# CS484 Final Project - Deep features for interactive segmentation

---

*Student Name:* Yash Harshal Barve

*Student ID:* 20930353

*Student Email:* yhbarve@uwaterloo.ca

---

## Motivation behind choosing this project

I found image segmentation for object detection really interesting. Additionally, working on A4 was really fun. Especially, the interactive GUI which did the computations and generated the segmentation based on the seeds that I chose was really cool. This project naturally appealed to me because it extends those same graph-cut ideas we explored in class and built in A4, now applied to deep feature representations.

---

## Theoretical background:

Deep convolutional features represent a significant advancement over traditional RGB color spaces for image segmentation tasks. While RGB values capture only low-level pixel intensities across three channels, deep features from pre-trained networks encode hierarchical visual concepts learned from millions of diverse images. This rich representational capacity allows segmentation algorithms to leverage semantic understanding rather than just color differences.

---

## Introduction to the project:

In this project, the aim is build an interactive segmentation tool that uses deep convolutional features in place of raw RGB values. We begin by loading a pretrained ResNet-50 backbone, stripping off its classification head, and bilinearly upsampling its feature maps to the original image size so that each pixel carries a high-level descriptor. We then implement a graph cut object (similar to CS484 A4) that instantiates a graph, create t-links & n-links, and performs max-flow. We fine-tune two hyperparameters (lambda & sigma). In the end, based on the hyperparameters, we create an interactive GUI that allows users to add additional foreground and background seeds and see the model run the max-flow algorithm based on it.

Additionally, we also implemented a basic RGB feature map and an interactive GUI (similar to deep feature map) for users to play with. We then compare the two models on a set of different types of images.

Overall, while the RGB baseline excels on brightly contrasting objects (e.g. a red rose on green foliage), the deep-feature version delivers more robust, semantically coherent cuts in cases of color ambiguity, lighting variation, or complex textures.

---

## Section 01: Install libraries

This project involves all the common libraries like `pytorch`, `torchvision`, `numpy`, `matplotlib`, and `skimage`.

Additionally, to perform the graph-cut, we will be using `PyMaxflow` (just like in CS484 A4).

This project also uses the ResNet-50 pretrained model which is part of `torchvision.models`.

I am also using the custom module that was used in CS484 A4, which will be imported in section 09. This is an external library that I have not implemented.

In [10]:
```
%matplotlib widget
import torch
import torchvision
from torchvision import models, transforms
import numpy as np
import matplotlib.pyplot as plt
from skimage import io, color
import maxflow
import torch.nn.functional as F
```

In [12]:
```
# Ensure that pymaxflow has the necessary graph classes
print("Graph classes:", [c for c in dir(maxflow) if "Graph" in c])
print("maxflow module path:", maxflow.__file__)
```

```
Graph classes: ['Graph', 'GraphFloat', 'GraphInt']
maxflow module path: /opt/anaconda3/lib/python3.12/site-packages/maxflow/__init__.py
```

---

# Section 02: Load and strip ResNet-50

In this section, we will first create an instance of the pre-trained (on ImageNet data) ResNet-50 model. We will then strip the last two layers (average pool and classification). We then preprocess the input image before passing it through the resnet-50 model. After the resnet layers, we bilinearly interpolate the feature map back to the orginal image dimensions, so each pixel has a deep feature vector.

*Note: The normalization values are taken from https://stackoverflow.com/questions/58151507/why-pytorch-officially-use-mean-0-485-0-456-0-406-and-std-0-229-0-224-0-2*

In [14]:
```
# Load & strip backbone

# instantiate a pre-trained resnet-50 model
_model   = models.resnet50(weights=models.ResNet50_Weights.IMAGENET1K_V1)
# we don't want the global average pool and classification layers, so we strip them off
# we wrap the rest of the layers into a sequential model
_encoder = torch.nn.Sequential(*list(_model.children()))[:-2])
# switching the model into evaluation mode
_encoder.eval()

# Preprocessing transform
# convert to PyTorch tensor and normalize
_preprocess = transforms.Compose([
    transforms.ToTensor(),
    transforms.Normalize(mean=[0.485,0.456,0.406], std =[0.229,0.224,0.225])
])

# Feature extraction helper
def get_deep_features(img: np.ndarray) -> torch.Tensor:
    H, W = img.shape[:2]
    # preprocess the image and add a batch dimension
    x = _preprocess(img).unsqueeze(0)
    with torch.no_grad():
        # run the image through the stripped down resnet-50 model
        feats = _encoder(x)
        # upsample back to original image size (HxW)
        feats = F.interpolate(feats, size=(H, W), mode='bilinear', align_corners=False)
```

```
    # remove the batch dimension
    return feats.squeeze(0).cpu()
```

# Notes on choosing ResNet-50 as the base model

ResNet-50 was selected as the feature extractor due to its optimal balance between representational power and computational efficiency. The residual architecture's skip connections help preserve gradient flow through deep networks, allowing effective learning of fine-grained features at multiple abstraction levels. [1]

My implementation extracts 2048-dimensional feature vectors from the final convolutional block (layer4) of ResNet-50, preserving the network's learned representations before information compression occurs in the global average pooling layer. These high-dimensional features capture complex visual patterns spanning textures, shapes, parts, and object-level semantics. Earlier layers include low-level textures and edges, while the fully connected layers lose spatial information. (Zeiler and Fergus, 2014) [2]

Sources:


1. https://en.innovatiana.com/post/discover-resnet-50#:~:text=%2D%20Ability%20to%20form%20deeper%20networks,deeper%20architectures%20than%20its%20predecessors
2. https://arxiv.org/abs/1311.2901


---

# Section 03: Graph Cut functions

In this section we define the two helper functions used in the GraphCut class. I chose to not explicitly define a class, but just the methods which will be called in a specific order similar to the `MyGraphCuts` class from A4.

`compute_n_links()` computes 4-neighbor “n-link” weights from deep-feature contrast.

`compute_t_links()` computes per-pixel “t-link” (source/sink) costs from feature-to-seed distances, enforcing hard FG/BG strokes.

`run_graphcut()` adds the n-links and t-links to the graph and performs min-cut.

In [16]:
```
# The following functions are very similar to the ones implemented in CS484 A4.

def compute_n_links(feats, lambda_reg=10, sigma=1.0, scale=100):
    f = feats.cpu().numpy()
    C, H, W = f.shape

    # horizontal
    # per-channel difference between each pixel and its immediate right neighbor
    dr = np.sum((f[:,:,1:] - f[:,:,:-1])**2, axis=0)
    wr = lambda_reg * np.exp(-dr / sigma**2)
    n_right = np.zeros((H, W), dtype=np.int32)
    n_right[:, :-1] = np.round(wr * scale)

    # vertical
    # per-channel difference between each pixel and its immediate neighbor below
    db = np.sum((f[:,1:,:] - f[:,:-1,:])**2, axis=0)
    wb = lambda_reg * np.exp(-db / sigma**2)
    n_below = np.zeros((H, W), dtype=np.int32)
    n_below[:-1, :] = np.round(wb * scale)

    return n_right, n_below

def compute_t_links(feats, scribble_mask, hard_cost=10**6, fg_label=1, bg_label=0):
    f = feats.cpu().numpy()
```

```
    C, H, W = f.shape
    flat = f.reshape(C, -1)
    mask = scribble_mask.flatten()

    # foreground and background seeds
    fg_idx = np.where(mask == fg_label)[0]
    bg_idx = np.where(mask == bg_label)[0]

    # compute centroids by calculating means
    mu_fg = flat[:, fg_idx].mean(axis=1)
    mu_bg = flat[:, bg_idx].mean(axis=1)

    # per pixel squared distances
    d_fg = np.sum((flat - mu_fg[:,None])**2, axis=0).reshape(H, W)
    d_bg = np.sum((flat - mu_bg[:,None])**2, axis=0).reshape(H, W)

    # costs
    t_source = d_bg.astype(np.int32)
    t_sink   = d_fg.astype(np.int32)

    # enforce hard seeds
    t_source[scribble_mask==fg_label] = int(hard_cost)
    t_sink  [scribble_mask==bg_label] = int(hard_cost)

    # lock seeds exactly
    t_sink  [scribble_mask==fg_label] = 0
    t_source[scribble_mask==bg_label] = 0

    return t_source, t_sink

def run_graphcut(n_right, n_below, t_source, t_sink):
    # create the graph, add n-links and t-links, solve the min-cut, and extract segmentation mask
    g       = maxflow.GraphFloat()
    H, W    = t_source.shape
    nodeids = g.add_grid_nodes((H, W))

    struct_x = np.array([[0,0,0],[0,0,1],[0,0,0]])
    struct_y = np.array([[0,0,0],[0,0,0],[0,1,0]])

    g.add_grid_edges(nodeids, n_right, structure=struct_x, symmetric=True)
    g.add_grid_edges(nodeids, n_below, structure=struct_y, symmetric=True)
    g.add_grid_tedges(nodeids, t_source, t_sink)
    g.maxflow()

    seg_bool = g.get_grid_segments(nodeids)
    return (~seg_bool).astype(np.uint8)
```

---

# Section 04: RGB baseline helpers

The next two functions mirror our deep-feature counterparts but operate on raw RGB: one builds contrast-weighted 4-neighbor “n-links” in color space, the other builds “t-links” from simple color centroids + hard seeds.

We’ll use these as the classic RGB graph-cut baseline to compare against our deep-feature segmentation.

In [18]:
```
def compute_n_links_rgb(img, lambda_reg=10, sigma=10.0, scale=100):
    # normalize into [0,1]
    I = img.astype(np.float32) / 255.0
```

```
    H, W, _ = I.shape

    # horizontal differences
    dr = np.sum((I[:, 1:, :] - I[:, :-1, :])**2, axis=2)
    wr = lambda_reg * np.exp(-dr / sigma**2)
    n_right = np.zeros((H, W), dtype=np.int32)
    n_right[:, :-1] = np.round(wr * scale)

    # vertical differences
    db = np.sum((I[1:, :, :] - I[:-1, :, :])**2, axis=2)
    wb = lambda_reg * np.exp(-db / sigma**2)
    n_below = np.zeros((H, W), dtype=np.int32)
    n_below[:-1, :] = np.round(wb * scale)

    return n_right, n_below


def compute_t_links_rgb(img, scribble_mask, hard_cost=10**6, fg_label=1, bg_label=0):
    I = img.astype(np.float32)/255.0
    H, W, _ = I.shape
    flat = I.reshape(-1,3)
    mask = scribble_mask.flatten()

    # means
    mu_fg = flat[mask==fg_label].mean(axis=0)
    mu_bg = flat[mask==bg_label].mean(axis=0)
    d_fg = np.sum((flat-mu_fg)**2,axis=1).reshape(H,W)
    d_bg = np.sum((flat-mu_bg)**2,axis=1).reshape(H,W)
    t_source = d_bg.astype(int)
    t_sink   = d_fg.astype(int)
    # hard seeds
    t_source[scribble_mask==fg_label] = hard_cost
    t_sink  [scribble_mask==bg_label] = hard_cost
    # lock seeds
    t_sink  [scribble_mask==fg_label] = 0
    t_source[scribble_mask==bg_label] = 0
    return t_source, t_sink
```

---

# Section 05: Prepare Test Image & Scribble

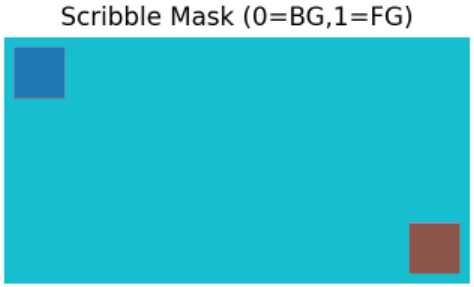
Before we use the interactive GUI to add manual seeds, we define a **fixed** scribble mask that we'll use for **all** test images during our λ/σ/scale grid search.
Using the same base seeds everywhere lets us compare parameter settings apples-to-apples—even though this "dummy" scribble won't give a perfect segmentation.

Once we've picked our best hyper-parameters, we'll switch to the GUI so the user can add or correct seeds to get a final, accurate result.

In [20]:
```
# load the image
img = io.imread('images/rose.bmp')
H, W = img.shape[:2]

# create a manual scribble
scribble_mask = np.full((H, W), 2, dtype=np.uint8)
scribble_mask[ 10: 60,  10: 60]       = 0   # background
scribble_mask[H-60:H-10, W-60:W-10]   = 1   # foreground

# visualize the scribble
plt.figure(figsize=(4,4))
plt.imshow(scribble_mask, cmap='tab10')
```

```
plt.title("Scribble Mask (0=BG,1=FG)"); plt.axis('off')
plt.show()
```

Figure 1




---

# Section 06: Data-Driven σ & λ Sweep for Deep Features

Here we calculate the value of $\sigma$ using the horizontal and vertical neighbor differences. We first extract per-pixel deep features and compute all squared -$\ell_2$ neighbor-differences (horizontal & vertical). Taking the median of those squared-differences gives a robust estimate for $\sigma^2$ and for $\sigma = \sqrt{median}$.

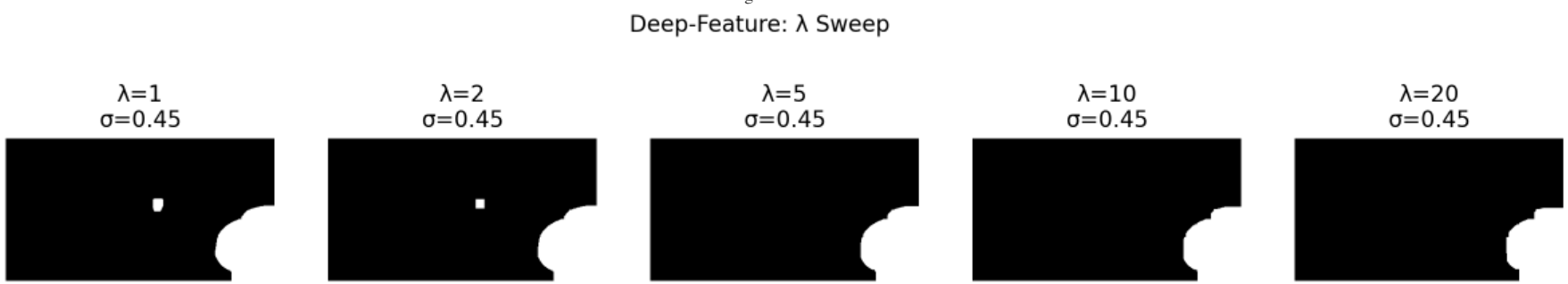
Then, holding $\sigma$ fixed, we sweep a small set of candidate regularization strengths $\lambda$ — which controls how strongly we penalize "jumps" between neighboring pixels. Next we run graph-cut for each, and visually pick the $\lambda$ that yields the best segmentation.

In [22]:
```
# extract deep features by passing the image through the resnet-50 model
feats = get_deep_features(img)

# compute data-driven σ_deep
f = feats.numpy()

# horizontal neighbor diff
dr = np.sum((f[:,:,1:] - f[:,:,:-1])**2, axis=0).ravel()

# vertical neighbor diff
db = np.sum((f[:,1:,:] - f[:,:-1,:])**2, axis=0).ravel()

median_diff = np.median(np.concatenate([dr, db]))
sigma_deep = np.sqrt(median_diff)
print(f"σ_deep = {sigma_deep:.4f} (median-diff² = {median_diff:.4f})")

# sweep λ_deep
scale_deep = 100
# list of candidates to try
lams = [1, 2, 5, 10, 20]
```

```
fig, axes = plt.subplots(1, len(lams), figsize=(15,3), sharey=True)
for ax, lam in zip(axes, lams):
    # create n-links
    n_r, n_b = compute_n_links(feats, lam, sigma_deep, scale_deep)
    # create t-links
    t_s, t_k = compute_t_links(feats, scribble_mask, hard_cost=H*W*lam)
    # create graph and run max-flow
    m = run_graphcut(n_r, n_b, t_s, t_k)
    ax.imshow(m, cmap='gray')
    ax.set_title(f"λ={lam}\no={sigma_deep:.2f}")
    ax.axis('off')
fig.suptitle("Deep-Feature: λ Sweep"); plt.show()
```

```
σ_deep = 0.4465 (median-diff² = 0.1994)
```

Figure 2




## Note

Since we're using a simple scribble, all 5 feature maps look very poor. However, we want to pick the smallest value of $\lambda$ that eliminates the leaking foreground into the background. In this case, when $\lambda = 1$ or $\lambda = 2$, we can see a part of the foreground (black) leak into the background (white). Therefore, we pick $\lambda = 5$, where this does not happen.

---

## Section 07: RGB-Baseline vs Deep Comparison

We calculate the $\sigma$ and $\lambda$ values for the RGB version using the same technique as above. We will then gnerate both the feature masks by creating the n-links, t-links, and running max-flow. We then compare the masks side by side.

In [24]:
```
# compute σ_rgb using similar technique as above
rgb = img.astype(np.float32)/255.0
dr_r = np.sum((rgb[:,:,1:] - rgb[:,:,:-1])**2, axis=2).ravel()
db_r = np.sum((rgb[:,1:,:] - rgb[:,:-1,:])**2, axis=2).ravel()
sigma_rgb = np.sqrt(np.median(np.concatenate([dr_r, db_r])))
print(f"σ_rgb = {sigma_rgb:.2f}")

# pick λ_rgb and same scale (here we are not sweeping)
lambda_rgb, scale_rgb = 10, 100

# compute both masks
```

```
n_r_rgb, n_b_rgb = compute_n_links_rgb(img, lambda_rgb, sigma_rgb, scale_rgb)
t_s_rgb, t_k_rgb = compute_t_links_rgb(img, scribble_mask, hard_cost=H*W*lambda_rgb)
mask_rgb = run_graphcut(n_r_rgb, n_b_rgb, t_s_rgb, t_k_rgb)

lambda_deep, scale_deep = 5, 100   # picked from the above λ sweep
n_r_d, n_b_d   = compute_n_links(feats, lambda_deep, sigma_deep, scale_deep)
t_s_d, t_k_d   = compute_t_links(feats, scribble_mask, hard_cost=H*W*lambda_deep)
mask_deep      = run_graphcut(n_r_d, n_b_d, t_s_d, t_k_d)

# visualize
plt.figure(figsize=(12,4))
plt.subplot(1,3,1); plt.imshow(img)
plt.title("Original")
plt.axis('off')

plt.subplot(1,3,2)
plt.imshow(mask_rgb, cmap='gray')
plt.title("RGB")
plt.axis('off')

plt.subplot(1,3,3)
plt.imshow(mask_deep, cmap='gray')
plt.title("Deep")
plt.axis('off')
plt.show()
```

```
σ_rgb = 0.08
```

Figure 3


# Note:

Again, this is not representative because of the simple scribble that we have used.

---

# Section 08: Multi-Image Deep Demo

Test the hyperparameter values on other images to ensure generalization.

The same scribble may work better for some images based on the position of the foreground object.

In [26]:
```
lambda_best, sigma_best, scale_best = lambda_deep, sigma_deep, scale_deep

for fname in ['bunny.bmp','hand.bmp','lama.jpg']:
    img2 = io.imread(f'images/'+fname)
    H2, W2 = img2.shape[:2]
    # create scribble (similar to the one created in previous blocks)
    scrib2 = np.full((H2,W2),2,np.uint8)
    scrib2[:H2//10,:W2//10] = 0
    scrib2[-H2//10:,-W2//10:] = 1

    # get deep features, generate n-links and t-links, and run max-flow
    feats2 = get_deep_features(img2)
    n2r,n2b = compute_n_links(feats2, lambda_best, sigma_best, scale_best)
    t2s,t2k = compute_t_links(feats2, scrib2, hard_cost=H2*W2*lambda_best)
    m2 = run_graphcut(n2r, n2b, t2s, t2k)

    plt.figure(figsize=(8,4))
    plt.subplot(1,2,1); plt.imshow(img2); plt.axis('off')
    plt.subplot(1,2,2); plt.imshow(m2, cmap='gray'); plt.axis('off')
    plt.suptitle(fname)
    plt.show()
```

Figure 4

bunny.bmp

Figure 5

hand.bmp

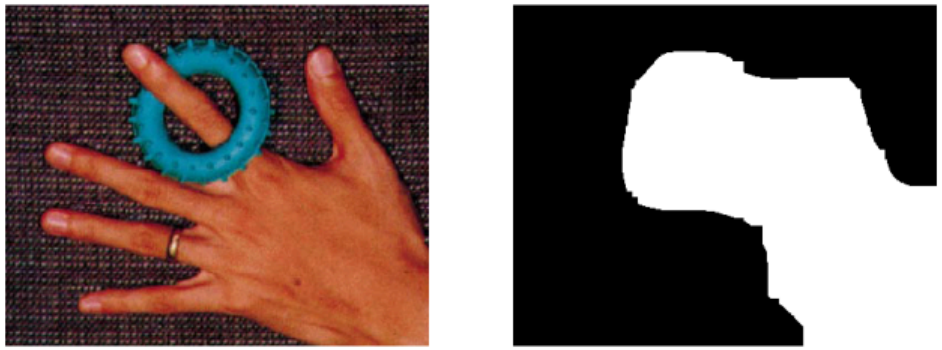


Figure 6

lama.jpg

---

## Section 09: Interactive GUI Session with 3 sample images

This is the interactive “wrapper” that ties everything together: you paint your own foreground/background seeds and it runs the deep-feature graph cut in real time. It really is the heart of the user experience for custom segmentation.

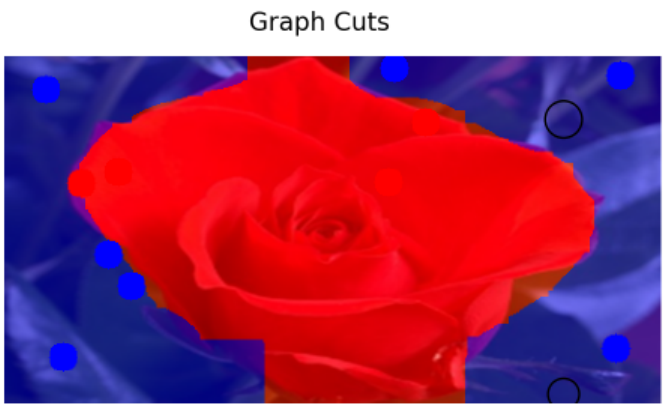
In [28]:
```
# using the custom modules from CS484 A4
from extlibs.asg1_error_handling import Figure, GraphCutsPresenter
```

```
# creating a class to for easier instantiation
class DeepFeatureGraphCuts:
    # setting labels for background, foreground, and none values in the seed mask
    bgr_value, obj_value, none_value = 0,1,2

    # set image, lambda, sigma, and scale
    def __init__(self, img, lambda_reg, sigma, scale=100):
        self.img, self.lambda_reg, self.sigma, self.scale = img, lambda_reg, sigma, scale
        self.fig = Figure()
        self.pres = GraphCutsPresenter(img, self)
        self.pres.connect_figure(self.fig)

    # extract deep features, generate n-links, t-links, and run max-flow
    def compute_labels(self, seed_mask):
        feats = get_deep_features(self.img)
        n_r, n_b = compute_n_links(feats, self.lambda_reg, self.sigma, self.scale)
        t_s, t_k = compute_t_links(feats, seed_mask, hard_cost=self.img.shape[0]*self.img.shape[1]*self.lambda_reg)
        m = run_graphcut(n_r,n_b,t_s,t_k)
        # re-lock seeds
        m[seed_mask==self.obj_value] = self.obj_value
        m[seed_mask==self.bgr_value] = self.bgr_value
        return m

    # run the interactive widget that allows users to add custom seeds
    def run(self):
        self.fig.show()
```

In [30]:
```
lambda_best, sigma_best, scale_best = lambda_deep, sigma_deep, scale_deep
```

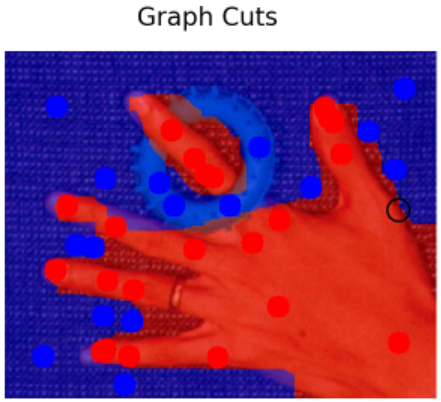
In [34]:
```
# Launch
img = io.imread('images/rose.bmp')
app = DeepFeatureGraphCuts(img, lambda_best, sigma_best, scale_best)
app.run()
# — left-click/drag = FG, right-click/drag = BG, Enter to apply
```

Figure 8



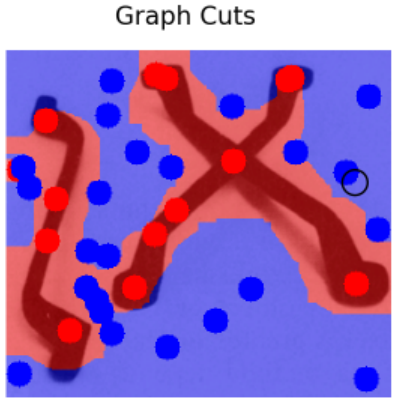

In [36]:
```
# Launch
img = io.imread('images/hand.bmp')
app = DeepFeatureGraphCuts(img, lambda_best, sigma_best, scale_best)
app.run()
# — left-click/drag = FG, right-click/drag = BG, Enter to apply
```

Figure 9



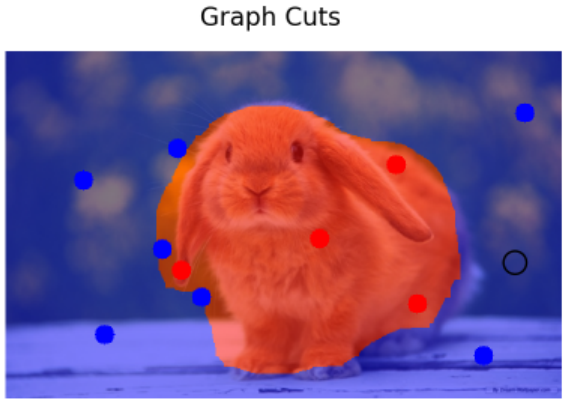

In [38]:
```
# Launch
img = io.imread('images/tools.bmp')
app = DeepFeatureGraphCuts(img, lambda_best, sigma_best, scale_best)
app.run()
# – left-click/drag = FG, right-click/drag = BG, Enter to apply
```

Figure 10




In [40]:
```
# Launch
img = io.imread('images/bunny.bmp')
app = DeepFeatureGraphCuts(img, lambda_best, sigma_best, scale_best)
app.run()
# – left-click/drag = FG, right-click/drag = BG, Enter to apply
```

Figure 11

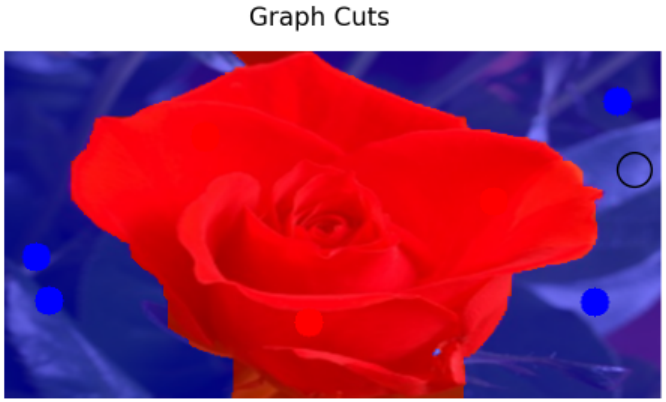



---

In [42]:
```python
class RGBGraphCuts:
    bgr_value, obj_value, none_value = 0, 1, 2

    def __init__(self, img, lambda_reg, sigma, scale=100):
        self.img = img
        self.lambda_reg = lambda_reg
        self.sigma = sigma
        self.scale = scale

        # set up the interactive figure + presenter
        self.fig = Figure()
        self.pres = GraphCutsPresenter(img, self)
        self.pres.connect_figure(self.fig)

    def compute_labels(self, seed_mask):
        # build RGB n-links and t-links
        n_r, n_b = compute_n_links_rgb(
            self.img, self.lambda_reg, self.sigma, self.scale
        )
        t_s, t_k = compute_t_links_rgb(
            self.img, seed_mask,
            hard_cost=self.img.shape[0] * self.img.shape[1] * self.lambda_reg
        )

        # run the graph-cut
        seg = run_graphcut(n_r, n_b, t_s, t_k)

        # re-lock the strokes exactly
        seg[seed_mask == self.obj_value] = self.obj_value
        seg[seed_mask == self.bgr_value] = self.bgr_value
        return seg

    def run(self):
        self.fig.show()
```

In [44]:
```python
img = io.imread('images/rose.bmp')
app_rgb = RGBGraphCuts(img, lambda_rgb, sigma_rgb, scale_rgb)
app_rgb.run()
```

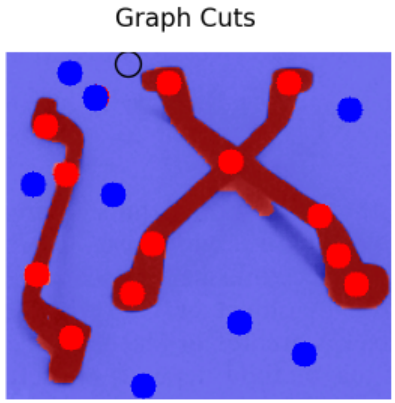
Figure 12




(x, y) = (378.9, 223.4) [2.]

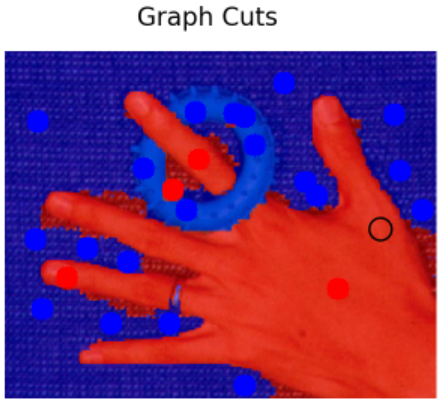
In [46]:
```
img = io.imread('images/tools.bmp')
app_rgb = RGBGraphCuts(img, lambda_rgb, sigma_rgb, scale_rgb)
app_rgb.run()
```

Figure 13




In [48]:
```
img = io.imread('images/hand.bmp')
app_rgb = RGBGraphCuts(img, lambda_rgb, sigma_rgb, scale_rgb)
app_rgb.run()
```

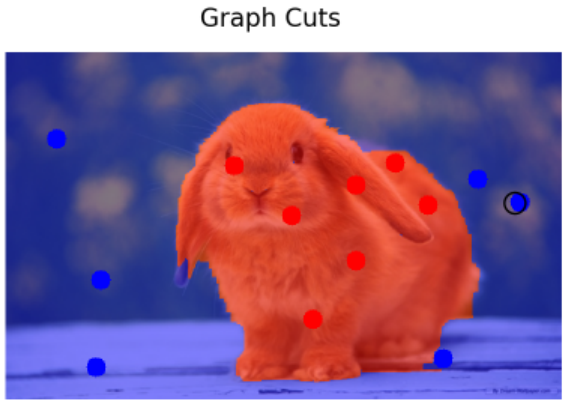
Figure 14




(x, y) = (379.5, 86.2) [2.]

In [50]:
```
lambda_rgb, sigma_rgb, scale_rgb = 10, sigma_rgb, 100
img = io.imread('images/bunny.bmp')
app_rgb = RGBGraphCuts(img, lambda_rgb, sigma_rgb, scale_rgb)
app_rgb.run()
```

Figure 15



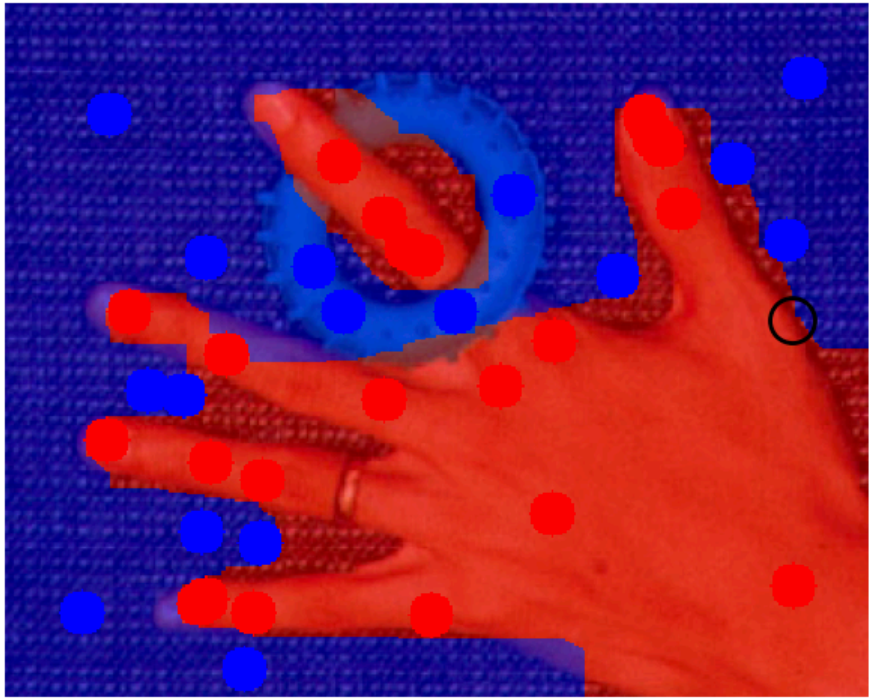

---

## Screenshots of the results

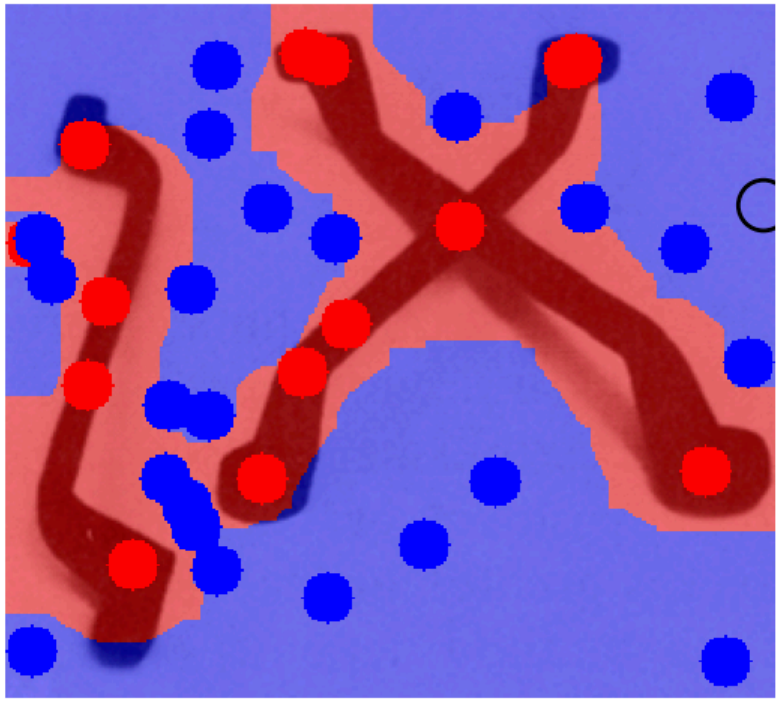
Since the matplotlib widget is not exported properly, I am attaching the screenshots of the results along with my observations and conclusion below.

## Deep Feature Map results

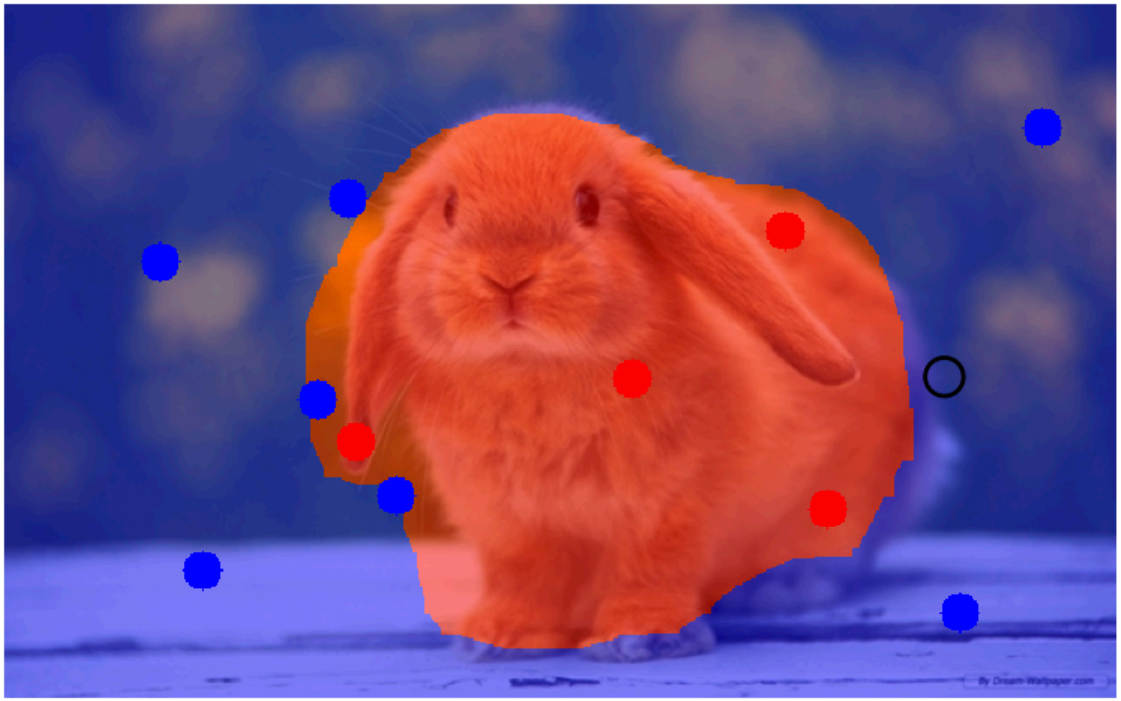
1. `rose.bmp`



2. `hand.bmp`



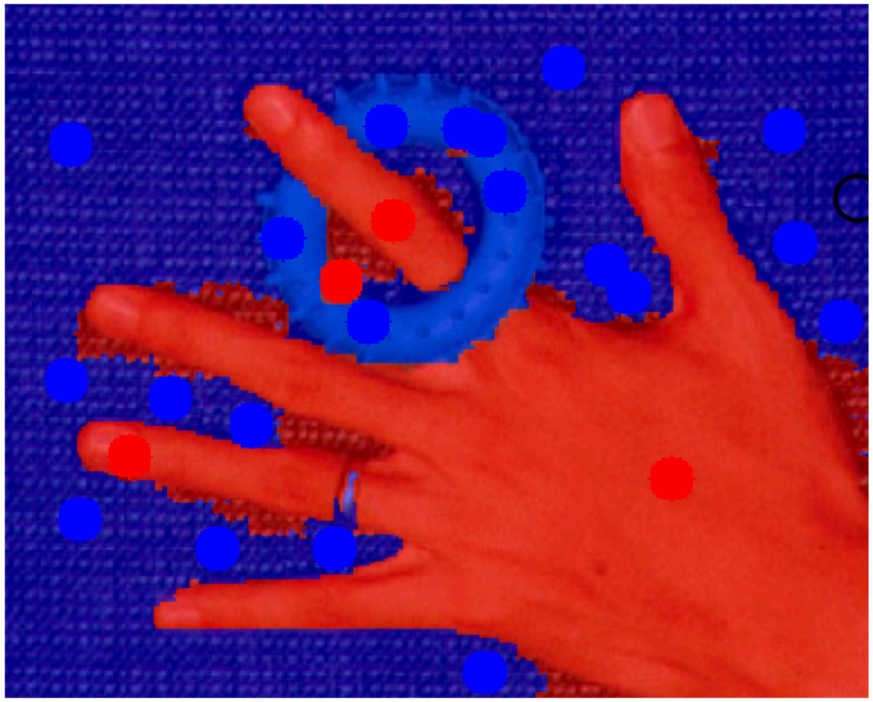
3. `model.bmp`



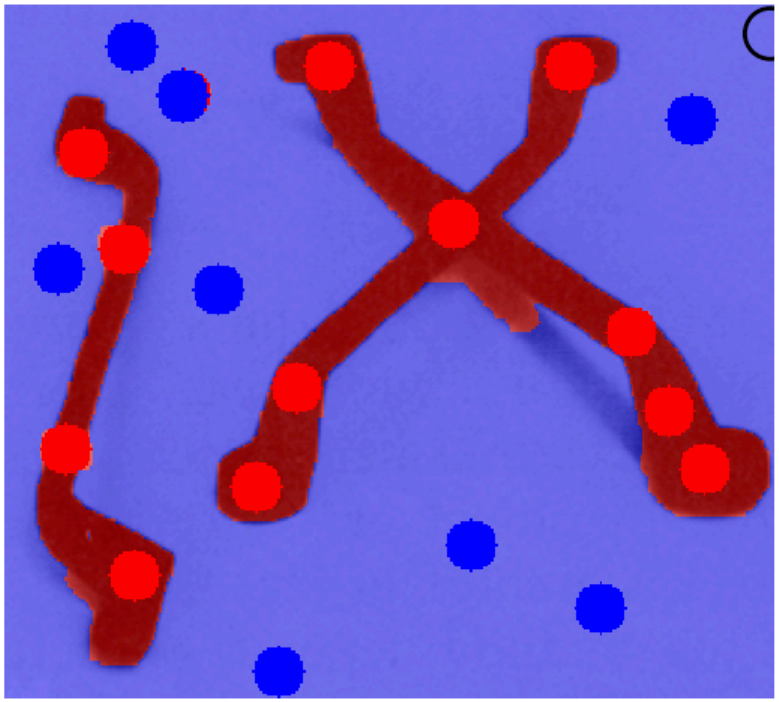
4. `bunny.bmp`

# RGB Feature Map Results

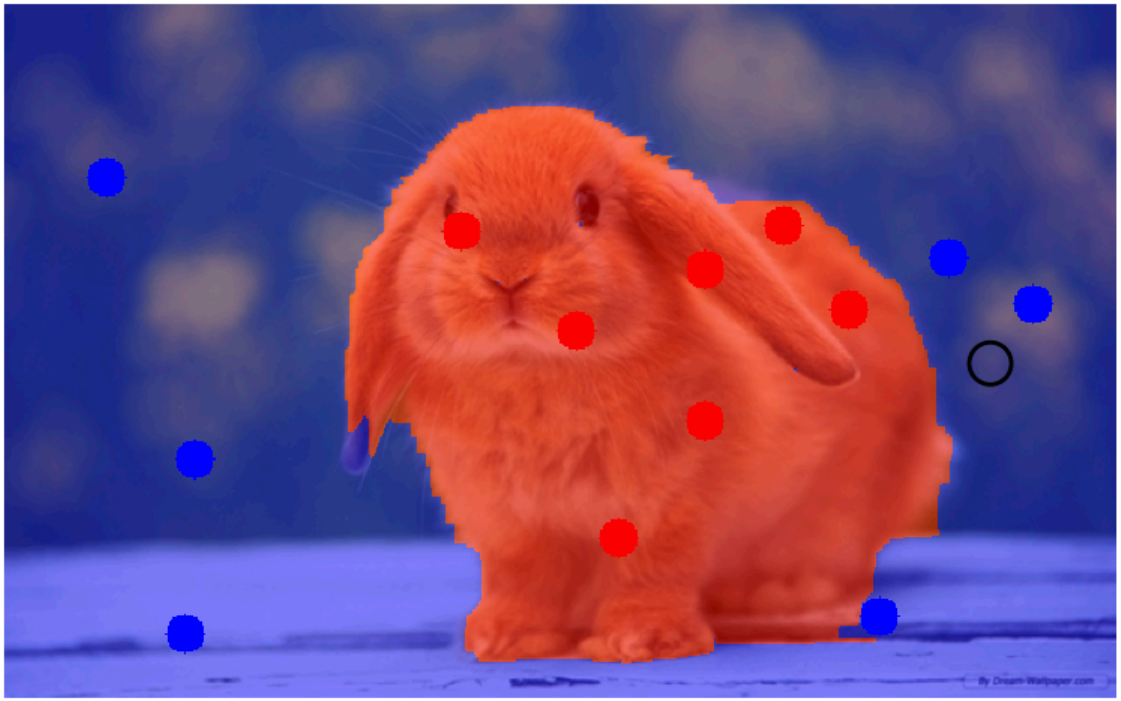
1. `rose.bmp`



2. `hand.bmp`



3. `model.bmp`



4. `bunny.bmp`

---

## Conclusion of results

- The RGB feature map, even though being a really simple model, works excellently in segmentation scenarious based purely on color. This can be seen in Example 1 (rose.bmp) where the subject is fairly straightforward. The deep feature map also does a good job, but takes a few more user seeds.
- The deep feature map seems to do slightly better in scenarios that involve object detection in a tricky background. In Example 3 (hand.bmp), the deep feature does a better job with lesser user seeds.
- **Note: I also tried to run `lama.bmp` for which the deep feature map is expected to do a better job because of the complicated background. However, my machine ran out of memory and crashed. (over multiple tries)**

---

## Possible explanation of the results

1. RGB Feature maps may actually do a better job in simple classification tasks because of the less complicated nature of the model.
2. We can improve our deep feature map using a better base model (currently using the ResNet-50). This can be done by either picking a different base model, stripping off more layers from the existing model, or adding more encoders / decoders.
3. The testing dataset is possibly very low (with only 4 images). This is primarily because of the memory-related constraints. The machine is not able to handle complex images. (such as `lama.bmp`) Testing on a wider set of images should represent the superiority of the deep feature map.
4. Currently, the optimal hyperparameter value for $\lambda$ needs to be chosen using visual comparison. Also, only 5 candidates are being considered. If this can be calculated mathematically and using a wider range of candidates, it could lead to more optimal choice of hyperparameters.

## Future Scope of Improvement:

1. Using a better base model or choosing the right layer to extract feature vectors.
2. Using a wider range of image dataset to test and compare the performance of both the models.
3. Using better quantitative evaluation metrics.
4. Choose hyperparameters mathematically instead of visual comparison.
5. A machine with better memory resources can facilitate longer training and testing (and with complex images).

---