23MAT231M-Mathematics for AI

As part of AI&ML Minor Degree

Term Project Report

Even Sem 2024-25

TOPIC:

“Fooling a Convolutional Neural Network”

(using Adversarial Attack: Pixel Perturbations)

Submitted by,

Anjana C V [CB.EN.U4ECE22003]

Adarsh Venugopal [CB.EN.U4ECE22004]

K Harini [CB.EN.U4ECE22026]

P K Vikas [CB.EN.U4ECE22237]

INDEX

I. Description of Problem Statement

"Fooling a Convolutional Neural Network using Adversarial Attack: Pixel Perturbations"

In recent years, Deep Learning Models, especially Convolutional Neural Networks (CNNs), have demonstrated remarkable performance in image-based classification problems.

This project aims to investigate and implement a targeted one-pixel adversarial attack on a CNN classifier trained on the CIFAR-10 dataset. This is to show that Neural Networks are vulnerable to adversarial attacks, where minute, barely recognizable changes to an input image can cause the model to make incorrect predictions with high confidence, since its unable to detect the attack. The attack focuses on perturbing (changing) the values of a very limited number of pixels (as few as one or five) in an image to drastically alter the model's prediction, without significantly affecting the image's appearance to the human eye. This report also discusses the number of possible ways in which we can answer the question "Which Pixel to change"?

The project demonstrates:

- The vulnerability of CNNs to minor, well-placed perturbations.
- The use of particular algorithms to arrive at the pixel number and colour.
- The importance of robust model design and adversarial defence strategies.

Note:

- ❖ For the purpose of achieving results with slight computational ease, the dataset chosen is "CIFAR-10", which provides 32x32 images of various classes.
- ❖ Language chosen for implementation: Python. (.ipynb format)

Justification: MATLAB LiveScript was initially chosen, but due to device constraints and frequent crashes, we chose Python for implementation due to availability of optimised APIs, along with the use of Colaboratory due to the lack of GPU-equipped Laptops.

Dataset Description

Property	Details
Total Images	60,000
Image Size	32 × 32 pixels
Colour Type	Red, Green, Blue (3 colour channels-RGB)
Training Images	50,000
Test Images	10,000
Number of Classes	10
Class Categories	Airplane, Automobile, Bird, Cat, Deer, Dog, Frog, Horse, Ship, Truck

II. Survey of Fooling Methodologies

Multiple methods exist/ can be created to perform pixel perturbations, depending on

1. Size of the dataset
2. Nature of the dataset and its images
3. Level of computational complexity/simplicity
4. Need for automation
5. Level of minuteness in perturbation

Broad Division:

- **Random Pixel Perturbation:** Manually alter a few pixels for each image in the dataset and reclassify till desired results (targeted class, confidence) is attained.
- **Algorithmic Pixel Perturbation:** Run specifically created algorithms that arrive at
  - The position of pixel(s) to alter.
  - R,G,B values of the perturbation pixel.

Types of Algorithmic Perturbation Attacks:

1. **One-Pixel Attack:** Alters a single pixel in an image, causing misclassification despite minimal visible change.
2. **Fast Gradient Sign Method (FGSM):** Perturbs the image in the direction of the loss gradient, causing misclassification.
3. **Optimization-based Attacks:** These attacks use optimization techniques to find the best perturbations that cause misclassification. They iteratively refine the perturbations to maximize the attack's effectiveness, often without requiring access to model gradients.
4. **DeepFool:** Generates minimal perturbations to cross the model's decision boundary, causing misclassification.
5. **Targeted vs. Untargeted Attacks:** Targeted attacks aim for a specific misclassification, while untargeted attacks aim for any misclassification.
6. **Adversarial Patch:** Places a small patch on the image that causes significant misclassification.
7. **Black-box Attacks:** Attacker has no access to the model's parameters, instead querying the model to refine their attack.

III. Our Method

1) Approach

We observe the vulnerability of two different neural network models to adversarial pixel perturbations:

- **PureCNN**: A basic Convolutional Neural Network with standard convolutional and pooling layers.
- **Modified ResNet**: A deeper network architecture based on the ResNet (Residual Network) design, modified for the CIFAR-10 dataset.

Implementation:

- Both networks are trained on the CIFAR-10 dataset.
- Baseline classification accuracy is checked to confirm that both models are well-trained and suitable for evaluation.

Adversarial Attack Strategy:

Untargeted Adversarial Attack: The goal is to fool the model into misclassifying an image, without aiming for a specific incorrect class.

- **Unguided Pixel Perturbation Method**:
    - **Random Generation** of $\{x, y, r, g, b\}$ vectors, where:
        - (x, y) = Pixel position
        - (r, g, b) = New colour values (0–255)
    - The generated vectors are applied to the original image to create a perturbed image.
    - The model's new prediction is recorded.
    - If the perturbed image causes a confident misclassification (prediction probability higher for the wrong class), the attack is considered successful.
- **Optimization Technique- guided Pixel Perturbation Method**:
    - **Differential Evolution (DE)** is used to iteratively refine the pixel positions and values for more effective perturbations.
    - DE searches for the most effective perturbation that reduces the model's confidence in the correct class.
    - DE then provides the $\{x, y, r, g, b\}$ vectors responsible for the perturbation.

Result Observation:

The effect of pixel perturbation is visualized by displaying the original, unaltered image alongside the perturbed image. For each case, the predicted class and its confidence score are compared before and after the perturbation. This is further confirmed by a noticeable drop in the confidence score of the true class, demonstrating the vulnerability of the model to small, strategically placed changes in the image.

2) Mathematical Modelling of Algorithms Involved

In our project, the possible methods to arrive at a perturbation vector are:

1) Manual Pixel Vector Generation

- Manually specify values for pixel positions (x, y) and their colour values (R, G, B).
- Apply the perturbation to the image.
- Reclassify the perturbed image using the neural network.
- Observe whether misclassification occurs and the confidence levels.

2) Randomized Pixel Vector Generation

- Randomly generate pixel positions (x, y) and colour values (R, G, B).
- Apply these random perturbations to the image.
- Reclassify and observe for misclassification and confidence shifts.

3) Differential Evolution (DE) Based Pixel Vector Generation

- Use a **Differential Evolution optimization algorithm** to intelligently search for the pixel positions and colour values that minimize the model's confidence in the true class.
- Apply the optimized perturbation vector to the image.
- Reclassify and record whether confident misclassification is achieved.

Mathematical Modelling of the methods involved:

Let:

- $\boldsymbol{I}$ be the original image of size $32 \times 32 \times 3$
- $\boldsymbol{f(I)}$ be the neural network classifier output (probability vector).
- $\boldsymbol{y_{true}}$ be the true class label.

We define a perturbation vector:

$$\boldsymbol{V} = [(\boldsymbol{x_1}, \boldsymbol{y_1}, \boldsymbol{R_1}, \boldsymbol{G_1}, \boldsymbol{B_1}), (\boldsymbol{x_2}, \boldsymbol{y_2}, \boldsymbol{R_2}, \boldsymbol{G_2}, \boldsymbol{B_2}) \dots \dots \dots \dots, (\boldsymbol{x_N}, \boldsymbol{y_N}, \boldsymbol{R_N}, \boldsymbol{G_N}, \boldsymbol{B_N})]$$

------for N pixels to be perturbed.

I. Manual Pixel Vector Generation

i. Select fixed values for V manually.

ii. Apply perturbation:

$$\boldsymbol{I' = perturb(I, V)}$$

iii. Reclassify:

$$\boldsymbol{f(I')}$$

iv. Observe change in class prediction and confidence.

II. Randomized Pixel Vector Generation

i. Randomly generate V within bounds:

$$x_i, y_i \in [0,31] \quad , \qquad R_i, G_i, B_i \in [0,255]$$

ii. Apply perturbation:

$$\boldsymbol{I' = perturb(I, V)}$$

iii. Reclassify:

$$\boldsymbol{f(I')}$$

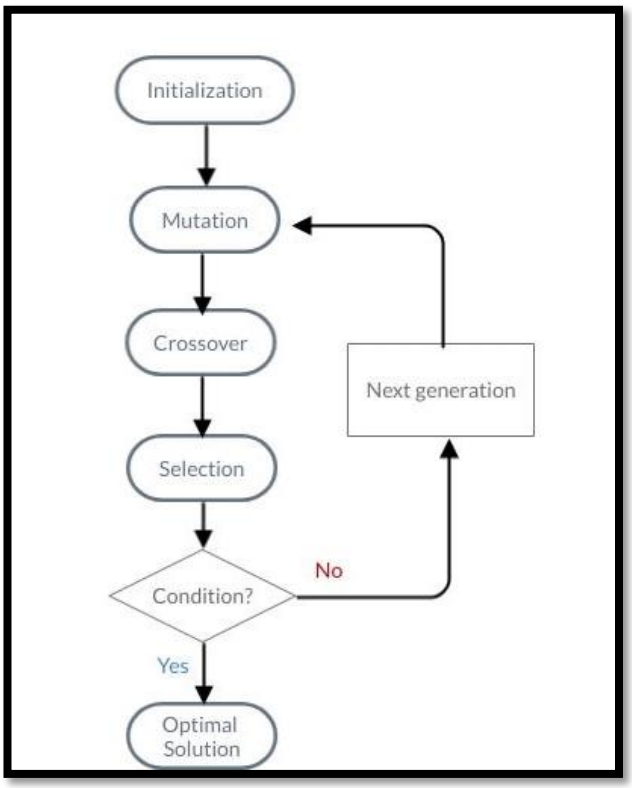
iv. Observe change in class prediction and confidence.

III. Differential Evolution (DE) Based Pixel Vector Generation

We will dive into detail for this method, since that is the chosen algorithm to visualise the final pixel attack.

Differential evolution is an evolutionary algorithm for solving global optimization problems by iteratively improving a candidate solution based on an evolutionary process. Differential Evolution (DE) is a population-based optimization algorithm designed for continuous, nonlinear, and non-differentiable objective functions. It is part of the family of evolutionary algorithms and works through iterative improvement of candidate solutions using mutation, crossover, and selection.

Core Concepts



- **Population:** A group of candidate solutions (vectors), each representing a point in the solution space.
- **Mutation:** For each target vector, a mutant vector is generated by adding the scaled difference of two randomly chosen vectors to a third one. This introduces diversity.
- **Crossover:** The mutant vector is combined with the target vector to produce a trial vector. This step blends information from different individuals.
- **Selection:** The trial vector competes with the target vector. The one with the better objective value survives to the next generation.

Standard DE Strategy Notation: DE/x/y/z

- x: How the base vector is selected (e.g., rand, best)
- y: Number of difference vectors used in mutation (usually 1 or 2)
- z: Crossover method (typically bin for binomial or exp for exponential)

The most commonly used variant is DE/rand/1/bin, which works as follows:

I. Mutation:

$$\boldsymbol{v = x_{r1} + F \cdot (x_{r2} - x_{r3})}$$

where x_{r1} , x_{r2} , x_{r3} are randomly chosen distinct vectors from the population, and F is a scaling factor (typically 0.5 to 0.9).

II. Crossover:

$$u_i = \begin{cases} v_i & \text{if } rand_i \leq P_{cr} \text{ or } i = j_{rand} \\ x_i & \text{otherwise} \end{cases}$$

where P_{cr} is the crossover probability and j_{rand} ensures at least one element from the mutant vector is chosen.

III. Selection:

$$x^{(t+1)} = \begin{cases} u & \text{if } f(u) \leq f(x) \\ x & \text{otherwise} \end{cases}$$

The algorithm repeats this process for a specified number of generations or until convergence.

Simple Example

We aim to minimize the Sphere function:

$$f(x) = x_1^2 + x_2^2$$

with variable bounds: $-5 \leq [\, x_1 \,,\, x_2 \,] \leq 5$

Setup Parameters

- Population size (N): 5
- Dimensions (D): 2
- Max Iterations: 2
- Strategy: DE/rand/1/bin
- Scaling Factor (F): 0.5
- Crossover Probability (Pcr): 0.7

Initial Population

Vector	Coordinates	$f(x)$
x_1	(1.7667, -4.1337)	20.2089
x_2	(4.8071, 0.6642)	23.5502
x_3	(-2.9232, -4.0439)	24.8983
x_4	(-4.3747, -4.7421)	41.6270
x_5	(-1.6587, 0.5680)	3.0741

Sample Step: Mutation and Crossover for $\mathbf{x_1}$

Choose:

- Target: x_4
- Random1: x_5
- Random2: x_3
    -
- **Mutation**:

$$v = x_4 + 0.5 \cdot (x_5 - x_3) = (-4.3747, -4.7421) + 0.5 \cdot (1.2645, 4.6119) = (-3.7425, -2.4362)$$

Crossover with $x_1$and probability $P_{cr} = 0.7 :\ u = (-3.7425, -2.4362)$

- **Selection**:

If $f(u) <\ f(x1);$ keep u ; otherwise, retain x_1

Repeat for all individuals.

Next Generation

After running mutation, crossover, and selection on all vectors, we get:

Vector	Coordinates	$f(x)$
x_1	(-1.3603, -1.9917)	5.8173
x_2	(-1.2491, -0.2358)	1.6158
x_3	(-2.9232, -4.0439)	24.8983
x_4	(0.5233, -0.0876)	0.2815
x_5	(-0.4676, 0.7903)	0.8433

Best solution after 2 iterations: $x = (0.5233, -0.0876), f(x) = 0.2815$

Basic Python code

```
import numpy as np
import pandas as pd

def sphere(x):
    return np.sum(x**2)

# Parameters
pop_size = 5
dimensions = 2
bounds = [-5, 5]
F = 0.5
Pcr = 0.7
generations = 2
np.set_printoptions(precision=4, suppress=True)
np.random.seed(42)

# Initialize population
population = np.random.uniform(bounds[0], bounds[1], size=(pop_size, dimensions))
fitness = np.array([sphere(ind) for ind in population])

def display_table(title, data, fitness):
    df = pd.DataFrame(data, columns=[f"x{i+1}" for i in range(dimensions)])
    df["f(x)"] = fitness
    df.index = [f"x{i+1}" for i in range(len(data))]
    print(f"\n{title}")
    print(df.round(4).to_string())

# Evolution loop
for gen in range(generations):
    display_table(f"Generation {gen} Population", population, fitness)
    new_population = np.copy(population)

    for i in range(pop_size):
        print(f"\nProcessing x{i+1}:")
        indices = list(range(pop_size))
        indices.remove(i)
        r1, r2, r3 = np.random.choice(indices, 3, replace=False)
        print(f"  Selected: r1 = x{r1+1}, r2 = x{r2+1}, r3 = x{r3+1}")

        # Mutation
        mutant = population[r1] + F * (population[r2] - population[r3])
        mutant = np.clip(mutant, bounds[0], bounds[1])
        print(f"  Mutant vector:     {mutant.round(4)}")

        # Crossover
        cross_points = np.random.rand(dimensions) < Pcr
        if not np.any(cross_points):
```

```
            cross_points[np.random.randint(0, dimensions)] = True
        trial = np.where(cross_points, mutant, population[i])
        print(f"  Crossover mask:    {cross_points}")
        print(f"  Trial vector:      {trial.round(4)}")

        # Selection
        trial_fitness = sphere(trial)
        if trial_fitness < fitness[i]:
            print(f"  Replaced x{i+1}: f(x) {fitness[i]:.4f} -> {trial_fitness:.4f}")
            new_population[i] = trial
            fitness[i] = trial_fitness
        else:
            print(f"  Kept x{i+1}: f(x) = {fitness[i]:.4f}")

    population = new_population

# Final output
display_table(f"Generation {generations} Population", population, fitness)
best_idx = np.argmin(fitness)
print(f"\nBest solution after {generations} generations: {population[best_idx].round(4)}")
print(f"Function value: {fitness[best_idx]:.4f}")
```

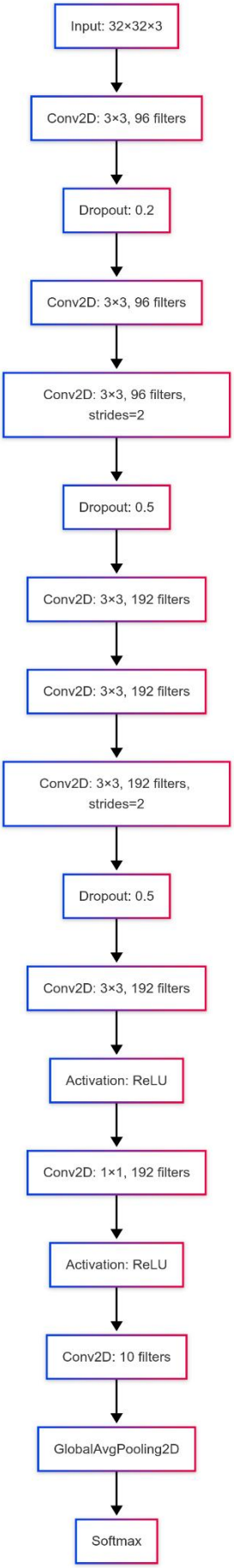
```
In [4]: runfile('C:/SEM 6 RESOURCES/Minor/Pr
RESOURCES/Minor/Project')

Generation 0 Population
        x1      x2     f(x)
x1 -1.2546  4.5071  21.8884
x2  2.3199  0.9866   6.3555
x3 -3.4398 -3.4401  23.6663
x4 -4.4192  3.6618  32.9375
x5  1.0112  2.0807   5.3518

Processing x1:
  Selected: r1 = x4, r2 = x5, r3 = x2
  Mutant vector:     [-5.      4.2088]
  Crossover mask:    [False  True]
  Trial vector:      [-1.2546  4.2088]
  Replaced x1: f(x) 21.8884 -> 19.2883

Processing x2:
  Selected: r1 = x3, r2 = x4, r3 = x1
  Mutant vector:     [-5.     -3.8627]
  Crossover mask:    [ True  True]
  Trial vector:      [-5.     -3.8627]
  Kept x2: f(x) = 6.3555

Processing x3:
  Selected: r1 = x2, r2 = x4, r3 = x1
  Mutant vector:     [0.7377 0.5639]
  Crossover mask:    [ True  True]
  Trial vector:      [0.7377 0.5639]
  Replaced x3: f(x) 23.6663 -> 0.8621
```

```
Processing x4:
  Selected: r1 = x1, r2 = x2, r3 = x3
  Mutant vector:     [1.6253 5.    ]
  Crossover mask:    [ True  True]
  Trial vector:      [1.6253 5.    ]
  Replaced x4: f(x) 32.9375 -> 27.6415

Processing x5:
  Selected: r1 = x1, r2 = x2, r3 = x3
  Mutant vector:     [1.6253 5.    ]
  Crossover mask:    [ True  True]
  Trial vector:      [1.6253 5.    ]
  Kept x5: f(x) = 5.3518

Generation 1 Population
        x1      x2     f(x)
x1 -1.2546  4.2088  19.2883
x2  2.3199  0.9866   6.3555
x3  0.7377  0.5639   0.8621
x4  1.6253  5.0000  27.6415
x5  1.0112  2.0807   5.3518

Processing x1:
  Selected: r1 = x3, r2 = x2, r3 = x5
  Mutant vector:     [1.3921 0.0168]
  Crossover mask:    [ True  True]
  Trial vector:      [1.3921 0.0168]
  Replaced x1: f(x) 19.2883 -> 1.9381
```

3) Description of Models Deployed

1. Pure CNN:

Convolutional Neural Networks (CNNs) have become the de facto standard for image recognition tasks because they can automatically learn spatial hierarchies of features via convolutional filters. The Pure CNN architecture implemented here follows a streamlined design, emphasizing repeated blocks of 3×3 convolutions, dropout for regularization, and global average pooling for output aggregation.



Custom Callback for Visualization:

- P**urpose**

A specialized callback monitors training progress without manual intervention.

- **Behavior**
    - **On Training Begin:** Initializes empty lists to record per-epoch loss and accuracy for both training and validation sets.
    - **On Epoch End:** Appends the latest loss and accuracy measurements to these lists.
    - **On Training End:** Automatically generates side-by-side plots of:
        - **Accuracy over epochs** (training vs. validation)
        - **Loss over epochs** (training vs. validation)

The plots are both displayed inline and saved to Google Drive, providing visual feedback for diagnosing convergence and overfitting.

Data Handling

- **Loading CIFAR-10**
    - The script retrieves the standard CIFAR-10 split: 50,000 training and 10,000 test images, each labelled with one of ten classes.
- **Label Encoding**
    - Raw integer labels are converted to one-hot vectors, preparing them for categorical cross-entropy loss.
- **Color Normalization**
    - Each color channel (red, green, blue) is standardized by subtracting a pre-computed mean and dividing by a pre-computed standard deviation.

Data Augmentation

- **Technique**
    - Real-time image transformations are applied during training to artificially expand the dataset and introduce invariances:
        - **Horizontal flips**
        - **Random shifts** (up to 12.5% of width/height)
        - **Constant padding** for regions newly exposed by shifts
- **Benefit**
    - Augmentation helps the model generalize better to unseen images by preventing over-reliance on specific pixel patterns.

Network Architecture ("Pure CNN")

The model follows a straightforward sequence of convolutional blocks with occasional down-sampling and dropout for regularization:

- **Convolutional Blocks**
  - **Filters** increase from 96 to 192 to capture progressively more abstract features.
  - **3×3 kernels** dominate every convolutional layer—this small receptive field balances locality with computational efficiency.
- **Down-sampling**
  - After certain convolutional pairs, the spatial resolution is halved by using a stride of 2. This reduces computation and increases the effective receptive field.
- **Dropout Layers**
  - Applied at two stages, with rates of 20% then 50%, to randomly deactivate features during training and combat overfitting.
- **Bottleneck and Projection Layers**
  - A 3×3 convolution followed by a 1×1 convolution serves as a "bottleneck," enabling the network to rearrange and mix feature channels efficiently before final classification.
- **Classification Head**
  - A final 1×1 convolution maps features to ten channels (one per class).
  - **Global average pooling** reduces each feature map to a single number by spatial averaging, producing a 10-dimensional vector.
  - A **softmax activation** converts these logits into class probabilities.

Training Strategy

- **Optimizer & Loss**
  - The Adam optimizer is used with an initial learning rate of 0.0001.
  - Categorical cross-entropy serves as the loss function, penalizing the distance between predicted probabilities and true one-hot labels.
- **Learning Rate Scheduling**
  - A custom schedule reduces the learning rate by half after 150 epochs and by another factor of ten after 250 epochs. Gradual decay helps fine-tune weights as the model approaches convergence.
- **Checkpointing**
  - The model's weights are saved whenever validation loss improves. This ensures the best-performing version is preserved, even if later epochs overfit.

1. Convolution

$$Y_{i,j,f} = \sum_{u=1}^{k} \sum_{v=1}^{k} \sum_{c=1}^{C} W_{u,v,c,f} \; X_{i+u-1,\, j+v-1,\, c} + b_f$$

- $\boldsymbol{X_{i,j,c}}$**:** input pixel at row $\boldsymbol{i}$**,** column $\boldsymbol{j}$, channel $\boldsymbol{c}$ (e.g. red/green/blue).
- $\boldsymbol{W_{u,v,c,f}}$**:** weight of the convolutional filter number $\boldsymbol{f}$, at kernel row $\boldsymbol{u}$, column $\boldsymbol{v}$, input channel $\boldsymbol{c}$.
- $\boldsymbol{b_f}$**:** bias term for filter $\boldsymbol{f}$,.
- **k**: kernel size (here k=3 for a 3×3 filter).
- **C**: number of input channels (3 for RGB).
- $\boldsymbol{Y_{i,j,f}}$**:** resulting feature map value at spatial location $(\boldsymbol{i}, \boldsymbol{j})$ for filter $\boldsymbol{f}$.

We slide each $\boldsymbol{k} \times \boldsymbol{k}$ filter over the image, multiply its weights by the corresponding input patch, sum over spatial positions and channels, then add a bias.

2. ReLU Activation

$$Z_{i,j,f} = \max\left(0,\ Y_{i,j,f}\right)$$

- $\boldsymbol{Y_{i,j,f}}$**:** input to the activation (output of the convolution).
- $\boldsymbol{Z_{i,j,f}}$**:** output after activation.

ReLU zeroes out any negative values, keeping positives unchanged. This introduces nonlinearity and prevents negative "signals."

3. Dropout

$$\tilde{Z}_{i,j,f} = M_{i,j,f} \cdot Z_{i,j,f}, \quad M_{i,j,f} \sim \text{Bernoulli}(1-p)$$

- $\boldsymbol{Z_{i,j,f}}$**:** activation before dropout.
- $\boldsymbol{M_{i,j,f}}$**:** random mask value (1 with probability $(\boldsymbol{1-p})$**,** 0 with probability $\boldsymbol{p}$).
- $\boldsymbol{p}$: dropout rate (e.g. 0.5 means 50% of units dropped).
- $\boldsymbol{Z_{\sim i,j,f}}$: : activation after dropout.

Randomly "turn off" a fraction p of activations during training to reduce overfitting.

4. Strided Convolution (Downsampling)

Same formula as §1 but you only compute Y at positions

$\boldsymbol{(i,j)\epsilon}$ **{1,1+s,1+2s,... } for stride s>1**

s: stride length (e.g. s=2 halves spatial resolution).

Skip input positions, reducing the output dimensions by approximately 1/s.

5. 1×1 Convolution (Channel Projection)

$$U_{i,j,f'} = \sum_{c=1}^{F_{\text{in}}} W_{c,f'}^{(1\times1)} \; Z_{i,j,c} \; + \; b_{f'}$$

- $\boldsymbol{Z_{i,j,c}}$:input feature vector (length $\boldsymbol{F_{in}}$) at pixel $(\boldsymbol{i,j})$.
- $\boldsymbol{W_{c,f'(1\times1)}}$: weight connecting input channel ccc to output channel $\boldsymbol{f'}$.
- $\boldsymbol{b_{f'}}$: bias for output channel $\boldsymbol{f'}$.
- $\boldsymbol{U_{i,j,f'}}$: output feature map at $(\boldsymbol{i,j})$,channel $\boldsymbol{f'}$.
- A 1×1 conv is just a small fully-connected layer applied independently at each spatial location to mix channels.

6. Global Average Pooling

$$g_f \; = \; \frac{1}{H\,W} \; \sum_{i=1}^{H} \sum_{j=1}^{W} U_{i,j,f}$$

- $\boldsymbol{U_{i,j,f'}}$ **:** input feature map at location $(\boldsymbol{i,j})$ channel $\boldsymbol{f}$.
- **H×W**: spatial dimensions of $\boldsymbol{U}$.
- $\boldsymbol{g_f}$: single scalar summarizing channel f.

Compute the average of each feature map over all its pixels, collapsing spatial dimensions to a vector of length equal to the number of channels.

7. Softmax

$$\hat{y}_f \; = \; \frac{\exp\left(g_f\right)}{\sum_{k=1}^{F} \exp\left(g_k\right)}$$

- $\boldsymbol{g_f}$: $\boldsymbol{logit(pre-softmax\ score)}$for class $\boldsymbol{f}$.

- $\widehat{y_f}$: predicted probability of class f.
- F: total number of classes (10 for CIFAR-10).

Exponentiate each logit, then normalize by the sum so the outputs form a probability distribution summing to 1.

8. Cross-Entropy Loss

$$\mathcal{L} = -\sum_{f=1}^{F} y_f \log(\hat{y}_f)$$

- y_f : ground-truth one-hot label (1 for the true class, 0 otherwise).
- $\widehat{y_f}$: model's predicted probability for class fff.
- L: scalar loss to minimize.

Penalize the log-probability the model assigns to the correct class; encourages high confidence in the true class.

2. Modified RESNET:

ResNet (Residual Network) is a deep convolutional neural network architecture introduced by Kaiming He et al. in 2015, famous for its ability to train very deep networks — up to hundreds or even thousands of layers — without suffering from the vanishing gradient problem.

The proposed ResNet architecture is based on ResNet-14, a simplified version of ResNet-20 from the paper *"Deep Residual Learning for Image Recognition"* by He et al. This change was made because standard ResNet models are designed for high-resolution images (224×224) and perform poorly on lower-resolution data like ours (32×32). Using a smaller model like ResNet-14 helps retain the benefits of residual learning while better fitting our image size.

Component	Original ResNet-20	Our Version
Input convolutional layer	3×3 convolution, 16 filters	Same
Residual blocks per stage	[3, 3, 3]	[2, 2, 2] → results in shallower network
Residual block type	Basic block	Basic block implemented manually (custom code)
Shortcut connection	Identity or projection (1×1 convolution)	Manually controlled via conv_shortcut flag
Global average pooling	Yes	Yes
Fully connected (dense) layer	10 units (softmax)	Same
Regularization	L2 weight decay	Yes (kernel_regularizer=l2(1e-4))
Data Augmentation	Usually applied	Not included here (can be added)

Key Architectural Modifications

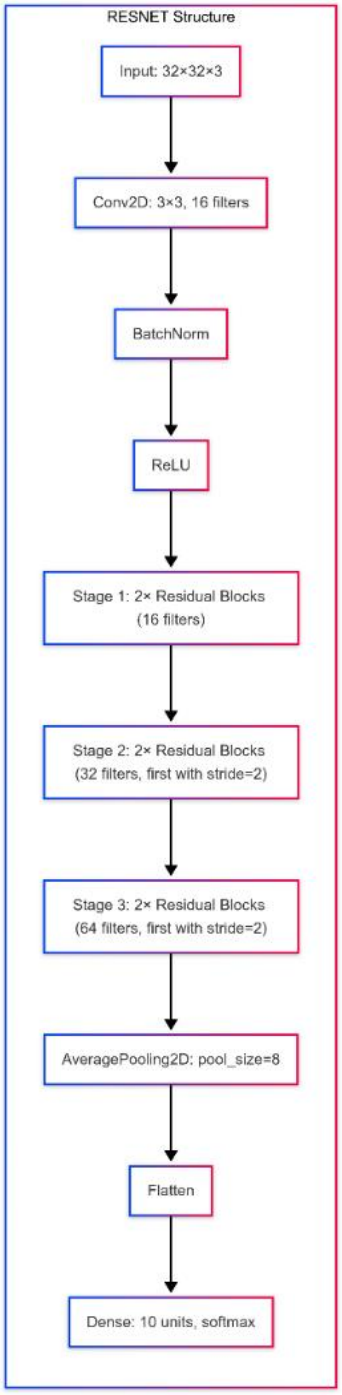
1. **Reduced Depth:**
   - **Change:** num_blocks=[2, 2, 2] (ResNet-14) instead of [3, 3, 3] (ResNet-20).
   - **Reason:** Reduces training time, complexity, and overfitting risk — suitable for limited resources and rapid testing.
2. **Custom Residual Block:**
   - **Structure:** Two Conv2D layers with optional 1×1 convolution for down sampling.
   - **Benefit:** Allows control over skip connections, strides, and filters independently.
3. **No Bottleneck Blocks:**
   - **Reason:** Bottlenecks (1×1 → 3×3 → 1×1) are optimized for deeper networks like ResNet-50+.
   - **Benefit:** Pure 3×3 convolutions are better suited for shallow networks handling small images like CIFAR-10.

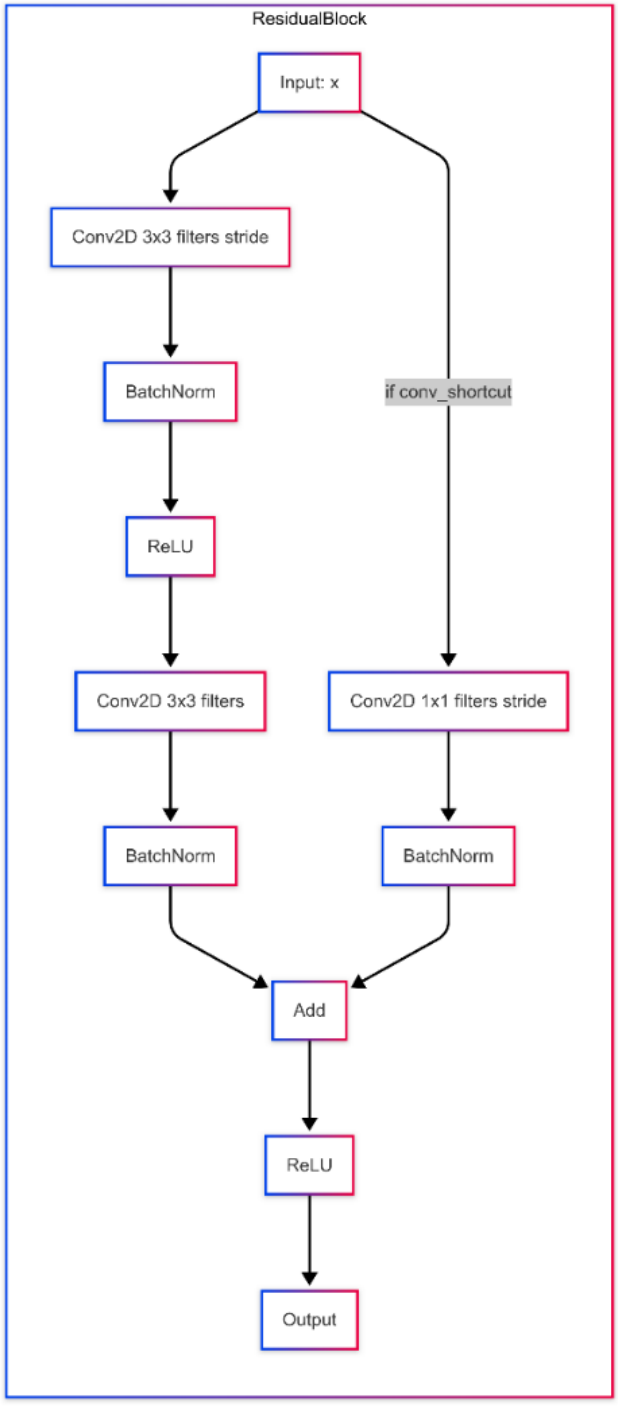
4. **Manual Down sampling:**
    - **Change:** Stride=2 applied in the first block of each new stage, along with a projection shortcut.
    - **Benefit:** Reduces spatial resolution while increasing feature depth, following the ResNet design pattern.
5. **Global Average Pooling and Dense Layer:**
    - **Change:** GlobalAvgPooling2D followed by a Dense(num_classes) layer replaces fully connected layers.
    - **Benefit:** Reduces parameter count, lowers overfitting, and aligns with modern CNN practices.
6. **Batch Normalization and L2 Regularization:**
    - **Applied:** After each convolution, with kernel_regularizer=l2(1e-4).
    - **Benefit:** Stabilizes learning, controls overfitting, and improves generalization.

Model Pipeline Overview

1. **Load and Preprocess Data:**
    - Normalize pixel values to [0,1].
    - Convert class labels to one-hot encoded vectors.
2. **Define Custom ResNet:**
    - Residual blocks with 2× Conv2D layers, Batch Normalization, ReLU, and skip connections.
    - Manual downsampling with stride=2 convolutions and projection shortcuts.
    - Global Average Pooling to condense feature maps, followed by a Dense softmax classifier.
3. **Compile Model:**
    - **Optimizer:** Adam for adaptive learning.
    - **Loss Function:** Categorical Crossentropy for multi-class classification.
    - **Metric:** Accuracy to track correct predictions.
4. **Train with Callbacks:**
    - **ModelCheckpoint:** Saves the best model based on validation accuracy.
    - **ReduceLROnPlateau:** Lowers learning rate when validation loss plateaus.
    - **Validation Split:** 10% of training data used for validation.
    - **Batch Size:** 64
    - **Epochs:** 50

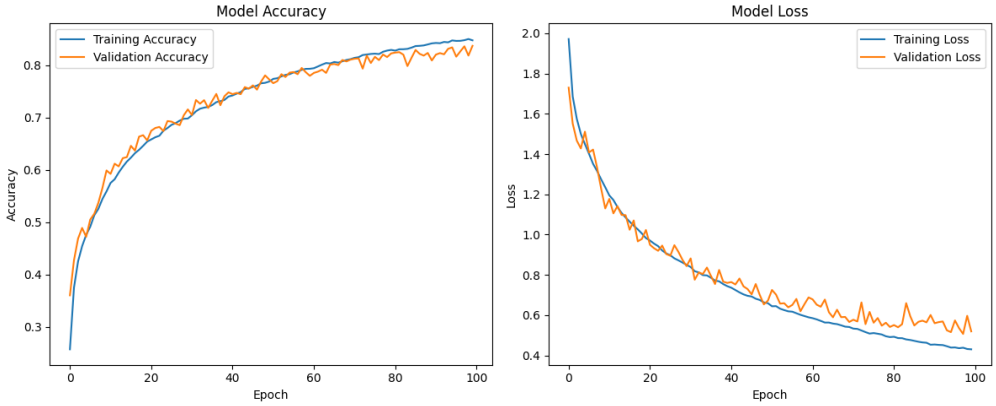
5. **Evaluate on Test Set:**
    - Load the best-performing model weights.
    - Report test loss and accuracy.
    - Generate classification reports with precision, recall, and F1-score.
6. **Export Trained Model:**
    - Save the final trained model as resnet_cifar10.h5 for reuse or deployment.

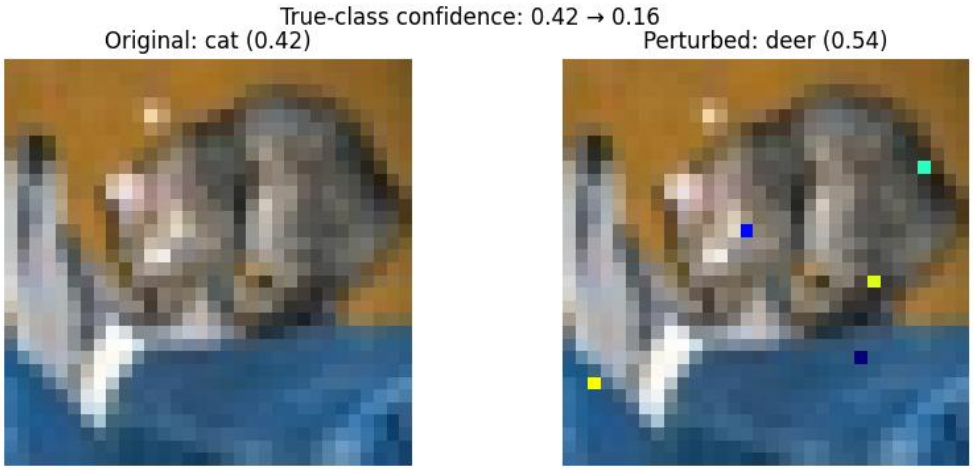




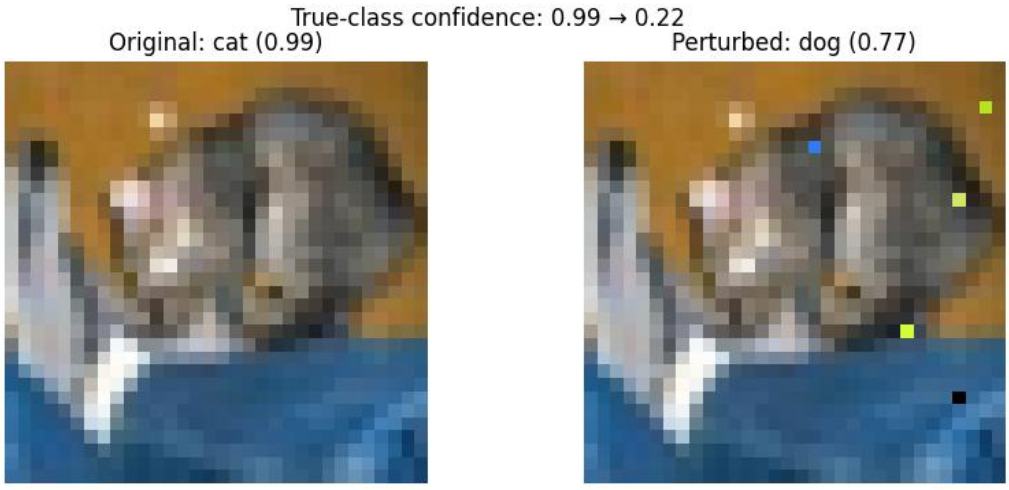
4) Results

- Pure CNN





- Modified ResNet



Summary of Results:

Model	Epochs	Accuracy (%)	Pixels Perturbed	True Class Confidence (Before → After)	Misclassified As	Confidence	DE Max Iter
Pure CNN	100	84.63	5	0.42 → 0.16	Deer	0.54	100
Modified ResNet	50	81.69	5	0.99 → 0.22	Dog	0.77	200

The experiment demonstrates the vulnerability of both a Pure CNN and a Modified ResNet to adversarial perturbations generated using Differential Evolution (DE). Despite differences in architecture and classification accuracy, both models showed significant drops in confidence for the true class after perturbing just **5 pixels**.

Factors impacting the obtained Result:

- **Number of Differential Evolution (DE) Iterations**

  Increasing the number of iterations allows the DE algorithm to converge more effectively and identify better perturbation pixel positions and values. However, this comes at the cost of significantly increased computational time.

- **Initial Model Classification Accuracy**

  The model's baseline classification accuracy and robustness directly influence the effectiveness of perturbation. A stronger, well-regularized model may resist small perturbations better, requiring more sophisticated or intense attacks.

- **Perturbation Algorithm Selection**

  The choice of optimization algorithm (e.g., Differential Evolution in this case) affects both the efficiency of finding adversarial perturbations and the overall attack success rate. Different algorithms have varying trade-offs in terms of speed, computational load, and attack precision.

IV. Difficulties Faced

- Balancing perturbation strength while maintaining image similarity.
- Handling long computation times, especially with high DE iterations.
- Managing device and resource constraints during model training.
- Dataset choice: Used CIFAR-10 (32×32 images) for computational ease. Other image datasets with high-resolution images cannot be used in our case due to computational time & complexity
- Language Choice: Initially attempted in MATLAB LiveScript, but due to device constraints, crashes, lack of GPU, and better API support in Python with Colab for GPU acceleration — final implementation was done in Python (.ipynb).

V. Conclusion

This project demonstrated the vulnerability of deep learning models to adversarial attacks using Differential Evolution (DE). Both Pure CNN and modified ResNet on CIFAR-10 were successfully fooled by modifying just 5 pixels, leading to reduced confidence and misclassification.

Key takeaways:

- Pure CNN achieved 84.63% accuracy, while ResNet reached 81.69% baseline classification accuracy.
- DE was able to reduce the confidence of true class predictions and force misclassifications in both models.

Future Work

- **Test on Larger and Diverse Datasets:** Extend experiments to higher-resolution datasets like CIFAR-100 or ImageNet for broader validation.
- **Implement Alternative Attack Algorithms:** Compare Differential Evolution with other optimization-based and gradient-based attacks like PGD, FGSM, or Genetic Algorithms.
- **Hyperparameter Tuning Automation:** Automate the tuning of DE parameters (population size, mutation factor, crossover rate) for improved attack efficiency.

Thank You