Ramakrishna Mission Vivekananda Educational & Research Institute
Belur Math, Howrah, West Bengal
School of Mathematical Sciences, Department of Computer Science

Assignment - 2: Lightweight CNNs and Explainable AI
M.Sc. Computer Science and Big Data Analytics Date: 1st March 2026
Course: **CS411: Applications of Computer Vision and Deep Learning**
Deadline: 16th-March-2026, 11:59 P.M.
Instructor: Jimut Bahan Pal Max marks: 100

Instructions: Read Carefully and Attempt All Questions

Implementation Guidelines:

- All code must run end-to-end on **Google Colaboratory** (CPU runtime — no GPU required)
- Use the provided starter code for idea, don't assume they might be working correctly — verify and debug as needed.
- Your notebook must be self-contained: include all `!pip install` commands at the top
- Use **PyTorch** (`torch`, `torchvision`) throughout — do **not** use TensorFlow or Keras
- Use `matplotlib` for all visualisations; every plot must have a title and labelled axes
- Add brief inline comments explaining what each code block does
- You may discuss ideas with peers, but all code must be written individually

Submission Format:

- Submit your **Jupyter notebook** (`.ipynb`) with all cell outputs visible (do **not** clear outputs before submitting)
- Compress the file as **Name_ROLL.zip**

Submission Process:

- Email your submission to **jimutbahanpal@yahoo.com**
- Add **jpal.cs@gm.rkmvu.ac.in** as CC (Carbon Copy)
- Optionally CC your personal email to confirm delivery

Academic Integrity:
While you may use LLMs or agentic AI tools, you must:

- Fully understand any AI-generated code
- Be prepared to explain and justify every line during the viva
- Note: Inability to justify your work during the viva will result in negative marking

Deadline:
Submissions received after the deadline will incur a penalty of -5 **marks per day**. Plan accordingly and start early!

---

1. **Saliency Maps Using a Pre-trained PyTorch Model** (50)

Background. A *saliency map* highlights the input pixels that most influence a model's prediction. Given a model f and an input image $\mathbf{x}$, the vanilla-gradient saliency map is:

$$S(i,j) = \max_{c} \left| \frac{\partial f_k(\mathbf{x})}{\partial x_{i,j,c}} \right|$$

where k is the target class index and the maximum is taken over the three colour channels $c \in \{R, G, B\}$. In PyTorch, we enable gradient tracking on the *input tensor* (not the model parameters) and call `.backward()` to obtain these gradients.

Required packages (first cell of your notebook):

```
!pip install torch torchvision matplotlib Pillow requests
```

Starter imports:

```
import torch
import torch.nn.functional as F
import torchvision.models as models
import torchvision.transforms as transforms
from PIL import Image
import requests
from io import BytesIO
import numpy as np
import matplotlib.pyplot as plt
```

Tasks:

1. **Load the model and an image of your choice.** **[4]**
   Load **MobileNetV2** pre-trained on ImageNet:

   ```
   model = models.mobilenet_v2(
       weights=models.MobileNet_V2_Weights.IMAGENET1K_V1)
   model.eval()
   ```

   Download **any publicly accessible JPEG/PNG image** using `requests`, open it with `PIL.Image`, and display it with `matplotlib`. Print the image size (width $\times$ height).

2. **Preprocess the image.** **[4]**
   Write a function with the following exact signature:

   ```
   def preprocess(pil_img):
       # Resize to 224x224, convert to tensor, normalise with
       # ImageNet mean=[0.485,0.456,0.406] std=[0.229,0.224,0.225].
       # Returns tensor of shape (1, 3, 224, 224),
       # with requires_grad=True.
       transform = transforms.Compose([
           transforms.Resize((224, 224)),
           transforms.ToTensor(),
           transforms.Normalize(mean=[0.485, 0.456, 0.406],
                                std=[0.229, 0.224, 0.225]),
       ])
   ```

```
    x = transform(pil_img).unsqueeze(0)
    x.requires_grad_(True)
    return x
```

Call the function, then print the output shape and dtype.

3. **Make a prediction and decode the top-5 labels.** **[4]**

   Use the ImageNet labels bundled in `torchvision`:

```
weights    = models.MobileNet_V2_Weights.IMAGENET1K_V1
categories = weights.meta["categories"]
```

   Run a forward pass inside `torch.no_grad()`, apply `F.softmax` to get probabilities, and print the top-5 class names with their probabilities.

4. **Implement the saliency function.** **[10]**

   Implement the following function *exactly* as specified:

```
def generate_saliency_map(model, x, target_class=None):
    """
    Parameters
    ----------
    model        : torchvision model in eval() mode
    x            : preprocessed input tensor (1,3,224,224),
                   MUST have requires_grad=True
    target_class : class index; if None use argmax

    Returns
    -------
    saliency   : numpy array, shape (224, 224), values >= 0
    pred_class : int, the class index used
    """
    model.zero_grad()
    logits = model(x)

    if target_class is None:
        target_class = int(logits.argmax(dim=1).item())

    score = logits[0, target_class]
    score.backward()

    saliency = x.grad.abs().squeeze(0)   # (3, 224, 224)
    saliency, _ = saliency.max(dim=0)    # (224, 224)
    return saliency.detach().numpy(), target_class
```

   Important: Verify that `x.grad` is non-`None` after calling `score.backward()`.

5. **Visualise the saliency map.** **[5]**

   Create a single `matplotlib` figure with two subplots side by side:

   - **Left**: the original PIL image
   - **Right**: the saliency map with the `viridis` colormap and a `plt.colorbar()`

   Use the predicted class name as the figure title. Save the figure as `saliency_viridis.png`.

6. **Explore three colormaps.** **[5]**

Replot the saliency map using `hot`, `plasma`, and `gray` in a single row of three subplots. In a markdown cell, write 2–3 sentences explaining which colormap makes the salient regions clearest and why.

7. **Saliency for the second-ranked class.** **[6]**

   Obtain the second-highest predicted class index from Task 3. Call `generate_saliency_map` again with that index as `target_class` (**Note:** call `preprocess` again first, because `backward()` consumes the gradient graph). Display the two saliency maps side by side and comment on any visual differences.

8. **SmoothGrad: reduce noise by averaging.** **[12]**

   Vanilla saliency maps are often noisy. **SmoothGrad** averages saliency maps computed from N copies of the input, each with independent Gaussian noise added:

$$\widetilde{S}(\mathbf{x}) = \frac{1}{N}\sum_{i=1}^{N} S(\mathbf{x} + \boldsymbol{\varepsilon}_i)\,, \qquad \boldsymbol{\varepsilon}_i \sim \mathcal{N}(0, \sigma^2).$$

   Implement the following function:

```
def smooth_grad(model, pil_img, target_class=None,
                n_samples=20, noise_level=0.10):
    """
    Returns the averaged saliency map, shape (224, 224).

    For each of n_samples iterations:
      1. Call preprocess(pil_img) to get a fresh tensor x.
      2. Add Gaussian noise:
             x_noisy = x + noise_level * torch.randn_like(x)
         x_noisy must retain requires_grad=True.
      3. Compute saliency via
             generate_saliency_map(model, x_noisy, target_class).
    Return the element-wise mean of all n_samples saliency maps.
    """
    maps = []
    for _ in range(n_samples):
        x = preprocess(pil_img)
        noise   = torch.randn_like(x) * noise_level
        x_noisy = (x + noise).requires_grad_(True)
        saliency, tc = generate_saliency_map(
            model, x_noisy, target_class)
        if target_class is None:
            target_class = tc
        maps.append(saliency)
    return np.mean(maps, axis=0)
```

   (a) Call `smooth_grad` with default parameters (`n_samples=20`, `noise_level=0.10`).

   (b) Display the vanilla saliency (Task 4) and the SmoothGrad map side by side.

   (c) In a markdown cell, comment in 2–3 sentences on the visual difference between the two maps.

2. Counterfactual Explanations on MNIST (50)

Background. A *counterfactual explanation* answers: *"What is the smallest change to the input pixels that flips the model's prediction to a desired class?"* Unlike tabular data, operating on images makes the counterfactual directly *visually interpretable* — you can see exactly which pixels were modified and by how much. Given a trained model f, an original image $\mathbf{x}$, a target class t, and a desired probability p_t, the counterfactual $\mathbf{x}'$ is found by minimising:

$$\mathcal{L}(\mathbf{x}'|\mathbf{x}) = \big(f_t(\mathbf{x}') - p_t\big)^2 + \lambda \|\mathbf{x}' - \mathbf{x}\|_1$$

via gradient descent *on* $\mathbf{x}'$ (the model weights are frozen). The L_1 term keeps the counterfactual sparse, i.e. as few pixels as possible are changed.

Dataset. Use **MNIST** loaded via `torchvision.datasets.MNIST`: 70,000 greyscale 28×28 images, 10 digit classes (0–9), no extra packages required beyond `torchvision`.

Required packages (first cell of your notebook):

```
!pip install torch torchvision matplotlib numpy
```

Starter imports:

```
import torch
import torch.nn as nn
import torch.nn.functional as F
import torch.optim as optim
from torch.utils.data import DataLoader
import torchvision
import torchvision.transforms as transforms
import numpy as np
import matplotlib.pyplot as plt
```

Tasks:

1. **Load and explore the dataset.** **[2]**
   Load MNIST using the snippet below. Display one example image per digit class (10 images total) in a single row, with the digit label as the subplot title. Print the total number of training and test samples.

```
transform = transforms.Compose([
    transforms.ToTensor(),
    transforms.Normalize((0.1307,), (0.3081,))
])
train_set = torchvision.datasets.MNIST(
    root='./data', train=True,
    download=True, transform=transform)
test_set  = torchvision.datasets.MNIST(
    root='./data', train=False,
    download=True, transform=transform)
```

2. **Build and train a small CNN classifier.** **[3+5]**
   Define the following lightweight CNN (can use different CNN architectures):

```
class SmallCNN(nn.Module):
    def __init__(self):
        super().__init__()
        self.conv1 = nn.Conv2d(1, 16, 3, padding=1)
```

```
        self.conv2 = nn.Conv2d(16, 32, 3, padding=1)
        self.pool  = nn.MaxPool2d(2, 2)
        self.fc1   = nn.Linear(32 * 7 * 7, 128)
        self.fc2   = nn.Linear(128, 10)

    def forward(self, x):
        x = self.pool(F.relu(self.conv1(x)))  # 16x14x14
        x = self.pool(F.relu(self.conv2(x)))  # 32x7x7
        x = x.view(x.size(0), -1)
        x = F.relu(self.fc1(x))
        return self.fc2(x)                    # raw logits
```

(a) Print the model architecture and the total number of trainable parameters.

(b) Train for **5 epochs** using `CrossEntropyLoss` and Adam (`lr=1e-3`), with `batch_size=128`. Record and plot the training loss curve (epoch vs. loss). After training, evaluate on the test set and print test accuracy. The model should reach at least 98% test accuracy.

3. **Implement the counterfactual loss function.** **[6]**

Implement the function below. Note that `x_prime` now has shape $(1, 1, 28, 28)$ — a single greyscale MNIST image:

```
def cf_loss(model, x_prime, x_orig,
            target_class, p_target=0.90, lam=0.10):
    """
    x_prime  : shape (1, 1, 28, 28), requires_grad=True
    x_orig   : shape (1, 1, 28, 28), no grad needed
    Returns scalar tensor:
      loss = (softmax(model(x_prime))[target_class]
               - p_target)^2
           + lam * ||x_prime - x_orig||_1
    """
    model.eval()
    probs       = F.softmax(model(x_prime), dim=1)
    prob_target = probs[0, target_class]
    fidelity    = (prob_target - p_target) ** 2
    l1          = (x_prime - x_orig).abs().sum()
    return fidelity + lam * l1
```

4. **Implement the counterfactual search.** **[8]**

The Adam optimiser acts on `x_prime` only; model weights must **not** change.

```
def find_counterfactual(model, x_orig, target_class,
                        p_target=0.90, lam=0.10,
                        lr=0.05, n_iter=500):
    """
    Returns
    -------
    x_cf      : tensor, shape (1, 1, 28, 28), detached
    loss_hist : list of float, one value per iteration
    """
    x_prime   = x_orig.clone().detach().requires_grad_(True)
    optimizer = optim.Adam([x_prime], lr=lr)
    loss_hist = []
```

```
    for _ in range(n_iter):
        optimizer.zero_grad()
        loss = cf_loss(model, x_prime, x_orig,
                       target_class, p_target, lam)
        loss.backward()
        optimizer.step()
        loss_hist.append(loss.item())
    return x_prime.detach(), loss_hist
```

5. **Run the search on three digit pairs.** **[9]**

   From the test set, find one correctly classified example of each source digit below and search for the stated target:

Source digit	Target digit	Intuition
3	8	closing the open strokes of a 3
1	7	adding the horizontal bar
4	9	closing the loop at the top

   For each pair, display a row of three images side by side:

   - **Original** image (source digit)
   - **Pixel difference** $\mathbf{x}' - \mathbf{x}$ (use a diverging colormap such as `bwr`, centred at zero)
   - **Counterfactual** image (should look like the target digit)

   Below each row, print the original predicted class, the counterfactual predicted class, and the L_1 pixel distance $\|\mathbf{x}' - \mathbf{x}\|_1$.

6. **Plot the loss curves.** **[4]**

   Plot all three loss histories on the same figure with different colours and a legend (e.g. "3→8", "1→7", "4→9"). Label axes and add a title. In a markdown cell, comment in 2–3 sentences on the convergence behaviour across the three pairs.

7. **Effect of the sparsity parameter λ.** **[8]**

   For the **3→8** pair, run `find_counterfactual` with:

   $$\lambda \in \{0.001,\ 0.01,\ 0.10,\ 0.50,\ 2.00\}$$

   keeping all other parameters fixed (`p_target=0.90`, `lr=0.05`, `n_iter=500`).

   (a) For each λ, display the counterfactual image and the pixel-difference map in a grid (5 columns, 2 rows).

   (b) Plot L_1 pixel distance vs. λ (log-scale x-axis recommended).

   (c) Explain in 3–4 sentences what you observe visually as λ increases, and why the L_1 distance changes in the direction it does.

8. **Failure analysis.** **[5]**

   Choose **one** digit pair for which the optimisation converges slowly or the counterfactual image looks unrealistic (not like a recognisable digit). In a markdown cell write 3–5 sentences analysing why this pair is harder than the others: consider the structural similarity of the digit shapes, the decision boundary geometry, and the effect of the L_1 regulariser.

Total: 100 marks ■