load`
with `weights_only=False` (the current default value), which uses the default
pickle module implicitly. It is possible to construct malicious pickle data
which will execute arbitrary code during unpickling (See
https://github.com/pytorch/pytorch/blob/main/SECURITY.md#untrusted-models for
more details). In a future release, the default value for `weights_only` will be
flipped to `True`. This limits the functions that could be executed during
unpickling. Arbitrary objects will no longer be allowed to be loaded via this
mode unless they are explicitly allowlisted by the user via
`torch.serialization.add_safe_globals`. We recommend you start setting
`weights_only=True` for any use case where you don't have full control of the
loaded file. Please open an issue on GitHub for any issues related to this
experimental feature.
  img = torch.load("coco_sample.pth", map_location=device)[3:7]  # Charger
directement sur le device

Iter 0, Loss: 76268104.0000
Iter 1, Loss: 59193700.0000
Iter 2, Loss: 64337624.0000
Iter 3, Loss: 57233560.0000
Iter 4, Loss: 62950600.0000
Iter 5, Loss: 58727268.0000
Iter 6, Loss: 63728116.0000
Iter 7, Loss: 58226552.0000
Iter 8, Loss: 58979008.0000
Iter 9, Loss: 59456816.0000
```

Fonction pour la visualisation de l'attaque

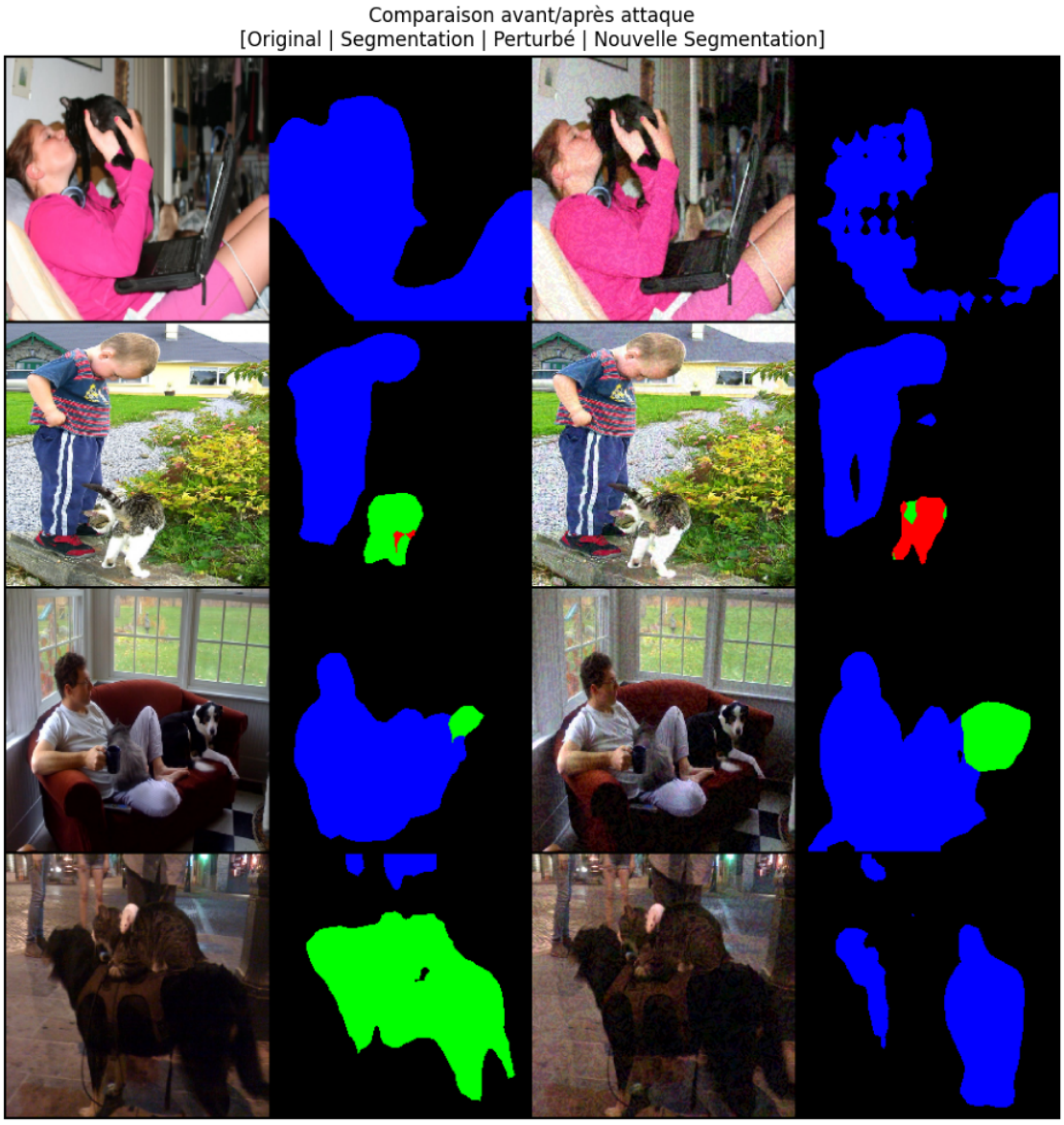
[5]:
```python
def visualize_attack_results(
    attack,
    model,
    original_image,
    normalize_transform,
    fig_size=(16, 16),
    title="Comparaison avant/après attaque\n[Original | Segmentation | Perturbé␣
 ↪| Nouvelle Segmentation]",
    device=torch.device("cuda" if torch.cuda.is_available() else "cpu")
):
    """
    Visualise les résultats d'une attaque adversarial pour la segmentation␣
 ↪sémantique
    Args:
        attack (Tensor): Tenseur de perturbation généré
        model (nn.Module): Modèle de segmentation pour la visualisation
        original_image (Tensor): Image originale non perturbée
        normalize_transform (transforms.Normalize): Transformation de␣
 ↪normalisation
```

```
        fig_size (tuple): Taille de la figure matplotlib
        title (str): Titre du graphique
        device (torch.device): Device utilisé pour les calculs
    """

    def create_pred_mask(z):
        """Mapping des classes vers les canaux RGB"""
        pred = torch.zeros_like(original_image).to(device)
        pred[:, 0, :, :] = (z == 1).float()  # Personnes (bleu)
        pred[:, 1, :, :] = (z == 2).float()  # Chats (rouge)
        pred[:, 2, :, :] = (z == 3).float()  # Chiens (vert)
        return pred

    with torch.no_grad():
        # Segmentation originale
        x_clean = normalize_transform(original_image.to(device))
        z_clean = model(x_clean)['out'][:, [0, 8, 12, 15], :, :]
        _, z_clean = z_clean.max(1)

        # Segmentation perturbée
        x_perturbed = normalize_transform(torch.clamp(original_image + attack,␣
↪0, 1).to(device))
        z_perturbed = model(x_perturbed)['out'][:, [0, 8, 12, 15], :, :]
        _, z_perturbed = z_perturbed.max(1)

    # Création de la visualisation
    visu = torch.cat([
        original_image.to(device),
        create_pred_mask(z_clean),
        torch.clamp(original_image + attack, 0, 1).to(device),
        create_pred_mask(z_perturbed)
    ], dim=-1)

    # Préparation pour matplotlib
    visu_grid = torchvision.utils.make_grid(visu, nrow=1)
    visu_np = visu_grid.permute(1, 2, 0).clamp(0, 1).detach().cpu().numpy()

    # Affichage
    plt.figure(figsize=fig_size)
    plt.imshow(visu_np)
    plt.title(title)
    plt.axis('off')
    plt.show()
```

[ ]:
```
# Initialisation des paramètres
W = torchvision.models.segmentation.DeepLabV3_ResNet50_Weights.
↪COCO_WITH_VOC_LABELS_V1
```

```
net_vis = torchvision.models.segmentation.deeplabv3_resnet50(weights=W).eval().
 ↪cpu()
img = torch.load("coco_sample.pth")[3:7]
img = torch.nn.functional.interpolate(img, size=(220, 220), mode="bilinear")  #␣
 ↪sert à redimensionner l'image pour la visualisation
normalize = torchvision.transforms.Normalize([0.485, 0.456, 0.406], [0.229, 0.
 ↪224, 0.225])

# Appel de la fonction
visualize_attack_results(
    attack=attaque,
    model=net_vis,
    original_image=img,
    normalize_transform=normalize,
    fig_size=(20, 12),
    title="Comparaison avant/après attaque\n[Original | Segmentation | Perturbé␣
 ↪| Nouvelle Segmentation]",
    device = torch.device('cpu')
)
```

```
<ipython-input-3-7c84ae1c342f>:4: FutureWarning: You are using `torch.load` with
`weights_only=False` (the current default value), which uses the default pickle
module implicitly. It is possible to construct malicious pickle data which will
execute arbitrary code during unpickling (See
https://github.com/pytorch/pytorch/blob/main/SECURITY.md#untrusted-models for
more details). In a future release, the default value for `weights_only` will be
flipped to `True`. This limits the functions that could be executed during
unpickling. Arbitrary objects will no longer be allowed to be loaded via this
mode unless they are explicitly allowlisted by the user via
`torch.serialization.add_safe_globals`. We recommend you start setting
`weights_only=True` for any use case where you don't have full control of the
loaded file. Please open an issue on GitHub for any issues related to this
experimental feature.
  img = torch.load("coco_sample.pth")[3:7]
```



1.4.4 Question 2 : Attaque Ciblée (Targeted)

1.4.5 Objectif

Forcer le modèle à confondre tous les objets détectés par des objets spécifiques (ex: “personnes”) sur l’image perturbée. Pour cela, nous avons une image cible4.jpg sur laquelle il y a un groupe de personnes. Cette image va nous servir de “target” pour recupérer les features souhaités.

1.4.6 Mécanismes

- **Construction du masque d’objets** :
  - Étape 1 : Utiliser le modèle de segmentation pour obtenir la prédiction originale

(`orig_seg`).
  - Étape 2 : Générer `object_mask = (orig_seg.argmax(1) != 0)`, qui marque tous les pixels non-fond (objets détectés).
  - Étape 3 : Redimensionner le masque pour correspondre à la résolution des features (`interpolate`).
  - But : Concentrer l'attaque uniquement sur les zones contenant des objets.
- **Alignement des features** :
  - Cible : Les features de l'image perturbée (`features`) doivent correspondre à celles d'une image cible ("cible4.jpg") sur les zones masquées.
- **Loss** :
  - `loss_content = MSE(features * mask, f_target * mask)` → Alignement ciblé.
  - `tv_loss` → Régularisation pour lisser la perturbation.
- **Optimisation** :
  - La perturbation est apprise via Adam, avec un learning rate de 0.001 et epochs=1500.
- **Résultat** : L'image perturbée est segmentée selon la classe cible (ex: "personne") uniquement sur les zones masquées.

1.4.7 Points Forts de l'Approche

- **Ciblage précis** : Le masque object_mask permet de concentrer l'attaque sur les zones critiques.
- **Régularisation TV** : Garantit une perturbation lisse et imperceptible.
- **Feature-space vs Pixel-space** : Comme démontré dans Inkawhich et al. (2019), attaquer les features améliore la transférabilité.

[ ]:
```
# Chargement des données
def load_data(path, size=256):
    transform = torchvision.transforms.Compose([
        torchvision.transforms.Resize((size, size)),
        torchvision.transforms.ToTensor(),
    ])
    if path.endswith('.pth'):
        data = torch.load(path, map_location=device)
        return torch.nn.functional.interpolate(data, size=(size, size),␣
↪mode='bilinear')
    else:
        img = transform(Image.open(path).convert('RGB')).unsqueeze(0)
        return img.to(device)
```

[ ]:
```
import torch
import torchvision
from torch.utils.checkpoint import checkpoint
from PIL import Image
import torch.nn as nn
import torchvision.transforms as transforms
import matplotlib.pyplot as plt

# Configuration
```

```python
class Config:
    resolution = 256
    lr = 0.001
    tv_weight = 0.002
    epochs = 1500

config = Config()
device = torch.device("cuda" if torch.cuda.is_available() else "cpu")

# Extracteur de feature
class FixedFeatureExtractor(nn.Module):
    def __init__(self):
        super().__init__()
        base = torchvision.models.segmentation.
 ↪deeplabv3_resnet50(weights="DEFAULT").backbone
        self.stem = nn.Sequential(base.conv1, base.bn1, base.relu, base.maxpool)
        self.layer1 = base.layer1
        self.layer2 = base.layer2
        self.layer3 = base.layer3
        self.layer4 = base.layer4
        self.downsample_factor = 8

    def forward(self, x):
        x = self.stem(x)
        x = self.layer1(x)
        x = self.layer2(x)  # Retrait du checkpoint pour stabilité
        x = self.layer3(x)
        x = self.layer4(x)
        return x

# Normalisation
normalize = transforms.Normalize([0.485, 0.456, 0.406], [0.229, 0.224, 0.225])

# Chargement des images
person_img = load_data("cible4.jpg")
img = load_data("coco_sample.pth")

# Calcul des caractéristiques cibles
with torch.no_grad():
    net = FixedFeatureExtractor().to(device).eval()
    f_target = net(normalize(person_img))
    target_features = f_target  # Utilisation directe des caractéristiques␣
 ↪spatiales

# Initialisation de l'attaque
attaque = nn.Parameter(torch.randn_like(img) * 0.1)  # Initialisation avec bruit
optimizer = torch.optim.Adam([attaque], lr=config.lr)
```

```python
# Modèle de segmentation pour masque
seg_model = torchvision.models.segmentation.
 ↪deeplabv3_resnet50(weights="DEFAULT").to(device).eval()

# Boucle d'entraînement
for epoch in range(config.epochs):
    optimizer.zero_grad()

    x_perturbed = torch.clamp(img + attaque, 0, 1)
    features = net(normalize(x_perturbed))

    # Génération du masque cible (tous les objets sauf fond)
    with torch.no_grad():
        orig_seg = seg_model(normalize(img))['out']
        object_mask = (orig_seg.argmax(1) != 0).float()  # Cible tous les objets
        object_mask = torch.nn.functional.interpolate(
            object_mask.unsqueeze(1),
            size=features.shape[-2:],
            mode='nearest'
        )

    # Calcul des pertes
    loss_content = nn.MSELoss()(features * object_mask, target_features *␣
 ↪object_mask)
    tv_loss = torch.sum(torch.abs(attaque[:,:,:,:-1] - attaque[:,:,:,1:]))
    loss = loss_content + config.tv_weight * tv_loss

    loss.backward()
    optimizer.step()
    if(epoch)%10==0:
      print(f"Epoch {epoch+1} | Loss: {loss.item():.4f}")
```

```
<ipython-input-4-9a09c7d5700c>:8: FutureWarning: You are using `torch.load` with
`weights_only=False` (the current default value), which uses the default pickle
module implicitly. It is possible to construct malicious pickle data which will
execute arbitrary code during unpickling (See
https://github.com/pytorch/pytorch/blob/main/SECURITY.md#untrusted-models for
more details). In a future release, the default value for `weights_only` will be
flipped to `True`. This limits the functions that could be executed during
unpickling. Arbitrary objects will no longer be allowed to be loaded via this
mode unless they are explicitly allowlisted by the user via
`torch.serialization.add_safe_globals`. We recommend you start setting
`weights_only=True` for any use case where you don't have full control of the
loaded file. Please open an issue on GitHub for any issues related to this
experimental feature.
  data = torch.load(path, map_location=device)
```

```
Epoch 1 | Loss: 397.7128
Epoch 11 | Loss: 352.7386
Epoch 21 | Loss: 311.5836
Epoch 31 | Loss: 274.1750
Epoch 41 | Loss: 240.3760
Epoch 51 | Loss: 210.0227
Epoch 61 | Loss: 182.9433
Epoch 71 | Loss: 158.9543
Epoch 81 | Loss: 137.8102
Epoch 91 | Loss: 119.2602
Epoch 101 | Loss: 103.0944
Epoch 111 | Loss: 89.0442
Epoch 121 | Loss: 76.8872
Epoch 131 | Loss: 66.4130
Epoch 141 | Loss: 57.4081
Epoch 151 | Loss: 49.6839
Epoch 161 | Loss: 43.0750
Epoch 171 | Loss: 37.4307
Epoch 181 | Loss: 32.6114
Epoch 191 | Loss: 28.4993
Epoch 201 | Loss: 24.9869
Epoch 211 | Loss: 21.9867
Epoch 221 | Loss: 19.4171
Epoch 231 | Loss: 17.2153
Epoch 241 | Loss: 15.3260
Epoch 251 | Loss: 13.6984
Epoch 261 | Loss: 12.2916
Epoch 271 | Loss: 11.0722
Epoch 281 | Loss: 10.0112
Epoch 291 | Loss: 9.0889
Epoch 301 | Loss: 8.2820
Epoch 311 | Loss: 7.5735
Epoch 321 | Loss: 6.9492
Epoch 331 | Loss: 6.3973
Epoch 341 | Loss: 5.9085
Epoch 351 | Loss: 5.4736
Epoch 361 | Loss: 5.0856
Epoch 371 | Loss: 4.7383
Epoch 381 | Loss: 4.4271
Epoch 391 | Loss: 4.1475
Epoch 401 | Loss: 3.8956
Epoch 411 | Loss: 3.6679
Epoch 421 | Loss: 3.4627
Epoch 431 | Loss: 3.2757
Epoch 441 | Loss: 3.1063
Epoch 451 | Loss: 2.9516
Epoch 461 | Loss: 2.8099
Epoch 471 | Loss: 2.6806
```

```
Epoch 481 | Loss: 2.5622
Epoch 491 | Loss: 2.4530
Epoch 501 | Loss: 2.3533
Epoch 511 | Loss: 2.2607
Epoch 521 | Loss: 2.1760
Epoch 531 | Loss: 2.0952
Epoch 541 | Loss: 2.0238
Epoch 551 | Loss: 1.9556
Epoch 561 | Loss: 1.8922
Epoch 571 | Loss: 1.8336
Epoch 581 | Loss: 1.7791
Epoch 591 | Loss: 1.7283
Epoch 601 | Loss: 1.6804
Epoch 611 | Loss: 1.6371
Epoch 621 | Loss: 1.5950
Epoch 631 | Loss: 1.5576
Epoch 641 | Loss: 1.5212
Epoch 651 | Loss: 1.4874
Epoch 661 | Loss: 1.4561
Epoch 671 | Loss: 1.4259
Epoch 681 | Loss: 1.3973
Epoch 691 | Loss: 1.3721
Epoch 701 | Loss: 1.3471
Epoch 711 | Loss: 1.3237
Epoch 721 | Loss: 1.3008
Epoch 731 | Loss: 1.2792
Epoch 741 | Loss: 1.2590
Epoch 751 | Loss: 1.2408
Epoch 761 | Loss: 1.2232
Epoch 771 | Loss: 1.2061
Epoch 781 | Loss: 1.1907
Epoch 791 | Loss: 1.1749
Epoch 801 | Loss: 1.1602
Epoch 811 | Loss: 1.1477
Epoch 821 | Loss: 1.1335
Epoch 831 | Loss: 1.1211
Epoch 841 | Loss: 1.1090
Epoch 851 | Loss: 1.0973
Epoch 861 | Loss: 1.0860
Epoch 871 | Loss: 1.0753
Epoch 881 | Loss: 1.0652
Epoch 891 | Loss: 1.0569
Epoch 901 | Loss: 1.0462
Epoch 911 | Loss: 1.0375
Epoch 921 | Loss: 1.0293
Epoch 931 | Loss: 1.0206
Epoch 941 | Loss: 1.0130
Epoch 951 | Loss: 1.0050
```

```
Epoch 961 | Loss: 0.9983
Epoch 971 | Loss: 0.9917
Epoch 981 | Loss: 0.9848
Epoch 991 | Loss: 0.9778
Epoch 1001 | Loss: 0.9713
Epoch 1011 | Loss: 0.9652
Epoch 1021 | Loss: 0.9593
Epoch 1031 | Loss: 0.9544
Epoch 1041 | Loss: 0.9490
Epoch 1051 | Loss: 0.9446
Epoch 1061 | Loss: 0.9381
Epoch 1071 | Loss: 0.9334
Epoch 1081 | Loss: 0.9285
Epoch 1091 | Loss: 0.9246
Epoch 1101 | Loss: 0.9198
Epoch 1111 | Loss: 0.9150
Epoch 1121 | Loss: 0.9114
Epoch 1131 | Loss: 0.9068
Epoch 1141 | Loss: 0.9030
Epoch 1151 | Loss: 0.8996
Epoch 1161 | Loss: 0.8952
Epoch 1171 | Loss: 0.8916
Epoch 1181 | Loss: 0.8876
Epoch 1191 | Loss: 0.8834
Epoch 1201 | Loss: 0.8802
Epoch 1211 | Loss: 0.8772
Epoch 1221 | Loss: 0.8741
Epoch 1231 | Loss: 0.8709
Epoch 1241 | Loss: 0.8683
Epoch 1251 | Loss: 0.8652
Epoch 1261 | Loss: 0.8619
Epoch 1271 | Loss: 0.8595
Epoch 1281 | Loss: 0.8564
Epoch 1291 | Loss: 0.8531
Epoch 1301 | Loss: 0.8512
Epoch 1311 | Loss: 0.8485
Epoch 1321 | Loss: 0.8466
Epoch 1331 | Loss: 0.8443
Epoch 1341 | Loss: 0.8420
Epoch 1351 | Loss: 0.8389
Epoch 1361 | Loss: 0.8360
Epoch 1371 | Loss: 0.8349
Epoch 1381 | Loss: 0.8314
Epoch 1391 | Loss: 0.8310
Epoch 1401 | Loss: 0.8275
Epoch 1411 | Loss: 0.8268
Epoch 1421 | Loss: 0.8242
Epoch 1431 | Loss: 0.8222
```

```
Epoch 1441 | Loss: 0.8204
Epoch 1451 | Loss: 0.8180
Epoch 1461 | Loss: 0.8154
Epoch 1471 | Loss: 0.8145
Epoch 1481 | Loss: 0.8128
Epoch 1491 | Loss: 0.8108
```

[ ]:
```python
# Initialisation des paramètres
W = torchvision.models.segmentation.DeepLabV3_ResNet50_Weights.
 ↪COCO_WITH_VOC_LABELS_V1
net_vis = torchvision.models.segmentation.deeplabv3_resnet50(weights=W).eval().
 ↪cpu()
img = load_data("coco_sample.pth")
normalize = transforms.Normalize([0.485, 0.456, 0.406], [0.229, 0.224, 0.225])

# Appel de la fonction
visualize_attack_results(
    attack=attaque,
    model=net_vis,
    original_image=img,
    normalize_transform=normalize,
    fig_size=(20, 12),
    title="Comparaison avant/après attaque\n[Original | Segmentation | Perturbé␣
 ↪| Nouvelle Segmentation]",
    device = torch.device('cpu')
)
```

```
<ipython-input-4-9a09c7d5700c>:8: FutureWarning: You are using `torch.load` with
`weights_only=False` (the current default value), which uses the default pickle
module implicitly. It is possible to construct malicious pickle data which will
execute arbitrary code during unpickling (See
https://github.com/pytorch/pytorch/blob/main/SECURITY.md#untrusted-models for
more details). In a future release, the default value for `weights_only` will be
flipped to `True`. This limits the functions that could be executed during
unpickling. Arbitrary objects will no longer be allowed to be loaded via this
mode unless they are explicitly allowlisted by the user via
`torch.serialization.add_safe_globals`. We recommend you start setting
`weights_only=True` for any use case where you don't have full control of the
loaded file. Please open an issue on GitHub for any issues related to this
experimental feature.
  data = torch.load(path, map_location=device)
```

Comparaison avant/après attaque
[Original | Segmentation | Perturbé | Nouvelle Segmentation]

2e methode: Ici, on va implémenter une attaque adversarial ciblée sur un modèle de segmentation DeepLabV3. Pour se faire, on va plutôt se servir de la méthode FGSM(Fast Gradient Sign method), utilisée pour perturber les images et augmenter la détection d'une classe spécifique. Une attention particulière a été portée à la normalisation des images et à la gestion des gradients pour assurer une perturbation efficace. Mais d'abord, faisons le point sur cette méthode.

La méthode Fast Gradient Sign Method (FGSM) est une technique d'attaque adversarial one-step conçue pour tromper les modèles de deep learning en ajoutant une perturbation imperceptible à une image, en exploitant les gradients du modèle par rapport à l'entrée pour générer une perturbation directionnelle. Son objectif est de maximiser la perte du modèle en suivant le signe du gradient, ce qui induit une erreur de prédiction.

1. **Calcul du gradient:** Pour une image d'entrée x, on calcule le gradient J(,x,y) de la fonction de perte J par rapport à x, où :

- représente les paramètres du modèle
- y est la cible (classe réelle ou ciblée)

Dans une attaque ciblée, la perte est inversée :

```
J = -score_classe_cible
```

Ceci pousse le modèle à prédire la classe cible.

2. **Perturbation:** La perturbation est générée en multipliant le signe du gradient par un hyperparamètre qui contrôle l'intensité :

```
x_adv = x +   · sign( J)
```

L'image adversariale x_adv est ensuite clampée entre 0 et 1 pour rester valide.

Cette méthode permet de : - Générer rapidement une perturbation efficace - Contrôler l'amplitude de la modification via - Maintenir la validité des valeurs de pixels - Exploiter les directions de plus forte pente du modèle

[1]:
```
import torch
import torchvision
from torchvision.models.segmentation import deeplabv3_resnet50,␣
 ↪DeepLabV3_ResNet50_Weights
import numpy as np
import matplotlib.pyplot as plt
import os
import requests
from PIL import Image

# Télécharge un fichier depuis une URL via requests. Utilise un stream pour␣
 ↪gérer les gros fichiers (chargement par blocs de 1024 octets), Sauvegarde␣
 ↪localement le fichier pour éviter de retélécharger à chaque exécution.
```

```python
def download_file(url, dest_path):
    print(f"Téléchargement du fichier depuis {url}...")
    response = requests.get(url, stream=True)
    with open(dest_path, "wb") as f:
        for chunk in response.iter_content(chunk_size=1024):
            f.write(chunk)
    print("Téléchargement terminé.")

# Charge des images prétraitées depuis un fichier .pth.
def load_custom_coco_data(file_path):
    print("Chargement des images COCO depuis le fichier PTH...")
    return torch.load(file_path)


"""
Objectif : Perturbe une image pour maximiser la classe target_class en segmentation.
Étapes :
    1. Clone l'image et active le calcul de gradient.
    2. Normalise l'image (moyenne/écart-type du modèle).
    3. Calcule la perte comme l'opposé du score moyen de la classe cible (pour maximiser celle-ci).
    4. Rétropropage le gradient et met à jour l'image avec le signe du gradient.
    5. Clampe les pixels entre 0 et 1 pour rester dans un espace valide.
"""
def apply_targeted_attack(model, image, target_class, epsilon=0.03, mean=None, std=None):
    image = image.clone().detach().requires_grad_(True)  # 1.
    image_normalized = (image - mean[:, None, None]) / std[:, None, None] # 2.
    output = model(image_normalized.unsqueeze(0))["out"]
    loss = -output[:, target_class, :, :].mean() # 3.
    model.zero_grad()  # 4.
    loss.backward()
    perturbed_image = image - epsilon * image.grad.sign()
    return torch.clamp(perturbed_image, 0, 1).detach() #5.

def create_pred_mask(z):
    pred = torch.zeros((z.size(0), 3, z.size(1), z.size(2)))
    pred[:, 0, :, :] = (z == 1).float()  # Personne (bleu)
    pred[:, 1, :, :] = (z == 2).float()  # Chat (rouge)
    pred[:, 2, :, :] = (z == 3).float()  # Chien (vert)
    return pred
```

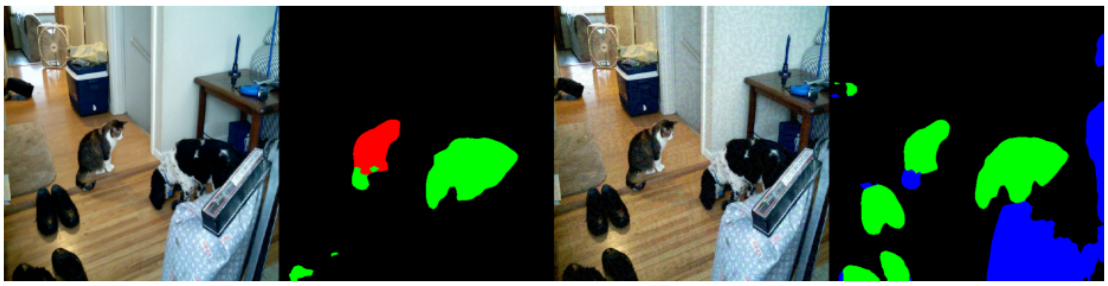
Exécutons et verifions les resultats, en choisissant comme modèle DeepLabV3_resnet50, et comme classe cible, la classe Personne(target_class = 15)

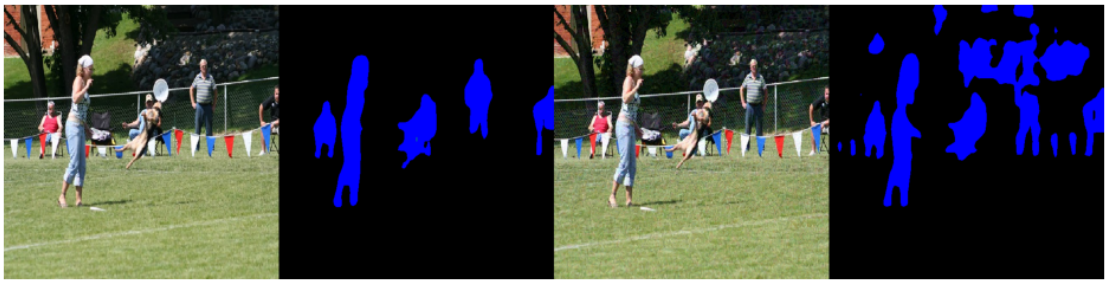
[ ]:
```python
# Configuration
coco_pth_path = "coco_sample.pth"
try:
    download_file("https://httpmail.onera.fr/21/
 ↪050a4e5c4611d260c1b8035b5dc8617e01A12h/coco_sample.pth", coco_pth_path)
except Exception as e:
    print(f"Failed to download the file: {e}")
    if not os.path.isfile(coco_pth_path):
        raise FileNotFoundError(f"The file {coco_pth_path} does not exist␣
 ↪locally. Please provide a valid file path.")

# Chargement du modèle avec ResNet50 comme backbone
weights = DeepLabV3_ResNet50_Weights.COCO_WITH_VOC_LABELS_V1
model = deeplabv3_resnet50(weights=weights).eval()
mean = torch.tensor(weights.transforms().mean)
std = torch.tensor(weights.transforms().std)

# Chargement des données
images = load_custom_coco_data(coco_pth_path)

# Paramètres d'attaque
epsilon = 0.03 # plus epsilon est grand, plus l'attaque est visible
target_class = 15  # 15:Personne, 8:Chat, 12:Chien

for i in range(min(10, len(images))):
    img = images[i]
    adv_img = apply_targeted_attack(model, img, target_class, epsilon, mean,␣
 ↪std)

    # Calcul des prédictions
    with torch.no_grad():
        # Original
        x_clean = (img - mean[:, None, None]) / std[:, None, None]
        z_clean = model(x_clean.unsqueeze(0))["out"][:, [0,8,12,15], :, :]
        _, z_clean = z_clean.max(1)

        # Adversaire
        x_adv = (adv_img - mean[:, None, None]) / std[:, None, None]
        z_adv = model(x_adv.unsqueeze(0))["out"][:, [0,8,12,15], :, :]
        _, z_adv = z_adv.max(1)

    # Visualisation
    visu = torch.cat([
        img.unsqueeze(0),
        create_pred_mask(z_clean),
        adv_img.unsqueeze(0),
        create_pred_mask(z_adv)
```

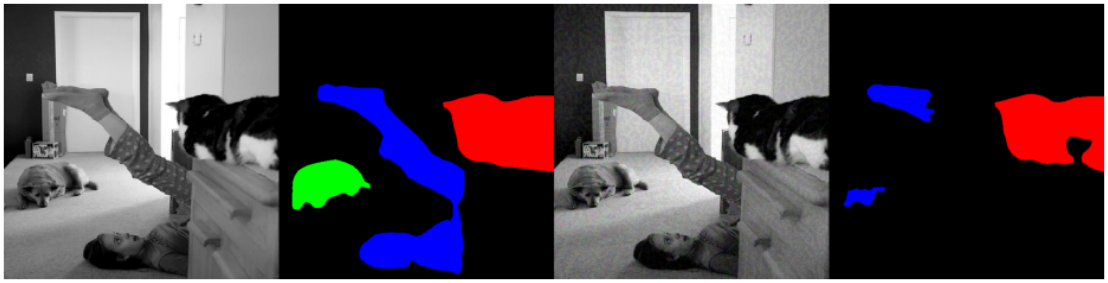
```
    ], dim=-1)

    visu = torchvision.utils.make_grid(visu, nrow=1).permute(1, 2, 0).
↪clamp(0,1).numpy()

    plt.figure(figsize=(12, 6))
    plt.imshow(visu)
    plt.axis('off')
    plt.show()
```

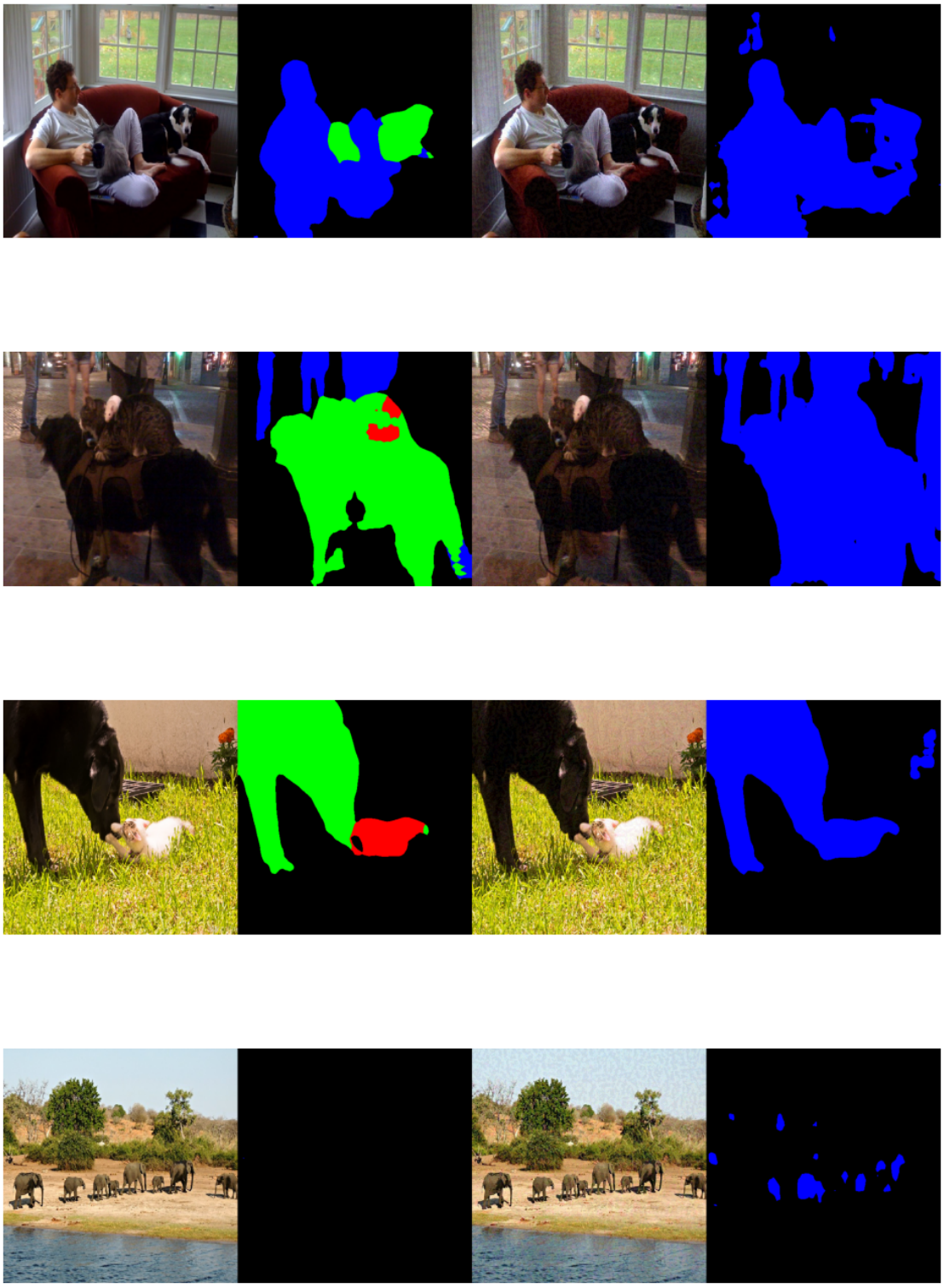
```
Téléchargement du fichier depuis https://httpmail.onera.fr/21/050a4e5c4611d260c1
b8035b5dc8617e01A12h/coco_sample.pth…
Téléchargement terminé.
Chargement des images COCO depuis le fichier PTH…
```

```
<ipython-input-7-b08cebc44114>:22: FutureWarning: You are using `torch.load`
with `weights_only=False` (the current default value), which uses the default
pickle module implicitly. It is possible to construct malicious pickle data
which will execute arbitrary code during unpickling (See
https://github.com/pytorch/pytorch/blob/main/SECURITY.md#untrusted-models for
more details). In a future release, the default value for `weights_only` will be
flipped to `True`. This limits the functions that could be executed during
unpickling. Arbitrary objects will no longer be allowed to be loaded via this
mode unless they are explicitly allowlisted by the user via
`torch.serialization.add_safe_globals`. We recommend you start setting
`weights_only=True` for any use case where you don't have full control of the
loaded file. Please open an issue on GitHub for any issues related to this
experimental feature.
  return torch.load(file_path)
```









Pour aller plus loin, on peut appliquer également cette méthode pour la classe chat (target_class = 8), afin de mieux apprécier nos résultats

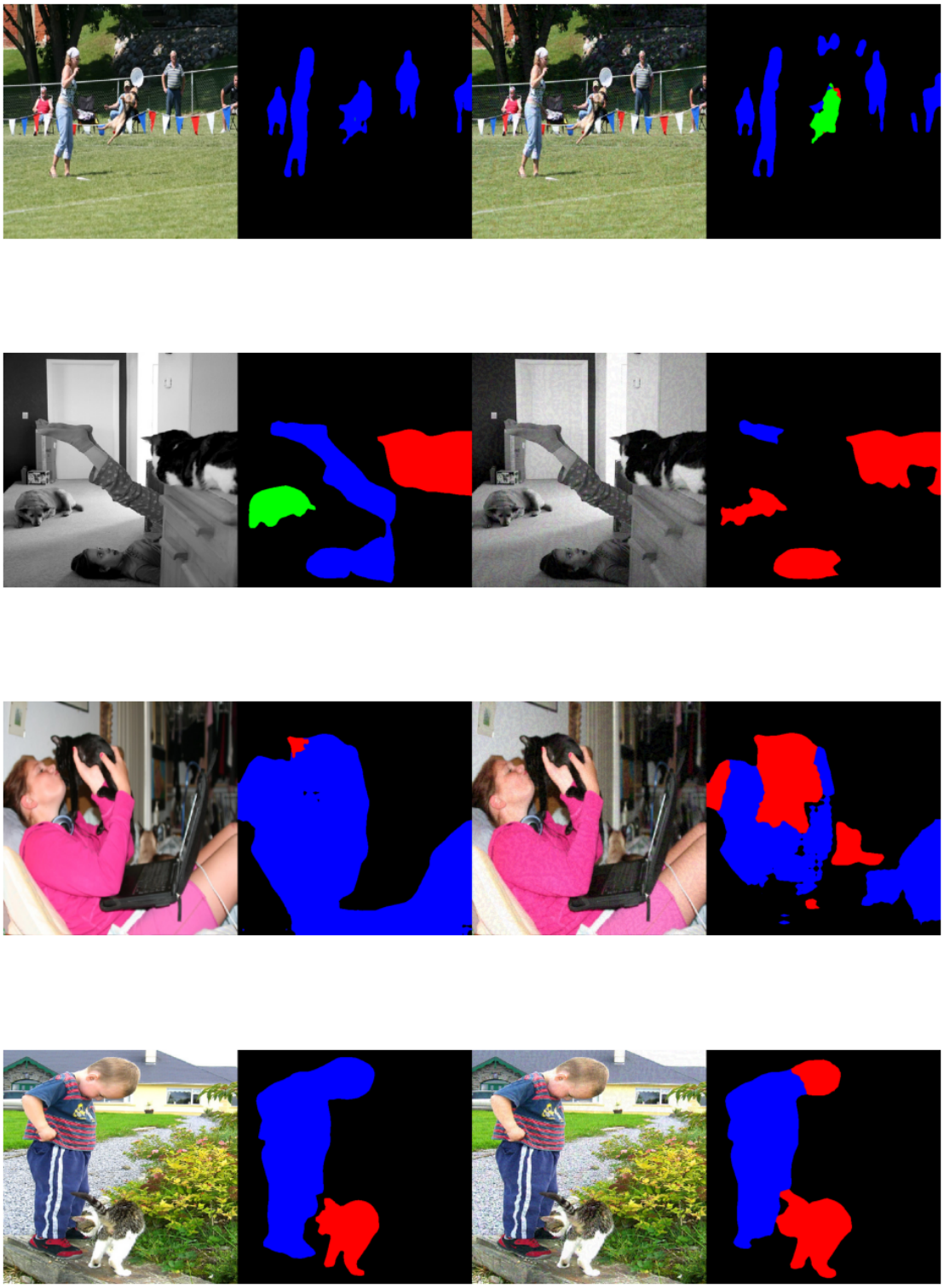
[3]:
```
# Configuration
coco_pth_path = "coco_sample.pth"
try:
    download_file("https://httpmail.onera.fr/21/
 ↪050a4e5c4611d260c1b8035b5dc8617e01A12h/coco_sample.pth", coco_pth_path)
except Exception as e:
    print(f"Failed to download the file: {e}")
    if not os.path.isfile(coco_pth_path):
        raise FileNotFoundError(f"The file {coco_pth_path} does not exist␣
 ↪locally. Please provide a valid file path.")

# Chargement du modèle avec ResNet50 comme backbone
weights = DeepLabV3_ResNet50_Weights.COCO_WITH_VOC_LABELS_V1
model = deeplabv3_resnet50(weights=weights).eval()
mean = torch.tensor(weights.transforms().mean)
std = torch.tensor(weights.transforms().std)

# Chargement des données
images = load_custom_coco_data(coco_pth_path)

# Paramètres d'attaque
epsilon = 0.03 # plus epsilon est grand, plus l'attaque est visible
target_class = 8  # 15:Personne, 8:Chat, 12:Chien

for i in range(min(10, len(images))):
    img = images[i]
    adv_img = apply_targeted_attack(model, img, target_class, epsilon, mean,␣
 ↪std)

    # Calcul des prédictions
    with torch.no_grad():
        # Original
        x_clean = (img - mean[:, None, None]) / std[:, None, None]
        z_clean = model(x_clean.unsqueeze(0))["out"][:, [0,8,12,15], :, :]
        _, z_clean = z_clean.max(1)

        # Adversaire
        x_adv = (adv_img - mean[:, None, None]) / std[:, None, None]
        z_adv = model(x_adv.unsqueeze(0))["out"][:, [0,8,12,15], :, :]
        _, z_adv = z_adv.max(1)

    # Visualisation
    visu = torch.cat([
        img.unsqueeze(0),
        create_pred_mask(z_clean),
        adv_img.unsqueeze(0),
        create_pred_mask(z_adv)
```

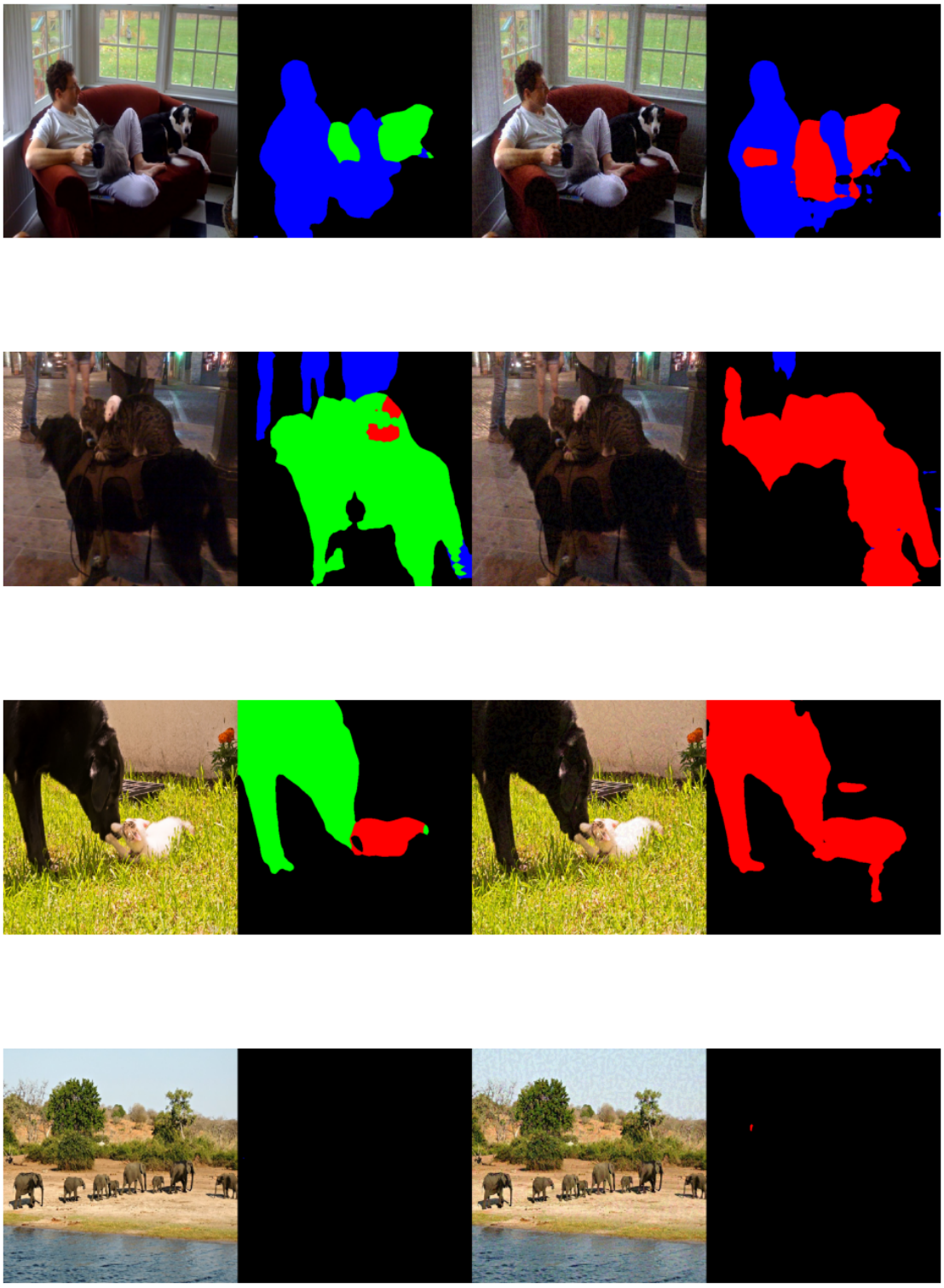
```
    ], dim=-1)

    visu = torchvision.utils.make_grid(visu, nrow=1).permute(1, 2, 0).
↪clamp(0,1).numpy()

    plt.figure(figsize=(12, 6))
    plt.imshow(visu)
    plt.axis('off')
    plt.show()
```

```
Téléchargement du fichier depuis https://httpmail.onera.fr/21/050a4e5c4611d260c1
b8035b5dc8617e01A12h/coco_sample.pth…
Failed to download the file: HTTPSConnectionPool(host='httpmail.onera.fr',
port=443): Max retries exceeded with url:
/21/050a4e5c4611d260c1b8035b5dc8617e01A12h/coco_sample.pth (Caused by
NameResolutionError("<urllib3.connection.HTTPSConnection object at
0x7f78f5bb5610>: Failed to resolve 'httpmail.onera.fr' ([Errno -2] Name or
service not known)"))
```

```
Downloading: "https://download.pytorch.org/models/deeplabv3_resnet50_coco-
cd0a2569.pth" to /root/.cache/torch/hub/checkpoints/deeplabv3_resnet50_coco-
cd0a2569.pth
100%|      | 161M/161M [00:01<00:00, 158MB/s]
<ipython-input-1-b08cebc44114>:22: FutureWarning: You are using `torch.load`
with `weights_only=False` (the current default value), which uses the default
pickle module implicitly. It is possible to construct malicious pickle data
which will execute arbitrary code during unpickling (See
https://github.com/pytorch/pytorch/blob/main/SECURITY.md#untrusted-models for
more details). In a future release, the default value for `weights_only` will be
flipped to `True`. This limits the functions that could be executed during
unpickling. Arbitrary objects will no longer be allowed to be loaded via this
mode unless they are explicitly allowlisted by the user via
`torch.serialization.add_safe_globals`. We recommend you start setting
`weights_only=True` for any use case where you don't have full control of the
loaded file. Please open an issue on GitHub for any issues related to this
experimental feature.
  return torch.load(file_path)
```

```
Chargement des images COCO depuis le fichier PTH…
```





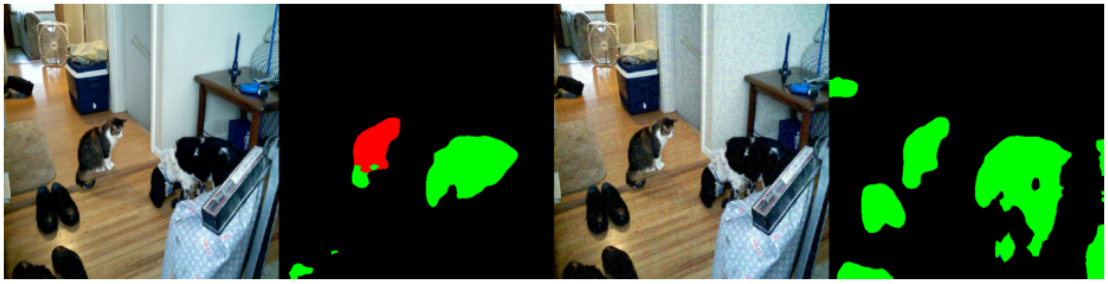
on peut également appliquer également cette méthode pour la classe chien(target_class = 12), afin de mieux apprécier nos résultats

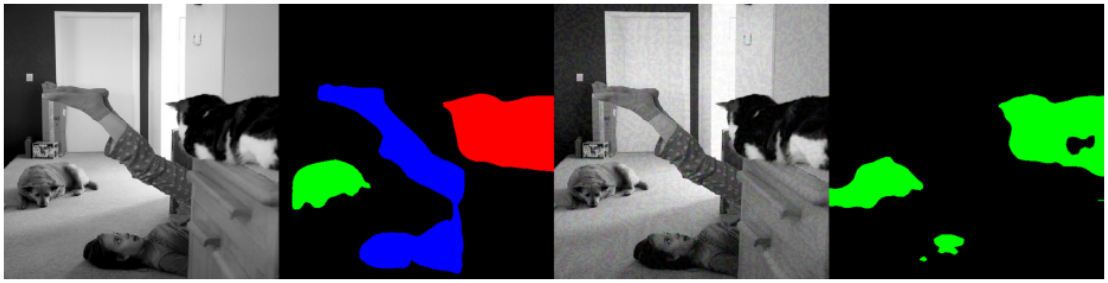
[4]:
```
# Configuration
coco_pth_path = "coco_sample.pth"
try:
    download_file("https://httpmail.onera.fr/21/
 ↪050a4e5c4611d260c1b8035b5dc8617e01A12h/coco_sample.pth", coco_pth_path)
except Exception as e:
    print(f"Failed to download the file: {e}")
    if not os.path.isfile(coco_pth_path):
        raise FileNotFoundError(f"The file {coco_pth_path} does not exist␣
 ↪locally. Please provide a valid file path.")

# Chargement du modèle avec ResNet50 comme backbone
weights = DeepLabV3_ResNet50_Weights.COCO_WITH_VOC_LABELS_V1
model = deeplabv3_resnet50(weights=weights).eval()
mean = torch.tensor(weights.transforms().mean)
std = torch.tensor(weights.transforms().std)

# Chargement des données
images = load_custom_coco_data(coco_pth_path)

# Paramètres d'attaque
epsilon = 0.03 # plus epsilon est grand, plus l'attaque est visible
target_class = 12  # 15:Personne, 8:Chat, 12:Chien

for i in range(min(10, len(images))):
    img = images[i]
    adv_img = apply_targeted_attack(model, img, target_class, epsilon, mean,␣
 ↪std)

    # Calcul des prédictions
    with torch.no_grad():
        # Original
        x_clean = (img - mean[:, None, None]) / std[:, None, None]
        z_clean = model(x_clean.unsqueeze(0))["out"][:, [0,8,12,15], :, :]
        _, z_clean = z_clean.max(1)

        # Adversaire
        x_adv = (adv_img - mean[:, None, None]) / std[:, None, None]
        z_adv = model(x_adv.unsqueeze(0))["out"][:, [0,8,12,15], :, :]
        _, z_adv = z_adv.max(1)

    # Visualisation
    visu = torch.cat([
        img.unsqueeze(0),
        create_pred_mask(z_clean),
        adv_img.unsqueeze(0),
        create_pred_mask(z_adv)
```

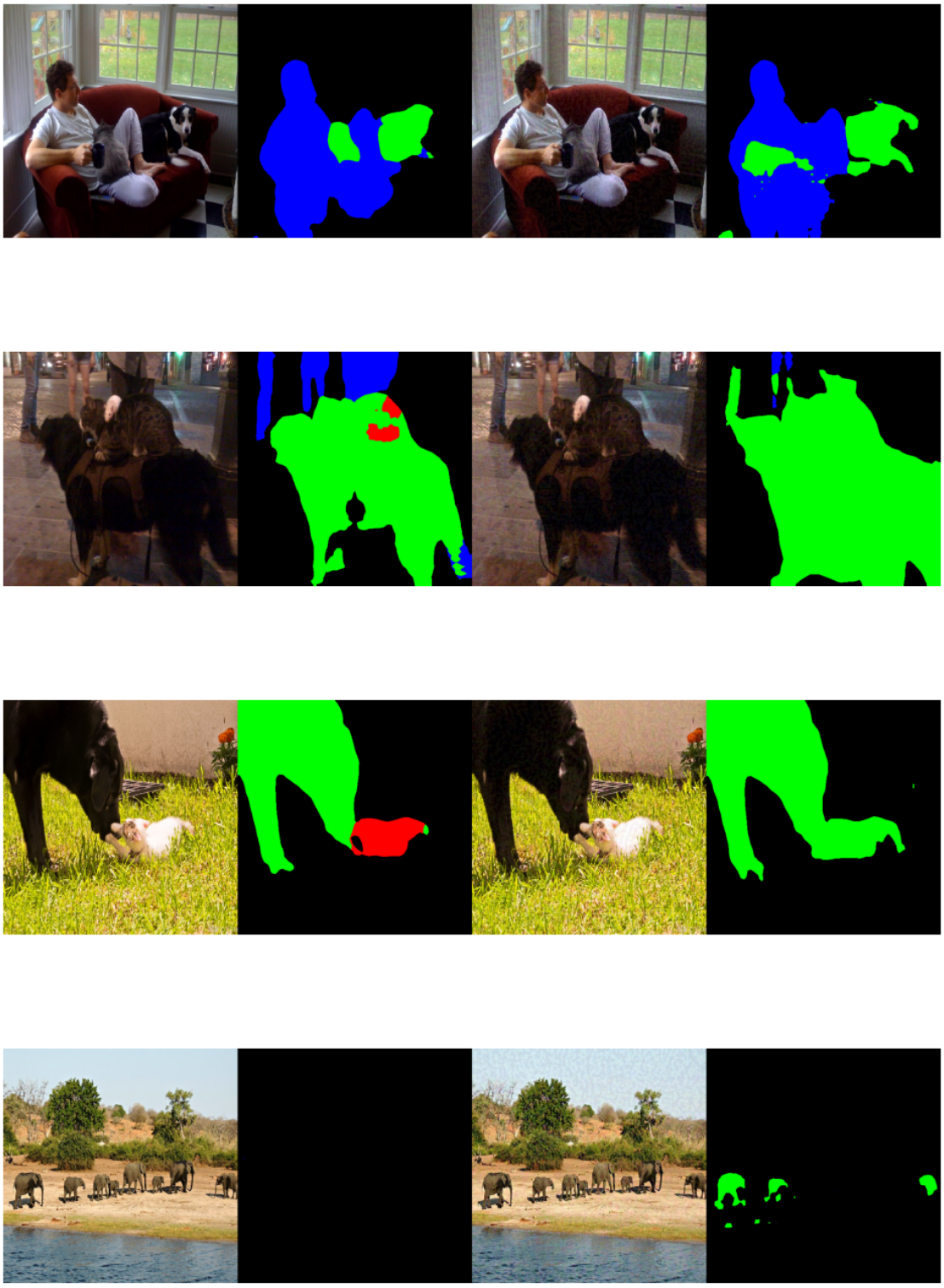
```
    ], dim=-1)

    visu = torchvision.utils.make_grid(visu, nrow=1).permute(1, 2, 0).
↪clamp(0,1).numpy()

    plt.figure(figsize=(12, 6))
    plt.imshow(visu)
    plt.axis('off')
    plt.show()
```

```
Téléchargement du fichier depuis https://httpmail.onera.fr/21/050a4e5c4611d260c1
b8035b5dc8617e01A12h/coco_sample.pth…
Failed to download the file: HTTPSConnectionPool(host='httpmail.onera.fr',
port=443): Max retries exceeded with url:
/21/050a4e5c4611d260c1b8035b5dc8617e01A12h/coco_sample.pth (Caused by
NameResolutionError("<urllib3.connection.HTTPSConnection object at
0x7f78e3f37c10>: Failed to resolve 'httpmail.onera.fr' ([Errno -2] Name or
service not known)"))
Chargement des images COCO depuis le fichier PTH…
```

```
<ipython-input-1-b08cebc44114>:22: FutureWarning: You are using `torch.load`
with `weights_only=False` (the current default value), which uses the default
pickle module implicitly. It is possible to construct malicious pickle data
which will execute arbitrary code during unpickling (See
https://github.com/pytorch/pytorch/blob/main/SECURITY.md#untrusted-models for
more details). In a future release, the default value for `weights_only` will be
flipped to `True`. This limits the functions that could be executed during
unpickling. Arbitrary objects will no longer be allowed to be loaded via this
mode unless they are explicitly allowlisted by the user via
`torch.serialization.add_safe_globals`. We recommend you start setting
`weights_only=True` for any use case where you don't have full control of the
loaded file. Please open an issue on GitHub for any issues related to this
experimental feature.
  return torch.load(file_path)
```







Conclusion:

Ainsi, cette méthode est assez simple à implémenter, rapide à exécuter et efficace pour manipuler les

prédictions d'un modèle tout en appliquant des perturbations minimales aux entrées. elle permet également de contrôler la taille de ces perturbations grâce à un paramètre ajustable, tout en étant adaptable à diverses architectures de réseaux de neurones. De plus, l'utilisation des gradients offre une transparence sur l'impact des variations d'entrée, ce qui est précieux pour étudier la robustesse des modèles.

1.4.8 Question 3 : Impact de la Norme ($L\infty$)

Objectif Évaluer le compromis entre invisibilité et efficacité.

Méthode

- Génération d'attaques pour différents (0.3, 0.04).
- Contrainte : `attaque.clamp_(- , )` pour limiter la perturbation.

Observation

- = 0.6 → le bruit est légèrement plus visible et la loss plus faible (0.0150)
- = 0.04 → Perturbation quasi-invisible mais succès quasiment identique.

[2]:
```
import torch
import torchvision
from torch.utils.checkpoint import checkpoint
from PIL import Image
import torch.nn as nn
import torchvision.transforms as transforms
import matplotlib.pyplot as plt

# Configuration
class Config:
    resolution = 256
    lr = 0.001
    tv_weight = 0.002
    epochs = 1500
    epsilons = [0.6, 0.04]#, 0.05, 0.08, 0.1]   Différents niveaux de␣
 ↪perturbation

config = Config()
device = torch.device("cuda" if torch.cuda.is_available() else "cpu")

# Chargement des données
def load_data(path, size=256):
    transform = transforms.Compose([
        transforms.Resize((size, size)),
        transforms.ToTensor(),
    ])
    if path.endswith('.pth'):
        data = torch.load(path, map_location=device)
```

```
        return torch.nn.functional.interpolate(data, size=(size, size),␣
 ↪mode='bilinear')
    else:
        img = transform(Image.open(path).convert('RGB')).unsqueeze(0)
        return img.to(device)

# Normalisation
normalize = transforms.Normalize([0.485, 0.456, 0.406], [0.229, 0.224, 0.225])

# Modèle de segmentation
seg_model = torchvision.models.segmentation.
 ↪deeplabv3_resnet50(weights="DEFAULT").to(device).eval()

# Fonction d'attaque avec contrainte de norme
def run_attack(epsilon):
    # Chargement des données
    person_img = load_data("cible4.jpg")
    img = load_data("coco_sample.pth")

    # Initialisation perturbation
    attaque = nn.Parameter(torch.zeros_like(img, device=device))
    optimizer = torch.optim.Adam([attaque], lr=config.lr)

    # Masque des objets originaux
    with torch.no_grad():
        orig_seg = seg_model(normalize(img))['out']
        object_mask = (orig_seg.argmax(1) != 0).cpu()

    # Entraînement
    for epoch in range(config.epochs):
        optimizer.zero_grad()

        # Contrainte de norme Linf
        attaque.data.clamp_(-epsilon, epsilon)

        x_perturbed = torch.clamp(img + attaque, 0, 1)
        output = seg_model(normalize(x_perturbed))['out']

        # Calcul de la loss (cible : classe personne)
        loss = torch.nn.functional.cross_entropy(
            output,
            torch.full_like(output.argmax(1), 15, device=device) * object_mask.
 ↪to(device)
        )
        if(epoch)%50==0:
          print(f"Epoch {epoch} | Loss: {loss.item():.4f}")
```

```
        loss.backward()
        optimizer.step()

    # Évaluation
    # with torch.no_grad():
    #     final_seg = seg_model(normalize(x_perturbed))['out'].argmax(1)
        #success_rate = ((final_seg == 15) & object_mask.to(device)).float().
↪mean().item()

    return attaque
```

[3]:
```
# Exécution pour différents epsilons
results = []
for eps in config.epsilons:
    print(f"Epsilon: {eps:.3f}")
    actual_eps = run_attack(eps)
    results.append((eps, actual_eps))
    print(f"Epsilon: {eps:.3f} ok")
```

```
Epsilon: 0.600

<ipython-input-2-08d6d31b05da>:27: FutureWarning: You are using `torch.load`
with `weights_only=False` (the current default value), which uses the default
pickle module implicitly. It is possible to construct malicious pickle data
which will execute arbitrary code during unpickling (See
https://github.com/pytorch/pytorch/blob/main/SECURITY.md#untrusted-models for
more details). In a future release, the default value for `weights_only` will be
flipped to `True`. This limits the functions that could be executed during
unpickling. Arbitrary objects will no longer be allowed to be loaded via this
mode unless they are explicitly allowlisted by the user via
`torch.serialization.add_safe_globals`. We recommend you start setting
`weights_only=True` for any use case where you don't have full control of the
loaded file. Please open an issue on GitHub for any issues related to this
experimental feature.
  data = torch.load(path, map_location=device)

Epoch 0 | Loss: 0.8141
Epoch 50 | Loss: 0.0417
Epoch 100 | Loss: 0.0314
Epoch 150 | Loss: 0.0273
Epoch 200 | Loss: 0.0249
Epoch 250 | Loss: 0.0233
Epoch 300 | Loss: 0.0221
Epoch 350 | Loss: 0.0212
Epoch 400 | Loss: 0.0205
Epoch 450 | Loss: 0.0198
Epoch 500 | Loss: 0.0193
Epoch 550 | Loss: 0.0188
```

```
Epoch 600 | Loss: 0.0184
Epoch 650 | Loss: 0.0181
Epoch 700 | Loss: 0.0177
Epoch 750 | Loss: 0.0174
Epoch 800 | Loss: 0.0172
Epoch 850 | Loss: 0.0169
Epoch 900 | Loss: 0.0167
Epoch 950 | Loss: 0.0165
Epoch 1000 | Loss: 0.0163
Epoch 1050 | Loss: 0.0161
Epoch 1100 | Loss: 0.0160
Epoch 1150 | Loss: 0.0158
Epoch 1200 | Loss: 0.0157
Epoch 1250 | Loss: 0.0155
Epoch 1300 | Loss: 0.0154
Epoch 1350 | Loss: 0.0153
Epoch 1400 | Loss: 0.0152
Epoch 1450 | Loss: 0.0150
Epsilon: 0.600 ok
Epsilon: 0.040
Epoch 0 | Loss: 0.8141
Epoch 50 | Loss: 0.0417
Epoch 100 | Loss: 0.0314
Epoch 150 | Loss: 0.0275
Epoch 200 | Loss: 0.0252
Epoch 250 | Loss: 0.0237
Epoch 300 | Loss: 0.0226
Epoch 350 | Loss: 0.0218
Epoch 400 | Loss: 0.0211
Epoch 450 | Loss: 0.0205
Epoch 500 | Loss: 0.0201
Epoch 550 | Loss: 0.0196
Epoch 600 | Loss: 0.0193
Epoch 650 | Loss: 0.0190
Epoch 700 | Loss: 0.0187
Epoch 750 | Loss: 0.0184
Epoch 800 | Loss: 0.0182
Epoch 850 | Loss: 0.0180
Epoch 900 | Loss: 0.0178
Epoch 950 | Loss: 0.0176
Epoch 1000 | Loss: 0.0174
Epoch 1050 | Loss: 0.0173
Epoch 1100 | Loss: 0.0172
Epoch 1150 | Loss: 0.0170
Epoch 1200 | Loss: 0.0169
Epoch 1250 | Loss: 0.0168
Epoch 1300 | Loss: 0.0167
Epoch 1350 | Loss: 0.0166
```

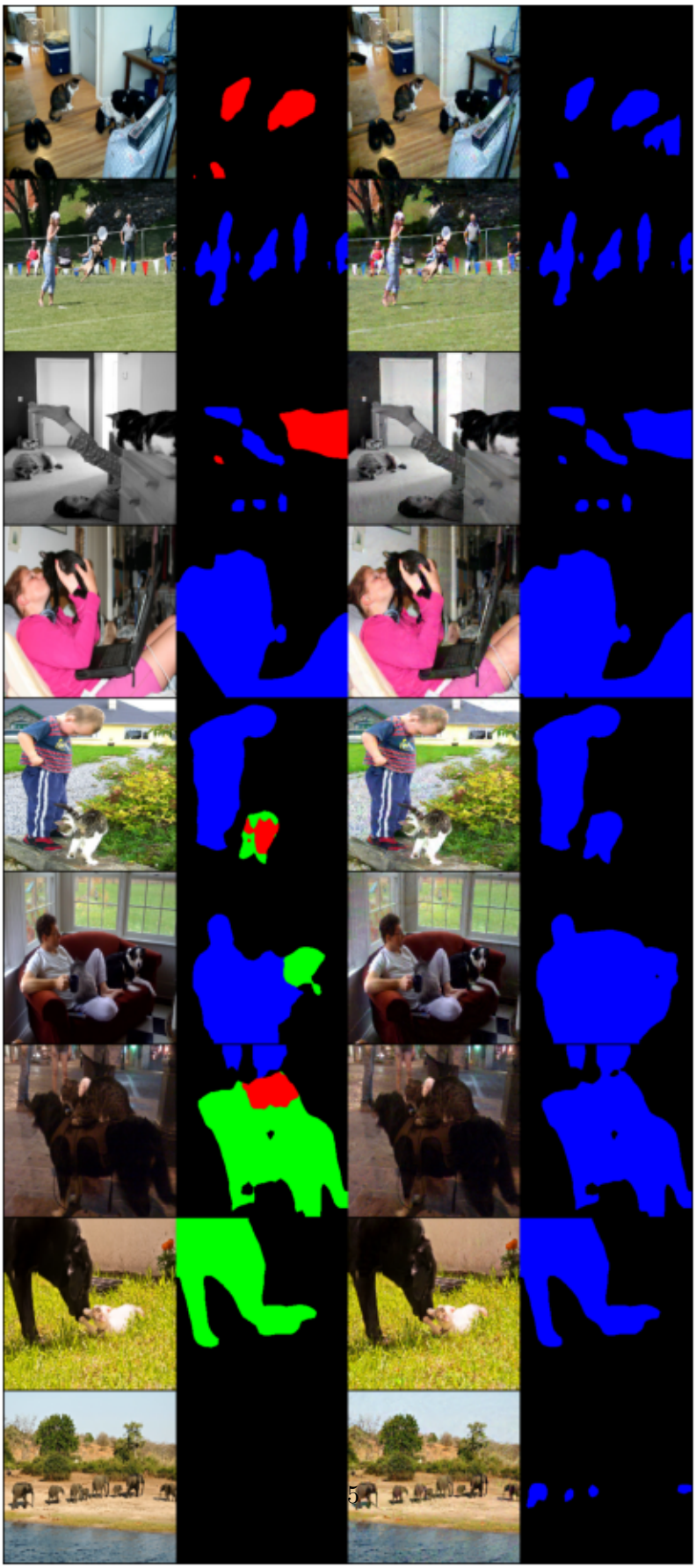
```
Epoch 1400 | Loss: 0.0165
Epoch 1450 | Loss: 0.0164
Epsilon: 0.040 ok
```

[8]:
```python
# Visualisation

# Initialisation des paramètres
W = torchvision.models.segmentation.DeepLabV3_ResNet50_Weights.
 ↪COCO_WITH_VOC_LABELS_V1
net_vis = torchvision.models.segmentation.deeplabv3_resnet50(weights=W).eval().
 ↪cpu()
img = load_data("coco_sample.pth")
normalize = transforms.Normalize([0.485, 0.456, 0.406], [0.229, 0.224, 0.225])


for r in results:
  print("epsilon = "+str(r[0]))
  visualize_attack_results(
      attack=r[1],
      model=net_vis,
      original_image=img,
      normalize_transform=normalize,
      fig_size=(20, 12),
      title="Comparaison avant/après attaque\n[Original | Segmentation |␣
 ↪Perturbé | Nouvelle Segmentation]",
      device=torch.device('cpu')
  )
```

```
<ipython-input-2-08d6d31b05da>:27: FutureWarning: You are using `torch.load`
with `weights_only=False` (the current default value), which uses the default
pickle module implicitly. It is possible to construct malicious pickle data
which will execute arbitrary code during unpickling (See
https://github.com/pytorch/pytorch/blob/main/SECURITY.md#untrusted-models for
more details). In a future release, the default value for `weights_only` will be
flipped to `True`. This limits the functions that could be executed during
unpickling. Arbitrary objects will no longer be allowed to be loaded via this
mode unless they are explicitly allowlisted by the user via
`torch.serialization.add_safe_globals`. We recommend you start setting
`weights_only=True` for any use case where you don't have full control of the
loaded file. Please open an issue on GitHub for any issues related to this
experimental feature.
  data = torch.load(path, map_location=device)
```

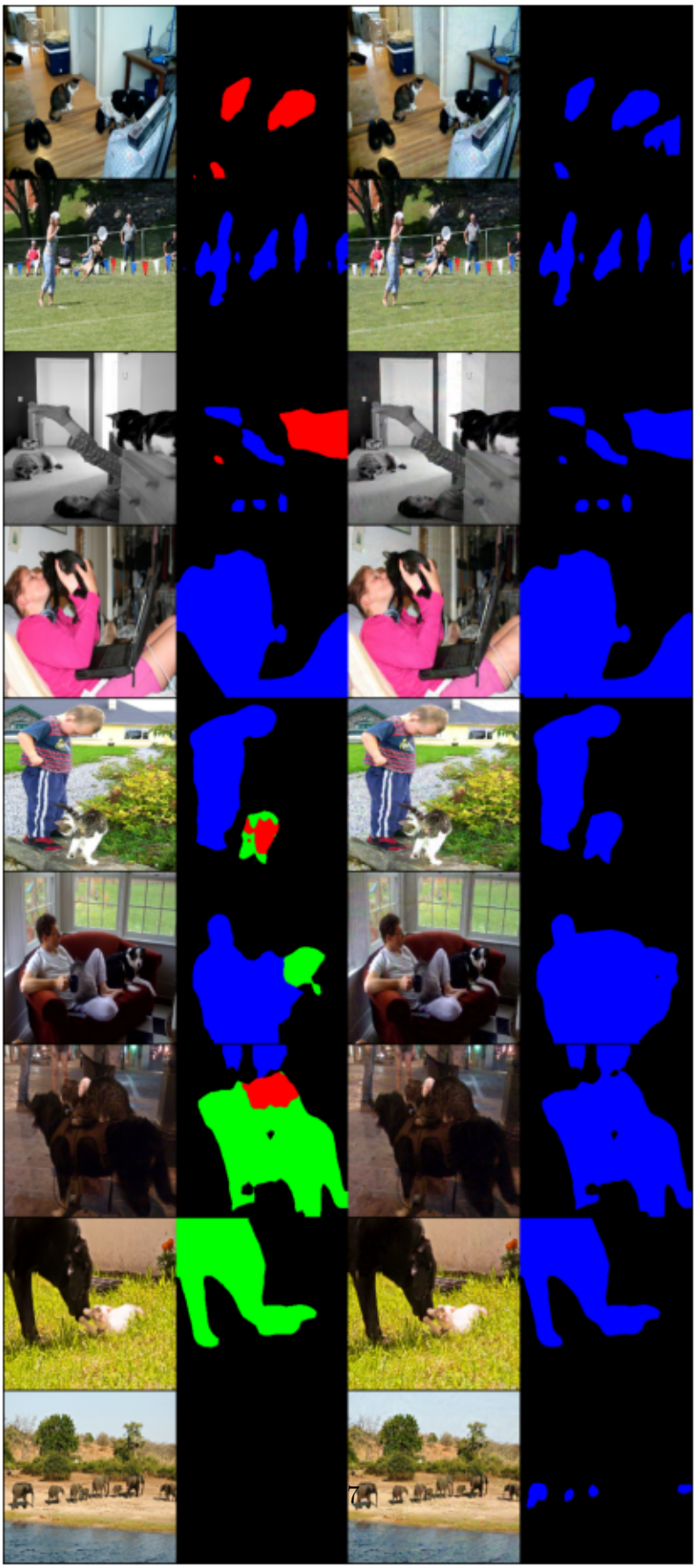
```
epsilon = 0.6
```

Comparaison avant/après attaque
[Original | Segmentation | Perturbé | Nouvelle Segmentation]



```
epsilon = 0.04
```

Comparaison avant/après attaque
[Original | Segmentation | Perturbé | Nouvelle Segmentation]



Nous obtenons quasiment les memes résultats:

- pour epsilon = 0.6, le bruit est légèrement plus visible et la loss plus faible (0.0150)
- Pour epsilon = 0.04, le bruit est moins visible et la loss plus grande (0.0164)

Dans les deux cas, notre attaque est très performante et transforme 100% de tous les objets détectés en personne(couleur bleu).

1.4.9 Question 4 : Transferabilité entre Modèles

Objectif : Tester l'attaque sur un modèle non entraîné (DeepLabV3-MobileNet).

Résultat :

- Taux de succès quasi identique

Explication: L'approche de notre attaque étant basée sur les features, elle est par nature plus transférable qu'une attaque plus classique.

Code : #### Application de l'attaque à MobileNet

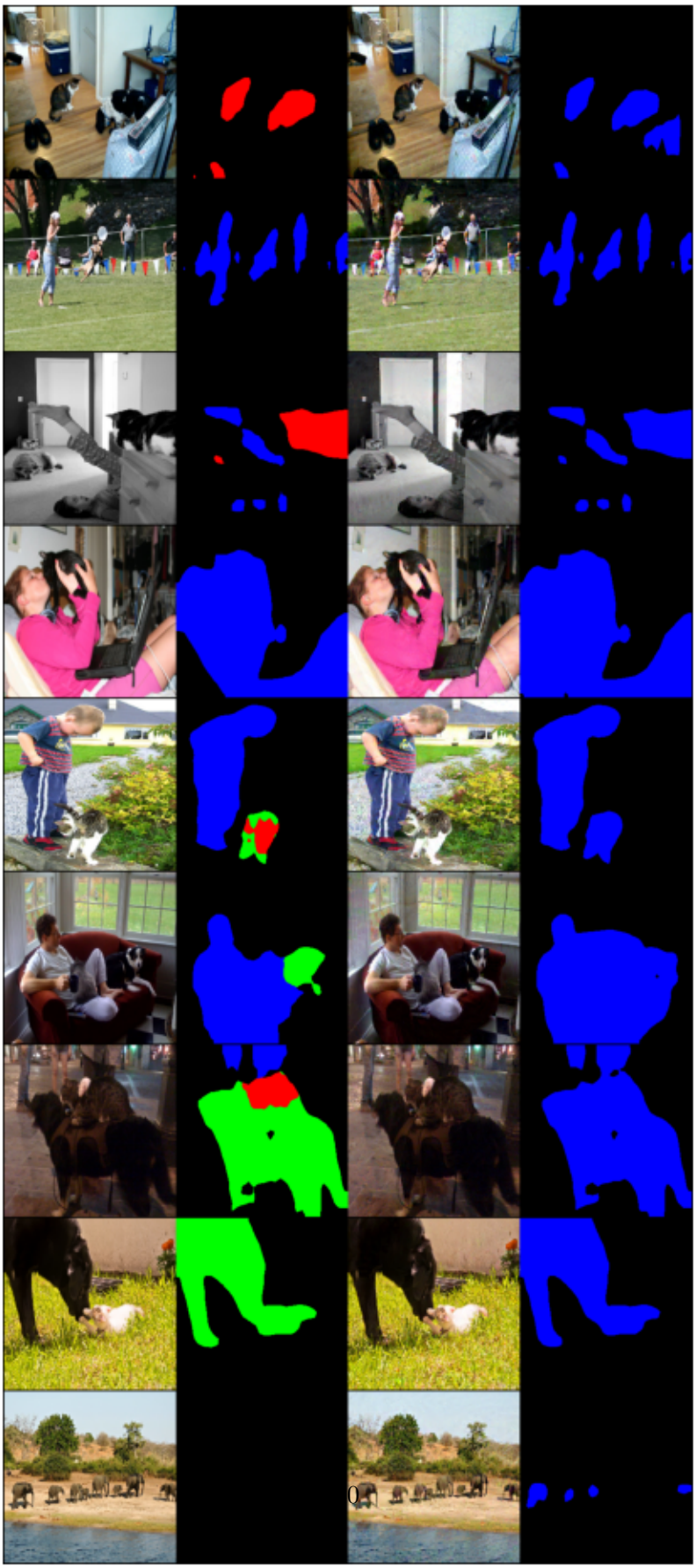
```
net_mobilenet = torchvision.models.segmentation.deeplabv3_mobilenet_v3_large().eval()
visualize_attack_results(attack, model=net_mobilenet, ...)
```

[9]:
```
# Initialisation des paramètres
W = torchvision.models.segmentation.DeepLabV3_MobileNet_V3_Large_Weights.
 ↪COCO_WITH_VOC_LABELS_V1
net = torchvision.models.segmentation.deeplabv3_mobilenet_v3_large(weights=W).
 ↪cpu().eval()
img = load_data("coco_sample.pth")
normalize = transforms.Normalize([0.485, 0.456, 0.406], [0.229, 0.224, 0.225])


for r in results:
  # print("epsilon = "+str(r[0]))
  visualize_attack_results(
      attack=r[1],
      model=net_vis,
      original_image=img,
      normalize_transform=normalize,
      fig_size=(20, 12),
      title="Comparaison avant/après attaque\n[Original | Segmentation |␣
 ↪Perturbé | Nouvelle Segmentation]",
      device=torch.device('cpu')
  )
```

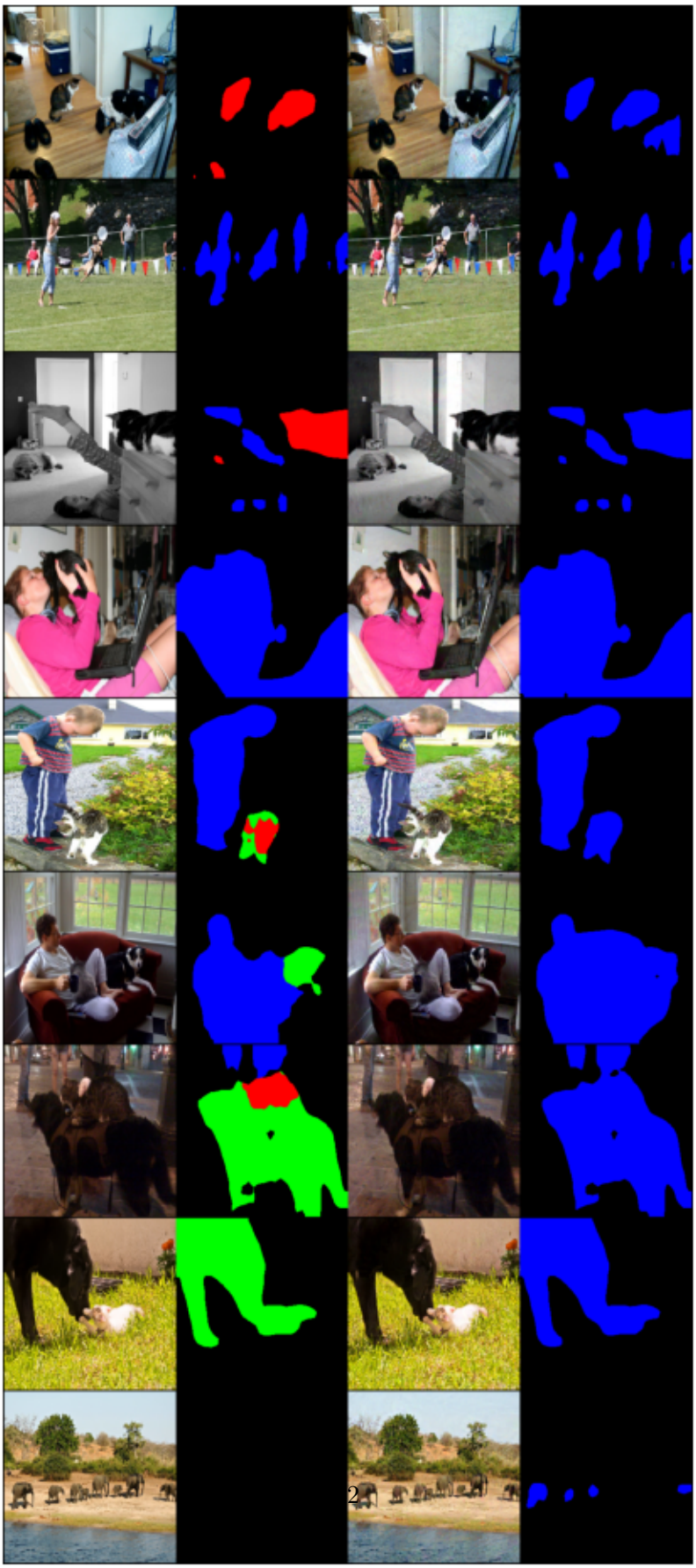
```
Downloading: "https://download.pytorch.org/models/deeplabv3_mobilenet_v3_large-
fc3c493d.pth" to
/root/.cache/torch/hub/checkpoints/deeplabv3_mobilenet_v3_large-fc3c493d.pth
```

```
100%|      | 42.3M/42.3M [00:00<00:00, 96.3MB/s]
<ipython-input-2-08d6d31b05da>:27: FutureWarning: You are using `torch.load`
with `weights_only=False` (the current default value), which uses the default
pickle module implicitly. It is possible to construct malicious pickle data
which will execute arbitrary code during unpickling (See
https://github.com/pytorch/pytorch/blob/main/SECURITY.md#untrusted-models for
more details). In a future release, the default value for `weights_only` will be
flipped to `True`. This limits the functions that could be executed during
unpickling. Arbitrary objects will no longer be allowed to be loaded via this
mode unless they are explicitly allowlisted by the user via
`torch.serialization.add_safe_globals`. We recommend you start setting
`weights_only=True` for any use case where you don't have full control of the
loaded file. Please open an issue on GitHub for any issues related to this
experimental feature.
  data = torch.load(path, map_location=device)
```

Comparaison avant/après attaque
[Original | Segmentation | Perturbé | Nouvelle Segmentation]



Comparaison avant/après attaque
[Original | Segmentation | Perturbé | Nouvelle Segmentation]



1.4.10 Question 5 : Attaque Multi-Modèles

Objectif : Générer une perturbation transférable via un entraînement sur deux modèles. **Méthode** :

- Perte totale : Somme des pertes pour DeepLabV3-ResNet50 et FCN-ResNet50.

Avantage : La perturbation cible des features communes, améliorant la robustesse.

Résultat :

- Meilleure transférabilité sur MobileNet.
- Visualisation cohérente sur les deux modèles cibles.

[ ]:
```
import torch
import torchvision
from torch.utils.checkpoint import checkpoint
from PIL import Image
import torch.nn as nn
import torchvision.transforms as transforms
import matplotlib.pyplot as plt

# Configuration
class Config:
    resolution = 200
    lr = 0.001
    tv_weight = 0.002
    epochs = 1500
config = Config()
device = torch.device("cuda" if torch.cuda.is_available() else "cpu")

# Feature Extractor
class FixedFeatureExtractor(nn.Module):
    def __init__(self):
        super().__init__()
        base = torchvision.models.segmentation.
 ↪deeplabv3_resnet50(weights="DEFAULT").backbone
        self.stem = nn.Sequential(base.conv1, base.bn1, base.relu, base.maxpool)
        self.layer1 = base.layer1
        self.layer2 = base.layer2
        self.layer3 = base.layer3
        self.layer4 = base.layer4
        self.downsample_factor = 8

    def forward(self, x):
        x = self.stem(x)
```

```
        x = self.layer1(x)
        x = self.layer2(x)
        x = self.layer3(x)
        x = self.layer4(x)
        return x

# Normalisation
normalize = transforms.Normalize([0.485, 0.456, 0.406], [0.229, 0.224, 0.225])

# Chargement des données
def load_data(path, size=200):
    transform = transforms.Compose([
        transforms.Resize((size, size)),
        transforms.ToTensor(),
    ])
    if path.endswith('.pth'):
        data = torch.load(path, map_location=device)
        return torch.nn.functional.interpolate(data, size=(size, size),␣
 ↪mode='bilinear')
    else:
        img = transform(Image.open(path).convert('RGB')).unsqueeze(0)
        return img.to(device)

person_img = load_data("cible4.jpg")
img = load_data("coco_sample.pth")

# Initialisation des deux modèles
model_names = ["deeplabv3_resnet50", "fcn_resnet50"]
feature_extractors = []
seg_models = []

for model_name in model_names:
    if model_name == "deeplabv3_resnet50":
        feature_extractor = FixedFeatureExtractor().to(device).eval()
        seg_model = torchvision.models.segmentation.
 ↪deeplabv3_resnet50(weights="DEFAULT").to(device).eval()
    elif model_name == "fcn_resnet50":
        feature_extractor = FixedFeatureExtractor().to(device).eval()
        seg_model = torchvision.models.segmentation.
 ↪fcn_resnet50(weights="DEFAULT").to(device).eval()
    feature_extractors.append(feature_extractor)
    seg_models.append(seg_model)

# Calcul des caractéristiques cibles pour chaque modèle
target_features_list = []
with torch.no_grad():
    for net in feature_extractors:
```

```
        f_target = net(normalize(person_img))
        target_features_list.append(f_target)

# Initialisation de l'attaque
attaque = nn.Parameter(torch.randn_like(img) * 0.1)  # Initialisation avec bruit
optimizer = torch.optim.Adam([attaque], lr=config.lr)

# Boucle d'entraînement
for epoch in range(config.epochs):
    optimizer.zero_grad()
    x_perturbed = torch.clamp(img + attaque, 0, 1)

    total_loss = 0
    for i, net in enumerate(feature_extractors):
        features = net(normalize(x_perturbed))

        # Génération du masque cible (tous les objets sauf fond)
        with torch.no_grad():
            orig_seg = seg_models[i](normalize(img))['out']
            object_mask = (orig_seg.argmax(1) != 0).float()  # Cible tous les␣
 ↪objets
            object_mask = torch.nn.functional.interpolate(
                object_mask.unsqueeze(1),
                size=features.shape[-2:],
                mode='nearest'
            )

        # Calcul des pertes
        loss_content = nn.MSELoss()(features * object_mask,␣
 ↪target_features_list[i] * object_mask)
        tv_loss = torch.sum(torch.abs(attaque[:,:,:,:-1] - attaque[:,:,:,1:]))
        loss = loss_content + config.tv_weight * tv_loss

        total_loss += loss

    total_loss.backward()
    optimizer.step()
    if(epoch+1)%10==0:
      print(f"Epoch {epoch+1} | Loss: {total_loss.item():.4f}")
```

```
<ipython-input-8-57a68222913e>:48: FutureWarning: You are using `torch.load`
with `weights_only=False` (the current default value), which uses the default
pickle module implicitly. It is possible to construct malicious pickle data
which will execute arbitrary code during unpickling (See
https://github.com/pytorch/pytorch/blob/main/SECURITY.md#untrusted-models for
more details). In a future release, the default value for `weights_only` will be
flipped to `True`. This limits the functions that could be executed during
```

```
unpickling. Arbitrary objects will no longer be allowed to be loaded via this
mode unless they are explicitly allowlisted by the user via
`torch.serialization.add_safe_globals`. We recommend you start setting
`weights_only=True` for any use case where you don't have full control of the
loaded file. Please open an issue on GitHub for any issues related to this
experimental feature.
  data = torch.load(path, map_location=device)
Downloading:
"https://download.pytorch.org/models/fcn_resnet50_coco-1167a1af.pth" to
/root/.cache/torch/hub/checkpoints/fcn_resnet50_coco-1167a1af.pth
100%|      | 135M/135M [00:00<00:00, 144MB/s]

Epoch 10 | Loss: 435.8216
Epoch 20 | Loss: 385.1381
Epoch 30 | Loss: 339.0403
Epoch 40 | Loss: 297.3867
Epoch 50 | Loss: 259.9897
Epoch 60 | Loss: 226.5983
Epoch 70 | Loss: 196.9587
Epoch 80 | Loss: 170.8123
Epoch 90 | Loss: 147.8639
Epoch 100 | Loss: 127.8354
Epoch 110 | Loss: 110.4244
Epoch 120 | Loss: 95.3580
Epoch 130 | Loss: 82.3597
Epoch 140 | Loss: 71.1870
Epoch 150 | Loss: 61.6152
Epoch 160 | Loss: 53.4217
Epoch 170 | Loss: 46.4076
Epoch 180 | Loss: 40.4048
Epoch 190 | Loss: 35.2777
Epoch 200 | Loss: 30.8995
Epoch 210 | Loss: 27.1661
Epoch 220 | Loss: 23.9719
Epoch 230 | Loss: 21.2378
Epoch 240 | Loss: 18.8920
Epoch 250 | Loss: 16.8751
Epoch 260 | Loss: 15.1348
Epoch 270 | Loss: 13.6301
Epoch 280 | Loss: 12.3286
Epoch 290 | Loss: 11.1932
Epoch 300 | Loss: 10.2012
Epoch 310 | Loss: 9.3307
Epoch 320 | Loss: 8.5666
Epoch 330 | Loss: 7.8933
Epoch 340 | Loss: 7.2965
Epoch 350 | Loss: 6.7688
Epoch 360 | Loss: 6.2971
```

```
Epoch 370 | Loss: 5.8769
Epoch 380 | Loss: 5.4975
Epoch 390 | Loss: 5.1572
Epoch 400 | Loss: 4.8470
Epoch 410 | Loss: 4.5659
Epoch 420 | Loss: 4.3119
Epoch 430 | Loss: 4.0815
Epoch 440 | Loss: 3.8710
Epoch 450 | Loss: 3.6793
Epoch 460 | Loss: 3.5044
Epoch 470 | Loss: 3.3436
Epoch 480 | Loss: 3.1949
Epoch 490 | Loss: 3.0598
Epoch 500 | Loss: 2.9358
Epoch 510 | Loss: 2.8210
Epoch 520 | Loss: 2.7147
Epoch 530 | Loss: 2.6167
Epoch 540 | Loss: 2.5272
Epoch 550 | Loss: 2.4435
Epoch 560 | Loss: 2.3657
Epoch 570 | Loss: 2.2929
Epoch 580 | Loss: 2.2252
Epoch 590 | Loss: 2.1631
Epoch 600 | Loss: 2.1043
Epoch 610 | Loss: 2.0491
Epoch 620 | Loss: 1.9992
Epoch 630 | Loss: 1.9518
Epoch 640 | Loss: 1.9053
Epoch 650 | Loss: 1.8634
Epoch 660 | Loss: 1.8243
Epoch 670 | Loss: 1.7862
Epoch 680 | Loss: 1.7518
Epoch 690 | Loss: 1.7194
Epoch 700 | Loss: 1.6881
Epoch 710 | Loss: 1.6584
Epoch 720 | Loss: 1.6306
Epoch 730 | Loss: 1.6060
Epoch 740 | Loss: 1.5814
Epoch 750 | Loss: 1.5568
Epoch 760 | Loss: 1.5340
Epoch 770 | Loss: 1.5138
Epoch 780 | Loss: 1.4948
Epoch 790 | Loss: 1.4746
Epoch 800 | Loss: 1.4568
Epoch 810 | Loss: 1.4395
Epoch 820 | Loss: 1.4241
Epoch 830 | Loss: 1.4097
Epoch 840 | Loss: 1.3930
```

```
Epoch 850 | Loss: 1.3805
Epoch 860 | Loss: 1.3666
Epoch 870 | Loss: 1.3522
Epoch 880 | Loss: 1.3401
Epoch 890 | Loss: 1.3293
Epoch 900 | Loss: 1.3183
Epoch 910 | Loss: 1.3069
Epoch 920 | Loss: 1.2956
Epoch 930 | Loss: 1.2856
Epoch 940 | Loss: 1.2759
Epoch 950 | Loss: 1.2672
Epoch 960 | Loss: 1.2585
Epoch 970 | Loss: 1.2483
Epoch 980 | Loss: 1.2405
Epoch 990 | Loss: 1.2332
Epoch 1000 | Loss: 1.2270
Epoch 1010 | Loss: 1.2179
Epoch 1020 | Loss: 1.2120
Epoch 1030 | Loss: 1.2040
Epoch 1040 | Loss: 1.1972
Epoch 1050 | Loss: 1.1907
Epoch 1060 | Loss: 1.1844
Epoch 1070 | Loss: 1.1779
Epoch 1080 | Loss: 1.1713
Epoch 1090 | Loss: 1.1645
Epoch 1100 | Loss: 1.1603
Epoch 1110 | Loss: 1.1556
Epoch 1120 | Loss: 1.1501
Epoch 1130 | Loss: 1.1470
Epoch 1140 | Loss: 1.1422
Epoch 1150 | Loss: 1.1366
Epoch 1160 | Loss: 1.1301
Epoch 1170 | Loss: 1.1252
Epoch 1180 | Loss: 1.1220
Epoch 1190 | Loss: 1.1178
Epoch 1200 | Loss: 1.1137
Epoch 1210 | Loss: 1.1093
Epoch 1220 | Loss: 1.1058
Epoch 1230 | Loss: 1.1012
Epoch 1240 | Loss: 1.0968
Epoch 1250 | Loss: 1.0939
Epoch 1260 | Loss: 1.0911
Epoch 1270 | Loss: 1.0859
Epoch 1280 | Loss: 1.0830
Epoch 1290 | Loss: 1.0795
Epoch 1300 | Loss: 1.0775
Epoch 1310 | Loss: 1.0741
Epoch 1320 | Loss: 1.0708
```

```
Epoch 1330 | Loss: 1.0656
Epoch 1340 | Loss: 1.0645
Epoch 1350 | Loss: 1.0610
Epoch 1360 | Loss: 1.0589
Epoch 1370 | Loss: 1.0564
Epoch 1380 | Loss: 1.0521
Epoch 1390 | Loss: 1.0509
Epoch 1400 | Loss: 1.0482
Epoch 1410 | Loss: 1.0447
Epoch 1420 | Loss: 1.0427
Epoch 1430 | Loss: 1.0398
Epoch 1440 | Loss: 1.0381
Epoch 1450 | Loss: 1.0353
Epoch 1460 | Loss: 1.0335
Epoch 1470 | Loss: 1.0312
Epoch 1480 | Loss: 1.0287
Epoch 1490 | Loss: 1.0267
Epoch 1500 | Loss: 1.0247
```

[ ]:
```python
# Visualisation des résultats (à définir selon vos besoins)
W = torchvision.models.segmentation.DeepLabV3_ResNet50_Weights.
  ↪COCO_WITH_VOC_LABELS_V1
net_vis = torchvision.models.segmentation.deeplabv3_resnet50(weights=W).eval().
  ↪cpu()

img = load_data("coco_sample.pth")

visualize_attack_results(
    attack=attaque,
    model=net_vis,
    original_image=img,
    normalize_transform=normalize,
    fig_size=(20, 12),
    title="Comparaison avant/après attaque\n[Original | Segmentation | Perturbé␣
  ↪| Nouvelle Segmentation]",
      device=torch.device('cpu')
)
```

```
<ipython-input-8-57a68222913e>:48: FutureWarning: You are using `torch.load`
with `weights_only=False` (the current default value), which uses the default
pickle module implicitly. It is possible to construct malicious pickle data
which will execute arbitrary code during unpickling (See
https://github.com/pytorch/pytorch/blob/main/SECURITY.md#untrusted-models for
more details). In a future release, the default value for `weights_only` will be
flipped to `True`. This limits the functions that could be executed during
unpickling. Arbitrary objects will no longer be allowed to be loaded via this
mode unless they are explicitly allowlisted by the user via
```

```
`torch.serialization.add_safe_globals`. We recommend you start setting
`weights_only=True` for any use case where you don't have full control of the
loaded file. Please open an issue on GitHub for any issues related to this
experimental feature.
  data = torch.load(path, map_location=device)
```

Comparaison avant/après attaque
[Original | Segmentation | Perturbé | Nouvelle Segmentation]

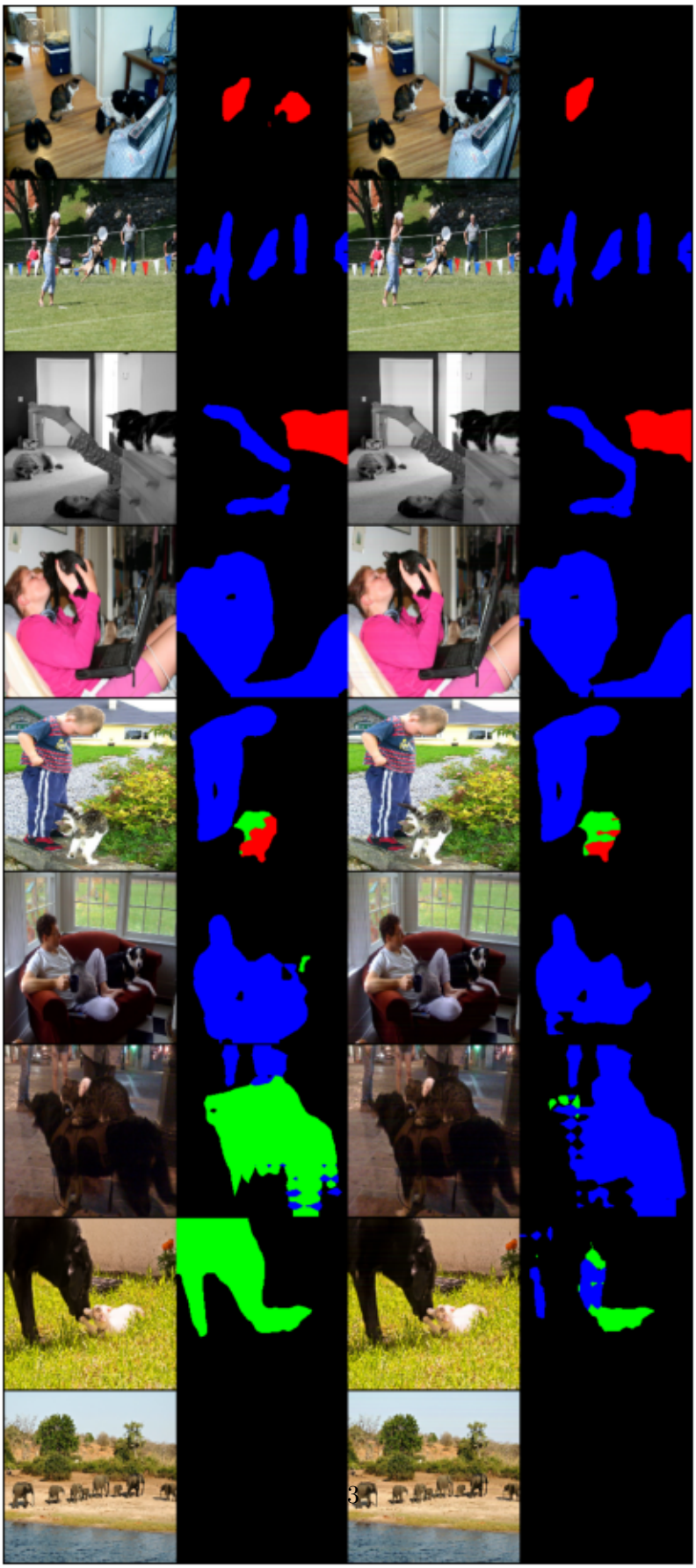
Notre attaque fournit d'assez bons résultats sur les sixième et septième images.

Etude de la transférabilité

[ ]:
```
# Visualisation des résultats (à définir selon vos besoins)
seg_model = torchvision.models.segmentation.fcn_resnet50(weights="DEFAULT").
 ↪cpu().eval()

img = load_data("coco_sample.pth")

visualize_attack_results(
    attack=attaque,
    model=seg_model,
    original_image=img,
    normalize_transform=normalize,
    fig_size=(20, 12),
    title="Comparaison avant/après attaque\n[Original | Segmentation | Perturbé␣
 ↪| Nouvelle Segmentation]",
      device=torch.device('cpu')
)
```

```
<ipython-input-8-57a68222913e>:48: FutureWarning: You are using `torch.load`
with `weights_only=False` (the current default value), which uses the default
pickle module implicitly. It is possible to construct malicious pickle data
which will execute arbitrary code during unpickling (See
https://github.com/pytorch/pytorch/blob/main/SECURITY.md#untrusted-models for
more details). In a future release, the default value for `weights_only` will be
flipped to `True`. This limits the functions that could be executed during
unpickling. Arbitrary objects will no longer be allowed to be loaded via this
mode unless they are explicitly allowlisted by the user via
`torch.serialization.add_safe_globals`. We recommend you start setting
`weights_only=True` for any use case where you don't have full control of the
loaded file. Please open an issue on GitHub for any issues related to this
experimental feature.
  data = torch.load(path, map_location=device)
```

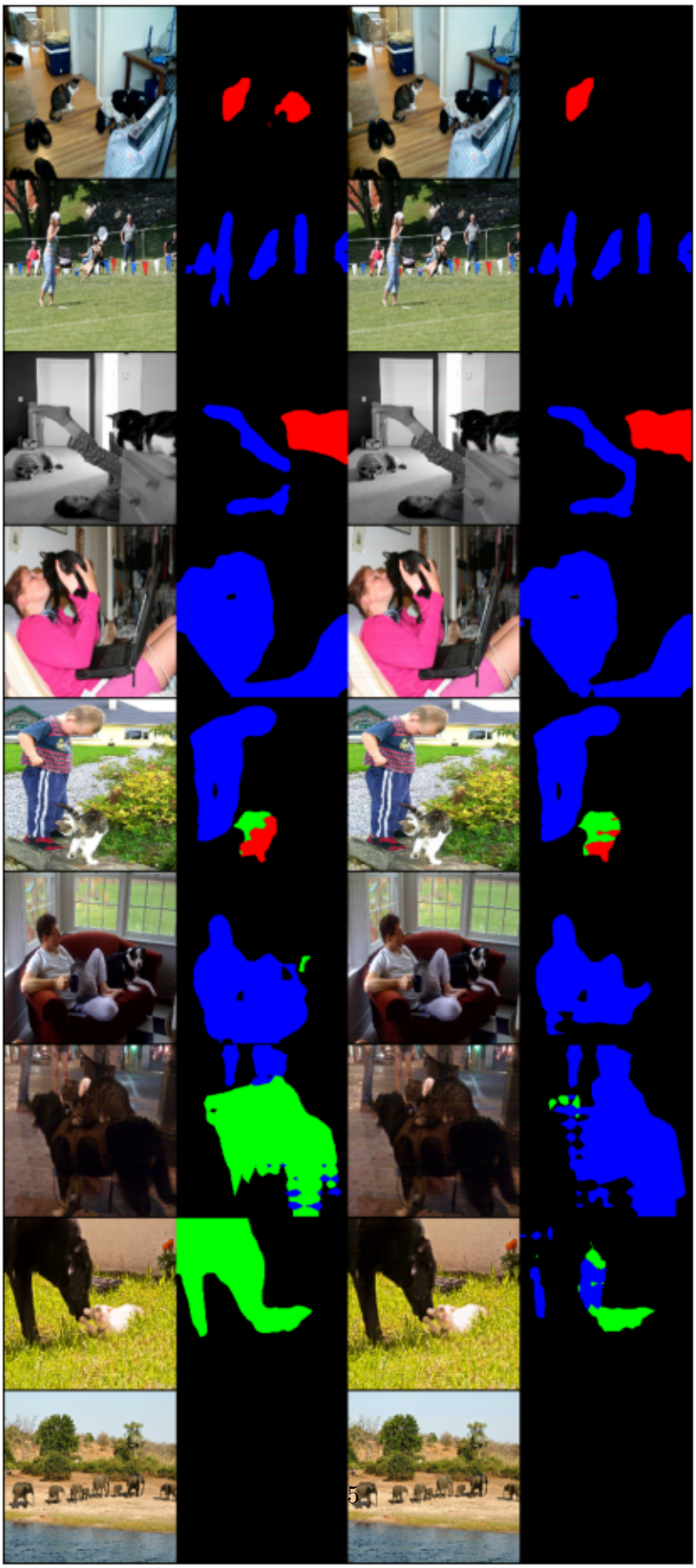
Comparaison avant/après attaque
[Original | Segmentation | Perturbé | Nouvelle Segmentation]



[ ]:
```
W = torchvision.models.segmentation.DeepLabV3_MobileNet_V3_Large_Weights.
 ↪COCO_WITH_VOC_LABELS_V1
net = torchvision.models.segmentation.deeplabv3_mobilenet_v3_large(weights=W).
 ↪cpu().eval()

img = load_data("coco_sample.pth")

visualize_attack_results(
    attack=attaque,
    model=seg_model,
    original_image=img,
    normalize_transform=normalize,
    fig_size=(20, 12),
    title="Comparaison avant/après attaque\n[Original | Segmentation | Perturbé␣
 ↪| Nouvelle Segmentation]",
      device=torch.device('cpu')
)
```

```
<ipython-input-8-57a68222913e>:48: FutureWarning: You are using `torch.load`
with `weights_only=False` (the current default value), which uses the default
pickle module implicitly. It is possible to construct malicious pickle data
which will execute arbitrary code during unpickling (See
https://github.com/pytorch/pytorch/blob/main/SECURITY.md#untrusted-models for
more details). In a future release, the default value for `weights_only` will be
flipped to `True`. This limits the functions that could be executed during
unpickling. Arbitrary objects will no longer be allowed to be loaded via this
mode unless they are explicitly allowlisted by the user via
`torch.serialization.add_safe_globals`. We recommend you start setting
`weights_only=True` for any use case where you don't have full control of the
loaded file. Please open an issue on GitHub for any issues related to this
experimental feature.
  data = torch.load(path, map_location=device)
```

Comparaison avant/après attaque
[Original | Segmentation | Perturbé | Nouvelle Segmentation]



Les résultats ci dessus montrent que notre attaque est transférable sur les modèles deeplabv3_resnet50, fcn_resnet50 et deeplabv3_mobilenet_v3_large. Les performances approximativement les memes.

1.5 Conclusion

TOut au long de ce projet, nous avons exploré les attaques adversariales appliquées à la segmentation d'images, en utilisant des modèles préentraînés et un jeu de données issu de MS-COCO. Nous avons réussi à générer des perturbations imperceptibles qui trompent le réseau, altérant ainsi ses prédictions.

Nos expérimentations ont permis de répondre aux objectifs fixés :

- Nous avons démontré la faisabilité d'une attaque non ciblée contre un modèle de segmentation.
- Nous avons généré des attaques ciblées conduisant à des prédictions spécifiques.
- Nous avons analysé l'impact de la norme de perturbation sur l'efficacité des attaques.
- Nous avons étudié la généralisation des attaques à d'autres architectures et leur apprentissage sur plusieurs réseaux.

Ces résultats soulignent la vulnérabilité des modèles de segmentation aux attaques adversariales et ouvrent la voie à des stratégies de défense et de robustification. Des travaux futurs pourraient inclure l'application de méthodes de défense comme l'entraînement adversarial ou la détection des perturbations.