INTERNSHIP REPORT

"ADVERSARIAL MACHINE LEARNING: EXPLORATION AND EXPERIMENTATION ON CNN"

DEFENCE RESEARCH AND DEVELOPMENT ORGANISATION (DRDO)

LABORATORY – SAG (Delhi)

SUBMITTED BY

SHIVAM DANGWAL

(G.B.P.I.E.T. UTTARAKHAND)

UNDER THE SUPERVISION OF

SRHI MOHAMMAD JAVED (Scientist E)

(DRDO SAG)

Index

Chapter 1: Convolutional Neural Network Models

I have built and experimented with various CNN architectures to tackle image classification problems across multiple datasets.

My work includes implementing **MobileNet**, **ResNet**, **VGG**, **AlexNet**, and **LeNet**, as well as utilizing **pre-trained models** and designing **custom CNN architectures** for specific tasks.

These models were applied to benchmark datasets such as **MNIST, CIFAR-10, CIFAR-100, and ImageNet**, allowing me to explore their performance, efficiency, and generalization capabilities.

Through these implementations, I have gained hands-on experience in fine-tuning pre-trained models, optimizing architectures for different datasets, and understanding the strengths and limitations of each model in handling diverse image recognition challenges.

1.1 CNN architecture

1.1.1 LeNet

1.1.1.1 Introduction

LeNet-5 is one of the first convolutional neural networks, developed by Yann LeCun et al. at AT&T Bell Labs for handwritten digit recognition. It was trained via gradient-based backpropagation and achieved state-of-the-art accuracy on the MNIST digit dataset (error $\ll 1\%$ per digit). LeNet demonstrated the feasibility of deep learning for vision tasks and prefigured modern CNN designs. It introduced the basic ideas of CNNs: convolutional feature maps, local receptive fields, weight sharing, and subsampling (pooling).

1.1.1.2 Architectural Structure

LeNet-5 consists of 7 layers with learnable weights: it alternates convolutional and pooling layers followed by fully connected layers.

- **Input Size:** 32x32 pixels.
- **Convolutional Layers:** Three conv layers with 5×5 kernels. The first has 6 filters (tanh activation) and outputs 28×28×6; the second has 16 filters (tanh) and outputs 10×10×16; the third has 120 filters (tanh) and outputs 1×1×120.
- **Pooling Layers:** Two average-pooling layers (2×2, stride 2). One follows the first conv (reducing 28×28 to 14×14) and one follows the second conv (reducing 10×10 to 5×5).
- **Flatten Layer:** After the third conv the 1×1×120 output is flattened into a 120-dimensional.
- **Fully Connected Layers:** Two FC layers. The first FC layer has 84 units (tanh).
- **Output Layer:** 10 neurons, Softmax (classifying digits 0–9).

1.1.1.3 Variants

The "LeNet" name refers to a family of related CNNs (LeNet-1 through LeNet-5). LeNet-5 is the most well-known; earlier versions had fewer layers or filters. Unlike later architectures, LeNet used average pooling and tanh nonlinearities, but its core idea of shared-weight convolutions and pooling persists in all CNNs.

1.1.1.4 Limitations

LeNet is very shallow and narrow by modern standards, limited to small grayscale images. It uses hand-designed pooling and nonoptimal activations (sigmoid), which yield inferior accuracy on

larger, colour datasets. The model is not competitive on complex benchmarks (e.g. CIFAR-10 or ImageNet) and was constrained by 1990s.

1.1.2 AlexNet

1.1.2.1 Introduction

AlexNet is a landmark deep CNN introduced by Krizhevsky et al. that won the 2012 ImageNet Large-Scale Visual Recognition Challenge. It demonstrated the power of deep learning for image classification, achieving far lower error (15.3% top-5) than prior methods. AlexNet popularized many modern deep learning techniques (ReLU activations, dropout, GPU training). The network contained ~60 million parameters across 5 convolutional layers and 3 fully-connected layers. It used ReLU (rectified linear) activations for faster convergence, implemented local response normalization, and applied dropout in the FC layers to prevent overfitting. The model dramatically outperformed the prior state of the art and demonstrated that deep CNNs could effectively utilize large-scale labeled datasets.

1.1.2.2 Architectural Structure

AlexNet consists of 5 convolutional layers followed by 3 fully-connected layers (≈60 million parameters).

- **Convolutional Layers:** Five conv layers all with ReLU.
  (1) 96 filters of size 11×11 (stride 4)
  (2) 256 filters of 5×5
  (3) 384 filters of 3×3
  (4) 384 filters of 3×3
  (5) 256 filters of 3×3
- **Pooling Layers:** Three max-pooling layers (3×3, stride 2) – after conv1, after conv2, and after conv5. Local Response Normalization was also used after conv1 and conv2 in the original.
- **Flatten Layer:** The output of the last pooling is flattened for the FC layers.
- **Fully Connected Layers:** Two hidden FC layers 4096 units each, ReLU with dropout, followed by a third FC 4096→4096→1000.
- **Output Layer:** 1000 neurons, Softmax for ILSVRC classes.

1.1.2.3 Variants

A common variant is CaffeNet (a near-identical architecture used in the Caffe framework). AlexNet's ideas (ReLU, dropout, data augmentation) were adopted in later nets. However, unlike networks like VGG or ResNet, AlexNet itself has no official successors with the same name. Instead, researchers moved to deeper or more efficient architectures (e.g. GoogleNet, ResNet).

1.1.2.4 Limitations

AlexNet's 60M parameters and 1.5B multiply-adds make it computationally heavy by modern standards. It uses large filters (11×11) and many parameters, leading to redundancy and slow inference. It also requires significant GPU memory and is not suitable for real-time or embedded deployment. Accuracy-wise, AlexNet is now outdated (modern models easily surpass its performance). Its training used early techniques (e.g. manual LR schedules) that have since been refined. Nevertheless, AlexNet's historical impact is immense.

1.1.3 VGG

1.1.3.1 Limitations

VGG (Simonyan & Zisserman, 2014) explored the effect of depth on CNN performance. By using very small (3×3) convolution filters and increasing network depth, VGG models achieved top results on ImageNet and finished 1st/2nd in the 2014 ILSVRC classification and localization challenges. VGG's introduction demonstrated that depth, combined with a uniform architecture of small filters, substantially improves accuracy on large-scale image recognition tasks.

1.1.3.2 Architectural Structure

VGG architectures use a homogeneous design. All layers use ReLU activations, and the design omits batch normalization. Total parameters are large: VGG-16 has ≈138 million weights.

- **Convolutional Layers:** 13 conv layers all 3×3, ReLU. They are arranged in 5 blocks: two conv layers with 64 filters; two conv layers with 128 filters; three conv layers with 256 filters; three conv layers with 512 filters; three conv layers with 512 filters. Same padding and stride 1 are used throughout.
- **Pooling Layers:** Five max-pooling layers (2×2, stride 2), one after each block.
- **Flatten Layer:** After the final pooling, the 7×7×512 feature map is flattened.
- **Fully Connected Layers:** Two hidden FC layers of 4096 units (ReLU) each, with dropout between them.
- **Output Layer:** 1000 neurons, Softmax for ImageNet classes.

1.1.3.3 Variants

Standard variants are VGG-16 and VGG-19 (differing in depth). There are also versions with batch normalization (VGG-16/19-BN) which train faster. Some use only two FC layers to reduce parameters. Beyond that, no major re-architecture was done: unlike later nets, VGG does not have skip connections or grouped convs.

1.1.3.4 Limitations

VGG's great depth comes with high cost: ~15 billion FLOPs per image and ~138M parameters. It is slow to run and prone to overfitting if data is limited. VGG lacks modern efficiency tricks (e.g. no bottleneck layers, no residual links), so it is less computationally efficient than networks like ResNet or Inception. In practice it has been largely superseded by deeper or more efficient architectures, but its clear design continues to serve as a baseline.

1.1.4 ResNet

1.1.4.1 Introduction

ResNet (Residual Network) was introduced by Kaiming He et al. to enable training of very deep CNNs. It introduced residual learning: layers learn the difference (residual) between inputs and outputs via identity skip connections. The core idea that bypass one or more convolutional layers, allowing the network to learn a residual mapping $\boldsymbol{F}(x)=h(x)-x$ instead of the full function $h(x)$. This simple change stabilizes gradients and enables training of networks much deeper than previously possible. ResNet won ILSVRC 2015 with a 152-layer model (and 3.57% test error ensemble), demonstrating that depth can continue to improve performance.

1.1.4.2 Architectural Structure

ResNet uses residual blocks instead of plain convolutional stacks. In a block, the input is added to the output of a few-layer convolutional subnetwork i.e. the "shortcut" connection. A typical ResNet-50 has ~25 million parameters, much smaller than VGG-16 despite greater depth.

- **Convolutional Layers:** 50 layers total i.e. counting conv and fully connected.
  (1) An initial 7×7 conv layer with 64 filters stride 2, then 3×3 max-pool stride 2.
  (2) Four "stages" of bottleneck blocks. Stage 1 has 3 blocks; Stage 2 has 4 blocks; Stage 3 has 6 blocks; Stage 4 has 3 blocks. Each bottleneck block has three conv layers (1×1, 3×3, 1×1) with ReLU. The first 1×1 reduces channels, the 3×3 is the core filter, the last 1×1 restores channels).
- **Pooling Layers:** A max-pooling (3×3, stride 2) follows the first conv layer. No intermediate pooling inside blocks, downsampling is done via stride in some convs. After the last block, a global average pooling is applied.
- **Flatten Layer:** No, after global average pooling the features are pooled to a 1×1×2048 vector.
- **Fully Connected Layers:** One FC layer 2048→1000, Softmax.
- **Output Layer:** 1000 neurons, Softmax (ImageNet classes).

1.1.4.3 Variants

Many derivatives of ResNet exist. ResNeXt (Xie et al., 2017) introduces the concept of cardinality by using grouped convolutions inside residual blocks; it showed that increasing the number of parallel paths ("transformations") can improve accuracy more effectively than increasing depth or width. Wide ResNets (Zagoruyko & Komodakis) reduce depth but increase filter widths to similar effect. DenseNet (Huang et al.) replaces residual summation with dense connections, but can be seen as a cousin of ResNet. Overall, the residual paradigm (learning F(x)+x) inspired architectures

like Google's ResNet-based Inception-ResNet and mobile variants (e.g. ResNet-18 in MobileNetV2).

1.1.4.4 Limitations

While much more efficient than very deep plain nets, ResNets still require substantial compute and memory for training. Very deep models (e.g. >200 layers) yield diminishing returns and can be prone to overfitting without extra data. Skip connections complicate architecture (e.g. requiring projection layers when dimensions change). Moreover, ResNet's default blocks are still relatively wide; they are not as parameter-efficient as some recent architectures (e.g. EfficientNet). In practice, ResNets may also suffer from "representation collapse" in extremely deep regimes, requiring specialized training tricks. However, they remain a robust choice for many tasks.

1.1.5 MobileNet

1.1.5.1 Introduction

MobileNet (Howard et al., 2017) is a family of lightweight CNNs designed for mobile and embedded vision applications. It prioritizes low latency and small model size while maintaining reasonable accuracy. Two global hyperparameters allow scaling the model: the width multiplier α uniformly shrinks (or expands) the number of channels, and the resolution multiplier ρ reduces input image size. The core idea is to replace standard convolutions with depth wise separable convolutions, dramatically reducing computation. The 2017 MobileNetV1 demonstrated that one can trade-off between latency and accuracy using simple hyperparameters, making it easier to deploy CNNs on resource-constrained devices.

1.1.5.2 Architectural Structure

A MobileNet base model uses an initial 3×3 standard convolution, followed by a sequence of 3×3 depth wise convolutions each followed by a 1×1 pointwise convolution.

- **Convolutional Layers:** One standard conv layer 3×3, 32 filters, stride 2 followed by 13 depth wise-separable convolution blocks. Each block has a depth wise 3×3 conv followed by a 1×1 pointwise conv and ReLU and BatchNorm used after each. The network roughly doubles channels after each down sampling, ending with 1024 channels before pooling.
- **Pooling Layers:** A final global average pooling over the spatial map is applied before the classifier.
- **Flatten Layer:** No, global average pool replaces flattening.
- **Fully Connected Layers:** One FC layer 1000 units, followed by Softmax.
- **Output Layer:** 1000 neurons, Softmax (ImageNet classes).

1.1.5.3 Variants

The MobileNet family has evolved. MobileNetV2 (Sandler et al., 2018) introduces inverted residual blocks with linear bottlenecks: each block expands channels with a 1×1 conv, applies a 3×3 depthwise conv, then projects back down with another 1×1 conv, skipping activation on the narrow layers. This significantly boosts accuracy per FLOP. MobileNetV2 also pioneered SSDLite, an optimized version of SSD for MobileNet. MobileNetV3 (Howard et al., 2019) further uses Neural Architecture Search and novel blocks (e.g. squeeze-and-excitation) to achieve ~20% faster inference on CPU and higher accuracy; MobileNetV3-Large/Small improve ImageNet top-1 by a few percent over V2 at similar latency. Additionally, the width/resolution multipliers from V1 remain in use across variants.

1.1.5.4 Limitations

Despite efficiency, MobileNets sacrifice some accuracy. The depthwise separable structure also has higher memory access costs and can be slower on some hardware despite fewer multiplies. Very small MobileNets e.g. α=0.25, lose accuracy rapidly. Moreover, MobileNets lack some advanced features (e.g. spatial attention) of larger models. In practice, one must carefully tune α and ρ for the deployment platform to balance speed vs accuracy. Nonetheless, MobileNet's design remains a key standard for efficient CNNs in embedded vision.

1.2 Datasets

1.2.1 MNIST

1.2.1.1 Introduction

The MNIST (Modified NIST) dataset is a benchmark collection of handwritten digit images assembled by Yann LeCun and colleagues (originally derived from NIST data) to foster research in document recognitionarchive.ics.uci.edu. It has become a "hello-world" dataset in machine learning, widely used for testing classification algorithms and deep learning models (for example, the original MNIST paper reported an SVM error rate of ~0.8% on this dataen.wikipedia.org). MNIST's popularity stems from its simplicity and the fact that the images are already centered and size-normalized to 28×28 pixels

1.2.1.2 Dataset structure

- **Size:** 70,000 total examples (60,000 for training, 10,000 for testing).
- **Image format:** Each sample is a 28×28 pixel 8-bit grayscale image (single-channel).
- **Data format:** Officially distributed in IDX file format (4 gzipped files: train-images-idx3-ubyte, etc.). The images are centered within the 28×28 frame and contrast-normalized.

1.2.1.3 Labels and classes

There are **10 classes**, corresponding to the digit values 0 through 9. Each image is labeled with the handwritten digit it represents. The class distribution is roughly uniform (about 6,000 training images per class).

1.2.2 CIFAR-10

1.2.2.1 Introduction

The CIFAR-10 dataset (2009) is a collection of small color images for object recognition research, created by Alex Krizhevsky, Vinod Nair, and Geoffrey Hinton. It is a labeled subset of the "80 million tiny images" dataset cs.toronto.edu. CIFAR-10 was designed to be a more challenging benchmark than MNIST by using natural images of everyday objects. It has become a standard testbed for image classification and deep learning (convolutional neural network) models.

1.2.2.2 Dataset structure

- **Size:** 60,000 total images (50,000 training, 10,000 testing).
- **Image format:** Each image is 32×32 pixels, 3-channel color (RGB).
- **Data batches:** The dataset is provided in five training batches of 10,000 images each, plus one test batch of 10,000 images. Each batch file contains a Python "pickled" NumPy array or a binary file with shape 10000×3072, where each row holds one image (3072 = 32×32×3). The first 1024 bytes are the red channel, next 1024 green, last 1024 blue for each image.
- **Per-class distribution:** 6,000 images per class (5,000 train + 1,000 test per class).

1.2.2.3 Labels and classes

There are 10 mutually exclusive classes in CIFAR-10. The class labels are: airplane, automobile, bird, cat, deer, dog, frog, horse, ship, and truck.

1.2.3 CIFAR-100

1.2.3.1 Introduction

CIFAR-100 is a companion to CIFAR-10 (same authors and creation) that offers a finer-grained classification challenge. It contains the same total number of images (60,000) but with 100 classes instead of 10. Each class has fewer examples, making recognition harder. CIFAR-100 shares the small 32×32 resolution of CIFAR-10 but includes an added hierarchy of 20 "superclasses" for coarse-grained labelscs.toronto.edu.

1.2.3.2 Dataset structure

- **Size:** 60,000 images (50,000 train, 10,000 test).
- **Image format:** 32×32 color (RGB) images, same as CIFAR-10.
- **Class breakdown:** 100 fine classes, each with 600 images (500 train + 100 test). These 100 classes are grouped into 20 superclasses (5 fine classes per coarse label).
- **Data batches:** Provided in the same format as CIFAR-10 (training batches of 10,000 images each). Each image record in the binary files includes two label bytes: the coarse label and the fine label (followed by 3072 pixel bytes).

1.2.3.3 Labels and classes

CIFAR-100's 100 fine classes cover a wide variety of objects and animals grouped into 20 superclasses.

Superclass	Classes
aquatic mammals	beaver, dolphin, otter, seal, whale
fish	aquarium fish, flatfish, ray, shark, trout
flowers	orchids, poppies, roses, sunflowers, tulips
food containers	bottles, bowls, cans, cups, plates
fruit and vegetables	apples, mushrooms, oranges, pears, sweet peppers
household electrical devices	clock, computer keyboard, lamp, telephone, television
household furniture	bed, chair, couch, table, wardrobe
insects	bee, beetle, butterfly, caterpillar, cockroach
large carnivores	bear, leopard, lion, tiger, wolf
large man-made outdoor things	bridge, castle, house, road, skyscraper

large natural outdoor scenes	cloud, forest, mountain, plain, sea
large omnivores and herbivores	camel, cattle, chimpanzee, elephant, kangaroo
medium-sized mammals	fox, porcupine, possum, raccoon, skunk
non-insect invertebrates	crab, lobster, snail, spider, worm
people	baby, boy, girl, man, woman
reptiles	crocodile, dinosaur, lizard, snake, turtle
small mammals	hamster, mouse, rabbit, shrew, squirrel
trees	maple, oak, palm, pine, willow
vehicles 1	bicycle, bus, motorcycle, pickup truck, train
vehicles 2	lawn-mower, rocket, streetcar, tank, tractor

1.2.4 ImageNet

1.2.4.1 Introduction

ImageNet is a large-scale image database created for visual object recognition research. Organized according to the WordNet hierarchy of nouns, ImageNet links each synset (object category) to hundreds or thousands of imagesimage-net.org. Since its release (circa 2009–2010), ImageNet has been "instrumental in advancing computer vision and deep learning research". The annual ImageNet Large Scale Visual Recognition Challenge (ILSVRC) made ImageNet particularly influential; Alex Krizhevsky *et al.* (2012) famously used it to demonstrate a deep convolutional network (AlexNet) that dramatically outperformed prior methods.

1.2.4.2 Dataset structure

- **Total size:** The full ImageNet (circa 2010) contains roughly **14.2 million images** covering **21,841 synsets**. These synsets are mostly concrete nouns (e.g. "cat," "bicycle," "goldfish") filtered from WordNet.
- **ILSVRC subset:** A popular subset (often called ImageNet-1K or ILSVRC-2012) uses *1,000* categories. It includes about **1,281,167 training images**, 50,000 validation images, and 100,000 test images. Each category in ILSVRC is typically a leaf synset (no child categories).
- **Image format:** Images are in RGB color with varying original resolutions. In practice, for model training they are commonly resized/cropped (for example, center-cropped to 224×224) but this is done at application time. The raw dataset images are usually stored as JPEG files.
- **Annotations:** Besides class labels, ImageNet also provides bounding-box annotations for a subset of images (over 1.2M images with bounding boxes) and occasionally segmentation masks for specific challenges.

1.2.4.3 Labels and classes

Categories in ImageNet are WordNet synsets (groups of synonyms for an object concept). For example, there are synsets for objects like *goldfish*, *tabby cat*, *mountain bike*, etc. The 1,000 classes in ILSVRC cover a broad range of everyday objects and animals. In the full hierarchy, there are 21,841 synsets represented, including common categories of animals, plants, vehicles, household objects, and so on.

Labels: IMAGENET 1000 Class List - WekaDeeplearning4j

1.3 Experimentation

1.3.1 MNIST Digit Classification

This project explores solving the MNIST digit classification problem using three different approaches:

a) **LeNet Architecture**
b) **Custom Architecture (Model 1)**
c) **Custom Architecture (Model 2)**

1.3.1.1 Dataset

The MNIST dataset contains 70,000 grayscale images of handwritten digits (0-9). Each image has a resolution of 28x28 pixels. The dataset is divided as follows:

- **Training Set**: 60,000 images
- **Test Set**: 10,000 images

1.3.1.2 Training Data Variety

The training dataset contains the following distribution of digit classes:

Digit (Value)	0	1	2	3	4	5	6	7	8	9
Count	5923	6742	5958	6131	5842	5421	5918	6265	5851	5949

1.3.1.3 Data Loading Functions

To read the dataset, the following functions were implemented:

For Images:

```python
def read_idx3_ubyte(file_path):

    with open(file_path, 'rb') as f:
        magic_number = int.from_bytes(f.read(4), 'big')
        if magic_number != 2051:
            raise ValueError("Magic number is not 2051")

        num_images = int.from_bytes(f.read(4), byteorder='big')
        num_rows = int.from_bytes(f.read(4), byteorder='big')
        num_cols = int.from_bytes(f.read(4), byteorder='big')

        images = np.frombuffer(f.read(), dtype=np.uint8)
        images = images.reshape(num_images, num_rows, num_cols)

        return images
```

For Labels:

```python
def read_idx1_ubyte(labels_path):

    with open(labels_path, "rb") as l:
        magic_number = int.from_bytes(l.read(4), "big")
        if magic_number != 2049:
            raise Exception("Magic number mismatch error")

        num_items = int.from_bytes(l.read(4), byteorder='big')

        labels = np.frombuffer(l.read(), dtype=np.uint8)

        if len(labels) != num_items:
            raise ValueError("Mismatch between number of labels and data size!")

        return labels
```

1.3.1.4 GPU Configuration

To optimize the GPU memory usage, the following configuration was applied:

```python
gpus = tf.config.experimental.list_physical_devices('GPU')
for gpu in gpus:
    tf.config.experimental.set_memory_growth(gpu, True)
```

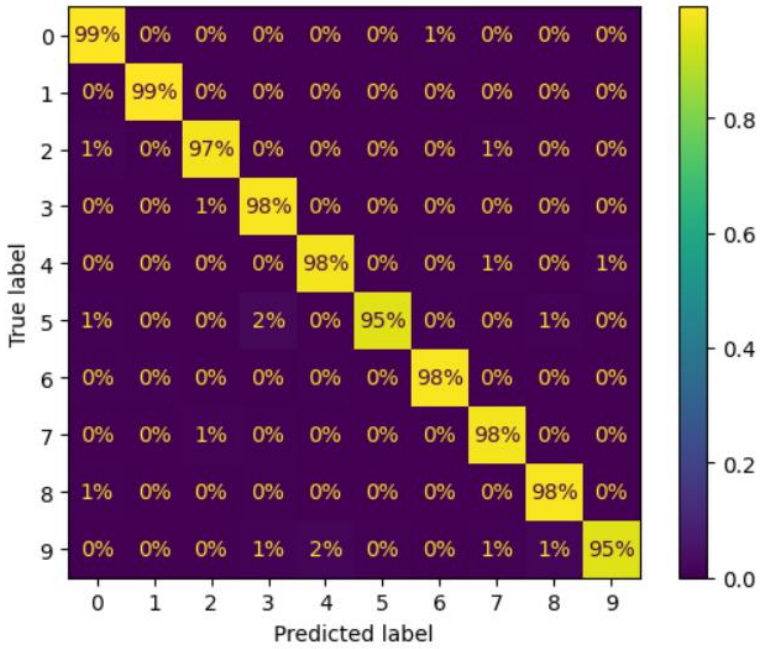
This ensures that TensorFlow dynamically allocates memory as needed during training instead of pre-allocating the entire GPU memory.

1.3.1.5 Approaches

(a) LeNet Architecture

LeNet is a classic Convolutional Neural Network (CNN) designed for handwritten character recognition.

- **Architecture**:
  - Convolutional Layers: Extract spatial features.
  - Subsampling (Pooling): Reduces spatial dimensions.
  - Fully Connected Layers: For classification.
- **Optimizer**: Stochastic Gradient Descent
- **Loss Function**: Sparse Categorical Crossentropy
- **Confusion Matrix:**

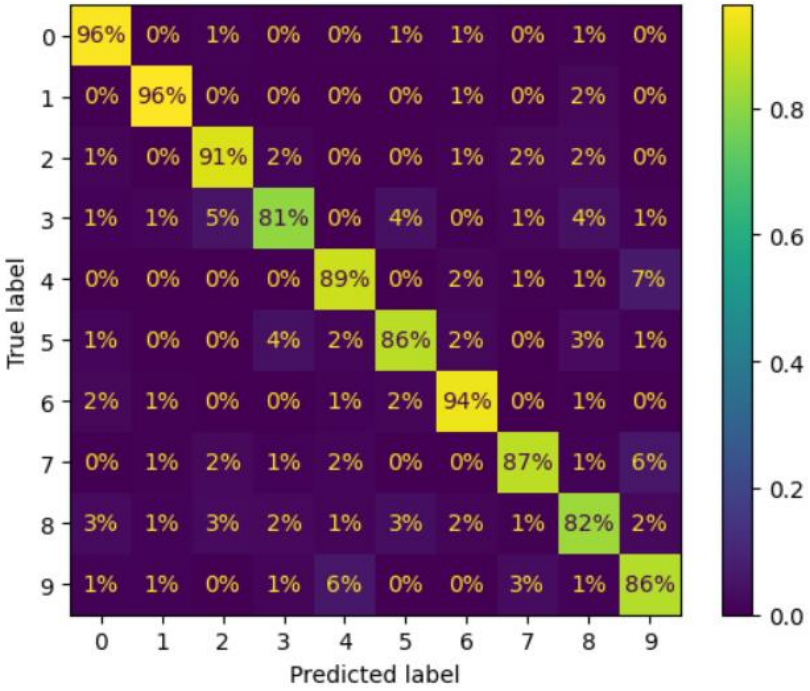


(b) Custom Architecture Model 1

This is a simpler CNN architecture designed to test alternative structures.

- **Features**:
  - **Convolutional Layers**: Two convolutional layers with 16 filters and kernel size (3, 3).
  - **Pooling Layers**: Two max pooling layers to reduce spatial dimensions.
  - **Flatten Layer**: Converts 2D feature maps into a 1D vector.
  - **Fully Connected Layers**:

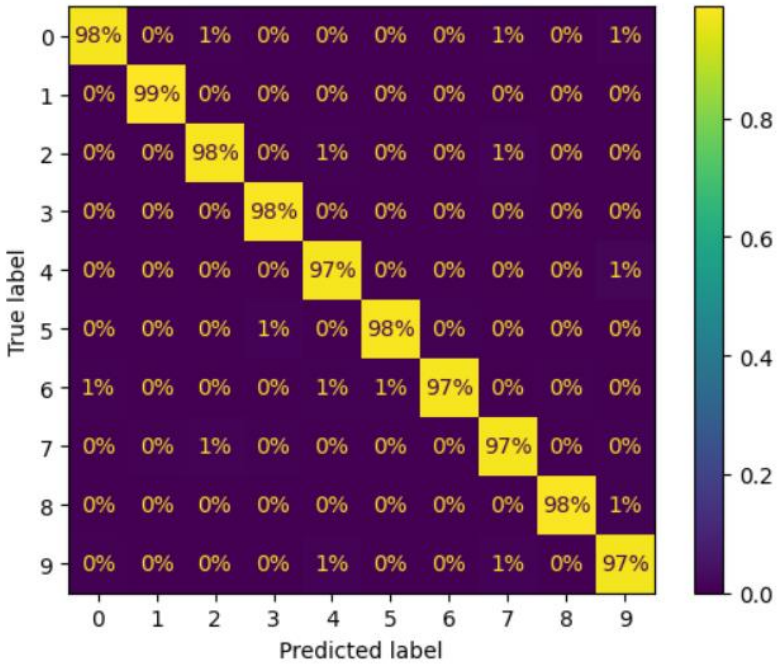
        - Dense Layer with 64 neurons and ReLU activation.
        - Output Layer with 10 neurons (Softmax activation for classification).
- **Optimizer**: Stochastic Gradient Descent
- **Loss Function**: Sparse Categorical Crossentropy
- **Confusion Matrix:**



(c) Custom Architecture Model 2

A deeper and more optimized custom architecture.

- **Features**:
    - **Convolutional Layers**: Three convolutional layers:
        - Layer 1: 16 filters, kernel size (3, 3), ReLU activation.
        - Layer 2: 32 filters, kernel size (3, 3), ReLU activation.
        - Layer 3: 16 filters, kernel size (3, 3), ReLU activation.
    - **Pooling Layers**: Three max pooling layers for spatial dimension reduction.
    - **Flatten Layer**: Converts 2D feature maps into a 1D vector.
    - **Fully Connected Layers**:
        - Dense Layer with 256 neurons and ReLU activation.
        - Output Layer with 10 neurons (Softmax activation for classification).
- **Optimizer**: Stochastic Gradient Descent
- **Loss Function**: Sparse Categorical Crossentropy



1.3.1.6 Results

Model	Test Sparse Categorical Accuracy (%)	Training Time (epochs)	Parameters
LeNet Architecture	**97.61**	15	44,426
Custom Architecture 1	**88.88**	15	28,794
Custom Architecture 2	**97.84**	15	16,346

1.3.1.7 Requirements

- Python 3.8+
- TensorFlow 2.x
- NumPy
- Matplotlib
- Scikit-learn

1.3.1.8 Conclusion

Custom Architecture Model 2 outperformed the others with the highest test Sparse Categorical Accuracy of **97.84%**, followed closely by **LeNet-5** with **97.61%**. **Custom Architecture Model 1** achieved a lower accuracy of **88.88%.**

1.3.2 CIFER-10 Classification

This project focuses on solving the CIFAR-10 image classification problem using three distinct architectures:

a. **AlexNet Architecture**
b. **Custom CNN Architecture (CNN_2)**
c. **VGG Architecture**

1.3.2.1 Dataset

The CIFAR-10 dataset consists of 60,000 color images (32x32 pixels) in 10 classes, with 6,000 images per class. The dataset is split into:

- **Training Set**: 50,000 images
- **Test Set**: 10,000 images

1.3.2.2 Training Data Variety

Data Augmentation: For each original image 2 more images were generated and then merged with the original training dataset.

The training dataset contains the following distribution of digit classes:

Digit (Value)	0	1	2	3	4	5	6	7	8	9
Count	15000	15000	15000	15000	15000	15000	15000	15000	15000	15000

Labels: airplane, frog, horse, ship, truck, dog, deer, cat, bird, automobile

1.3.2.3 Data Loading Functions

To read the dataset, the following functions were implemented:

```python
# Initialize empty lists to hold all data and labels
all_data = []
all_labels = []

def file_open(file_path):
    try:
        with open(file_path, 'rb') as f:
            data_dict = pickle.load(f, encoding='latin1')
            all_data.append(data_dict['data'])  # Append batch data
            all_labels.extend(data_dict['labels'])  # Extend labels
    except FileNotFoundError:
        print(f"File not found: {file_path}")
    except Exception as e:
        print(f"Error loading file {file_path}: {e}")

# Load all batches
base_path = r'.\Dataset\cifar-10-batches-py'
for i in range(1, 6):
    file_path = os.path.join(base_path, f"data_batch_{i}")
    file_open(file_path)

# Convert the list of arrays into a single numpy array
all_data = np.vstack(all_data)  # Stack all data arrays vertically
all_labels = np.array(all_labels)  # Convert labels to numpy array

print(f"Data shape: {all_data.shape}")  # Should be (50000, 3072)
print(f"Labels shape: {all_labels.shape}")  # Should be (50000,)
```

```
Data shape: (50000, 3072)
Labels shape: (50000,)
```

1.3.2.4 GPU Configuration

To optimize the GPU memory usage, the following configuration was applied:

```python
gpus = tf.config.experimental.list_physical_devices('GPU')
for gpu in gpus:
    tf.config.experimental.set_memory_growth(gpu, True)
```

This ensures that TensorFlow dynamically allocates memory as needed during training instead of pre-allocating the entire GPU memory.

1.3.2.5 Approaches

(a) AlexNet Architecture

AlexNet is a well-known deep learning architecture designed for image classification, known for its depth and ability to learn rich feature hierarchies.

- **Architecture Highlights**:
  - **Input Layer**: Takes 32x32x3 color images as input.
  - **Convolutional Layers**:
    - 5 convolutional layers with filters of varying sizes.
      - 1st layer: 96 filters, kernel size (3,3).
      - 2nd layer: 256 filters, kernel size (3,3).
      - 3rd layer: 381 filters, kernel size (3,3).
      - 4th layer: 384 filters, kernel size (3,3).
      - 5th layer: 256 filters, kernel size (3,3).
    - Each layer is followed by "ReLU" activation.
  - **MaxPooling2D layers: R**educe spatial dimensions, 2 layers of pool_size = (2,2)
  - **Fully Connected Layers**:
    - Two dense layers with 400 neurons each.
    - Output layer with 10 neurons for classification.
  - **Optimizer**: Stochastic Gradient Descent
  - **Loss Function**: Sparse Categorical Crossentropy
  - **Metrics:** Sparce Categorical Accuracy

(b) Custom CNN Architecture (CNN_2)

This is a custom-designed CNN tailored to handle the CIFAR-10 dataset.

- **Architecture Highlights**:
  - **Input Layer**: Accepts images of shape 32x32x3.
  - **Convolutional Layers**:
    - Four convolutional layers with "ReLU" activation.
    - Filters:
      - Layer 1: 32 filters, kernel size (3,3).
      - Layer 2: 32 filters, kernel size (3,3).
      - Layer 3: 64 filters, kernel size (3,3).
      - Layer 4: 64 filters, kernel size (3,3).

- **MaxPooling2D layers: R**educe spatial dimensions, 2 layers of pool_size = (2,2)
- **Dropout Layers**: 3 layers, added after each pooling layer and before output layer to reduce overfitting.
- **BatchNormalization layer:** It helps stabilize and accelerate the training process.
- **Fully Connected Layers**:
  - Dense Layer: 128neurons with ReLU activation.
  - Output Layer: 10 neurons with Softmax activation.
- **Optimizer**: Stochastic Gradient Descent
- **Loss Function**: Sparse Categorical Crossentropy
- **Metrics:** Sparce Categorical Accuracy

(c) VGG Architecture

VGG is a deeper CNN architecture that uses smaller filter sizes (3x3) but stacks more layers to improve learning capability.

- **Architecture Highlights**:
  - **Input Layer**: Accepts 32x32x3 color images.
  - **Convolutional Layers**:
    - 6 convolutional layers with 3x3 filters.
      - Layer 1: 64 filters
      - Layer 2: 64 filters
      - Layer 3: 256 filters
      - Layer 4: 256 filters
      - Layer 5: 256 filters
      - Layer 6: 256 filters
    - Each convolution is followed by ReLU activation.
  - **MaxPooling2D layers: R**educe spatial dimensions, 3 layers of pool_size = (2,2) and strides = (2,2)
  - **GlobalAveragePooling2D:** Used to minimizes the risk of overfitting
  - **Dropout Layer**: 1 layer, added before output layer to reduce overfitting.
  - **Fully Connected Layers**:
    - One dense layer with 2000 neurons.
    - Output layer with 10 neurons with Softmax activation.
  - **Optimizer**: Stochastic Gradient Descent
  - **Loss Function**: Sparse Categorical Crossentropy
  - **Metrics:** Sparce Categorical Accuracy

1.3.2.6 Results

Model	Test Sparse Categorical Accuracy (%)	Training Time (epochs)	Parameters
LeNet Architecture	**79.9**	25	6,027,815
Custom Architecture 2	**77.4**	50	272,234
VGG Architecture	**79.5**	25	2,490,682

1.3.2.7 Requirements

- Python 3.8+
- TensorFlow 2.x
- NumPy
- Matplotlib
- Scikit-learn
- pickle

1.3.2.8 Conclusion

LeNet outperformed the others with the highest test Sparse Categorical Accuracy of **77.9%**, followed closely by **VGG** with **77.5%**. **Custom Architecture Model 2** achieved a lower accuracy of 77.4**%.**

1.3.3 CIFAR-100 Classification

This project focuses on solving the CIFAR-100 image classification problem using two distinct architectures:

a. **ResNet Architecture**
b. **Pretrained-Xception Architecture**

1.3.3.1 Dataset

The CIFAR-100 dataset consists of 60,000 color images (32x32 pixels) in 100 classes, with 6,000 images per class. The dataset is split into:

- **Training Set**: 50,000 images
- **Test Set**: 10,000 images

1.3.3.2 Training Data Variety

Data Augmentation: For each original image 2 more images were generated and then merged with the original training dataset.

The training dataset contains **15000** instances of each class:

Labels:
beaver, dolphin, otter, seal, whale, aquarium fish, flatfish, ray, shark, trout, orchids, poppies, roses, sunflowers, tulips, bottles, bowls, cans, cups, plates, apples, mushrooms, oranges, pears, sweet peppers, clock, computer keyboard, lamp, telephone, television, bed, chair, couch, table, wardrobe, bee, beetle, butterfly, caterpillar, cockroach, skyscraper, cloud, forest, mountain, plain, sea, camel, cattle, chimpanzee, elephant, kangaroo, fox, porcupine, possum, raccoon, skunk, crab, lobster, snail, spider, worm, baby, boy, girl, man, woman, crocodile, dinosaur, lizard, snake, turtle, hamster, mouse, rabbit, shrew, squirrel, maple, oak, palm, pine, willow, bicycle, bus, motorcycle, pickup truck, train, lawn-mower, rocket, streetcar, tank, tractor, bear, leopard, lion, tiger, wolf, bridge, castle, house, road.

1.3.3.3 Data Loading Functions

To read the dataset, the following functions were implemented:

```python
import pickle
file_path = r'.\cifar-100-python\cifar-100-python\train'

with open(file_path, 'rb') as f:
    data_dict = pickle.load(f, encoding='latin1')

print(data_dict.keys())
```

```
dict_keys(['filenames', 'batch_label', 'fine_labels', 'coarse_labels', 'data'])
```

1.3.3.4 GPU Configuration

To optimize the GPU memory usage, the following configuration was applied:

```python
gpus = tf.config.experimental.list_physical_devices('GPU')
for gpu in gpus:
    tf.config.experimental.set_memory_growth(gpu, True)
```

This ensures that TensorFlow dynamically allocates memory as needed during training instead of pre-allocating the entire GPU memory.

1.3.3.5 Approaches

(a) ResNet Architecture

A **residual neural network** is a deep learning architecture in which the layers learn residual functions with reference to the layer inputs.

- **Architecture Highlights**:
  - **Residual Blocks**: Enables deeper networks by using shortcut connections to prevent vanishing gradients.
    - **Shortcut Connection:** If conv_shortcut=True, the input is used as the shortcut, a 1x1 convolution is applied to match the dimensions. Followed by Batch Normalization.
    - **First Convolution Layer:**
      - ◊ Uses a 3×3 kernel with stride=1
    - Followed by **Batch Normalization** and **ReLU** activation.
    - **Second Convolution Layer:**

        - ◊ Uses another 3×3 kernel with padding='same'.
      - Followed by Batch Normalization.
      - The shortcut is added to the processed tensor to allow gradient flow.
      - **Final Activation:** ReLU activation.
  - **ModifyNet Block:** This function builds a complete ResNet model using multiple **residual blocks**.
      - **Input Layer**: Takes 32x32x3 color images as input.
      - **1st Convolutional Layer**: 128 filters, 3×3 kernel
      - Followed with **Batch Normalization.**
      - **Activation:** ReLU activation.
      - **Dropout**: Reduces overfitting, dropout_rate=0.5.
      - **Residual Blocks:** Configured using num_blocks_list=[2, 2, 2, 2], meaning:
        - ◊ 2 blocks in each stage.
        - ◊ Each stage has **double the number of filters** from the previous one.
        - ◊ **Strides=2** for the first block in each new stage (downsampling)
      - **Global Average Pooling**: Reduces model complexity instead of using fully connected layers.
      - **Final Dense Layer**: 100 output neurons with a softmax activation
  - **Optimizer**: Model was compiled 3 times, adam(learning_rate=0.001), adam(learning_rate=0.0001) and adam(learning_rate=0.00005)
  - **Loss Function**: Sparse Categorical Crossentropy
  - **Metrics:** Sparce Categorical Accuracy

(b) Pretrained Xception Architecture

The **Xception** (Extreme Inception) model is a **deep convolutional neural network** that utilizes **depthwise separable convolutions** for more efficient feature extraction. This implementation **transfers learning** from **ImageNet** and fine-tunes it for CIFAR-100.

- **Architecture Highlights**:
  - **Preprocessing Function:** Resizes images to **71x71** to match the input requirements of **Xception**.
  - Uses tf.keras.applications.xception.preprocess_input() to normalize pixel values.
  - **Base Model: Xception (Pretrained on ImageNet)**
  - include_top=False: Removes fully connected layers.
  - **Global Average Pooling** (GlobalAveragePooling2D): Reduces feature maps into a single vector.
  - **Dense Layer** with **512 neurons** with ReLU activation.

- **Dropout Layer (0.5)** to prevent overfitting.
- **Output Layer** with **100 neurons** with softmax activation.
- **Freezing Pretrained Layers:** All layers of **Xception** are **frozen** to retain pretrained knowledge from **ImageNet** for **20 epochs.** This allows the model to **focus on learning CIFAR-100-specific features** in the final layers.
- **Unfreezing Pretrained Layers:** For **26 epochs.**
- **Optimizer**: Model was compiled 3 times, adam(learning_rate=0.0002) , adam(learning_rate=0.000005) and adam(learning_rate=0. 000002)
- **Loss Function**: Sparse Categorical Crossentropy
- **Metrics:** Sparce Categorical Accuracy

1.3.3.6 Results

Model	Test Sparse Categorical Accuracy (%)	Training Time (epochs)	Parameters
ResNet Architecture	**65.49**	26	46,203,236
Pretrained Xception	**72.93**	45	21,961,868

1.3.3.7 Requirements

- Python 3.8+
- TensorFlow 2.x
- NumPy
- Matplotlib
- Scikit-learn
- pickle

1.3.3.8 Conclusion

Pretrained Xception outperformed the others with the highest test Sparse Categorical Accuracy of **72.93%**, followed closely by **ResNet** with **65.49%**.

1.3.4 ImageNet Classification

This project explores solving the ImageNet classification problem using two different approaches:

a. **MobileNet Architecture**
b. **Pre-trained Architecture**

1.3.4.1 Dataset

The ImageNet dataset was downloaded from Kaggle, total 4 directories were downloaded which were merged to become 2: Train 1 and Train 2, and Test was also downloaded. Each image has a resolution of 256x256 pixels. The dataset is divided as follows:

- **Train Set 1**: 6,39,639 images
  - ImageNet-1k-0
  - ImageNet-1k-1
- **Train Set 2**: 6,40,658 images
  - ImageNet-1k-2
  - ImageNet-1k-3
- **Test Set**: 50,000 images
  - ImageNet-1k-valid

1.3.4.2 Training Data Variety

The dataset contains the following classes:

- text: imagenet 1000 class idx to human readable labels (Fox, E., & Guestrin, C. (n.d.). Coursera Machine Learning Specialization.)

1.3.4.3 Data Loading Functions

To read the data from directories, the following functions were implemented:

```
data = tf.keras.utils.image_dataset_from_directory('Train 2', image_size=(128, 128), batch_size=45)
```

data[0] contains: Images

data[1] contains: Labels
and data is a MapDataset

1.3.4.4 GPU Configuration

To optimize the GPU memory usage, the following configuration was applied:

```
gpus = tf.config.experimental.list_physical_devices('GPU')
for gpu in gpus:
    tf.config.experimental.set_memory_growth(gpu, True)
```

This ensures that TensorFlow dynamically allocates memory as needed during training instead of pre-allocating the entire GPU memory.

1.3.4.5 Data Preprocessor

- An iterator was created to inspect the first batch of raw data before applying preprocessing steps.

```
data_iterator = data.as_numpy_iterator()
test_iterator = test.as_numpy_iterator()
```

```
train_batch = data_iterator.next()
test_batch = test_iterator.next()
```

- Data Normalization: pixel values were normalized by scaling them to a range of [0, 1].
- After normalization, a new iterator was created to verify the preprocessed data:

```
scaled_iterator = data.as_numpy_iterator()
test_scaled_iterator = test.as_numpy_iterator()
```

```
batch = scaled_iterator.next()
test_batch = test_scaled_iterator.next()
```

- **as_numpy_iterator()** converts the dataset into a NumPy-compatible iterator
- **next()** retrieves the **next batch** of data from the iterator.

1.3.4.6 Data Visualization

- A subset of 16 images from a batch of training data was displayed to confirm the dataset's format and ensure proper labeling.

```
fig, ax = plt.subplots(ncols=4, nrows=4, figsize=(20, 20))
ax = ax.flatten()
for idx, img in enumerate(train_batch[0][:16]):
    ax[idx].imshow(img.astype(int))
    ax[idx].title.set_text(train_batch[1][idx])
```

- The visualization process was repeated to ensure the images and their labels were correctly displayed after data normalization.

1.3.4.7 Approaches

(a) MobileNet Architecture

MobileNet is a lightweight Convolutional Neural Network (CNN), it uses depth wise separable convolutions to reduce computational cost while maintaining accuracy. Images were resized to 128*128

- **Convolutional Layer:**
  - Initial Conv2D layer with 32 filters and kernel size (3,3).
  - Batch Normalization and ReLU activation for stability.
- **MobileNet Blocks:**
  - **Depth wise Convolution:** 3×3 filter applied per input channel.
  - **Pointwise Convolution:** 1×1 convolution to combine features.
  - Batch Normalization and ReLU activation after each step.
- **Feature Extraction:**
  - **Stride Convolutions:** Reduce spatial dimensions while extracting key patterns.
  - **Repeated Depth wise Separable Convolutions:** Five stacked blocks with 512 filters.
- **Flatten Layer:** Converts 2D feature maps into a 1D vector.
- **Fully Connected Layers:**
  - **Dense Layer:** 1000 neurons with Softmax activation for classification.
- **Optimizer**: Adam
- **Loss Function**: Sparse Categorical Crossentropy
- **Epochs:** 40

(b) Pre-trained Architecture

This is a simpler CNN architecture designed to test alternative structures.

- **Base Model:**
  - MobileNetV2 pretrained on ImageNet.
  - include_top=False removes the original classification head.
  - **Input shape:** (224, 224, 3) (RGB images).
  - Weights initialized from ImageNet.
- **Feature Extraction:**
  - **Base Model Frozen:** Prevents weight updates in the pretrained layers.
  - Extracts high-level image features.
- **Global Average Pooling:** Converts feature maps into a feature vector.
- **Fully Connected Layers:**
  - Dense(512, ReLU) for deeper feature representation.
  - Dense(256, ReLU) for further abstraction.
- **Output Layer:**
  - Dense(1000, softmax) for multi-class classification.
- **Optimizer**: Stochastic Gradient Descent
- **Loss Function**: Sparse Categorical Cross entropy
- **Epochs:** 10

1.3.4.8 Results

Model	Test Sparse Categorical Accuracy Top 1 (%)	Top 5 Accuracy	Training Time (epochs)	Parameters
MobileNet Architecture	**37.41**	62.54	40	7,538,280
Pre-trained Architecture	**58.35**	80.25	10	3,302,184

1.3.4.9 Requirements

- Python 3.8+
- TensorFlow 2.x
- NumPy
- Matplotlib
- Scikit-learn

1.3.4.10 Conclusion

Pre-trained Architecture outperformed the others with the highest test Sparse Categorical Accuracy of **58.35%**, followed closely by **MobileNet Architecture** with **37.41%**.

Chapter 2: Adversarial Machine Learning

I have explored **Adversarial Machine Learning** by implementing various attack and defense strategies on CNN models such as **MobileNet, ResNet, VGG, AlexNet, LeNet, and custom CNN architectures**.

In the domain of **evasion attacks**, I have utilized techniques like **Fast Gradient Sign Method (FGSM), Carlini & Wagner (L2) Attack, HopSkipJump, and DeepFool** to assess model vulnerabilities.

For **model extraction**, I implemented the **CopycatCNN** attack to replicate model behavior.

In **poisoning attacks**, I experimented with **Poisoning Backdoor and Clean Label Backdoor Attacks** to manipulate training data and compromise model integrity.

To counter these adversarial threats, I have applied various **defense mechanisms** such as **Adversarial Training, Feature Squeezing, Gaussian Augmentation, and JPEG Compression**, enhancing model robustness against adversarial perturbations.

This work has deepened my understanding of adversarial vulnerabilities in CNNs and the effectiveness of different defense strategies in mitigating attacks.

2.1 Adversarial Machine Learning using MNIST Dataset

2.1.1 GitHub Repository

- Adversarial MNIST

2.1.2 Overview

This project demonstrates several adversarial machine learning techniques applied to the

MNIST handwritten digits dataset.

- **Data preprocessing:** Loading and normalizing MNIST images.
- **Model handling:** Loading a pretrained LeNet-style model.
- **Evasion attack generation:** Using the Fast Gradient Sign Method (FGSM) to craft adversarial examples.
- **Defensive measures:** Applying adversarial training to mitigate the effects of adversarial examples.
- **Poisoning (Backdoor) attack:** Introducing a backdoor trigger into a subset of the training data to force misclassification.

2.1.3 Data Loading and Preprocessing

- **Custom Data Readers**: The notebook defines two helper functions to read MNIST data from IDX files: one for image data (read_idx3_ubyte) and one for labels (read_idx1_ubyte). These functions validate the magic numbers of the files to ensure correct formatting.
- **Normalization and Reshaping:** After loading, the image pixel values are normalized to a [0, 1] range (by dividing by 255) and reshaped to add an extra channel dimension.
- This prepares the data for processing by a convolutional neural network (CNN).

2.1.4 Model Setup

- Model: A model saved as model_leNet.h5 is loaded and compiled using TensorFlow.
- Classifier Wrapper: To facilitate adversarial attacks, the project wraps the Keras model using ART's TensorFlowV2Classifier. This abstraction makes it easier to generate adversarial examples and to later apply defenses.

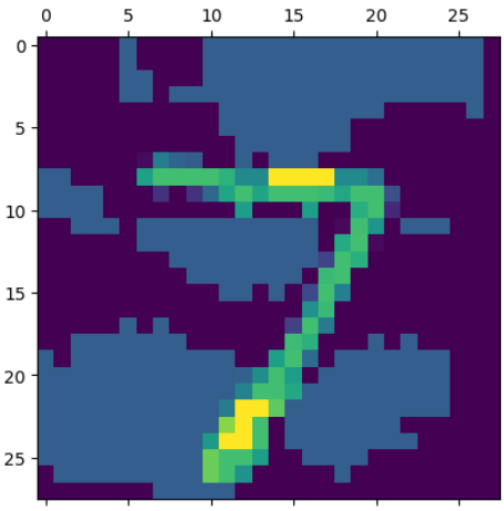
2.1.5 Evasion Attack with Fast Gradient Sign Method (FGSM)

- **Attack Implementation:** The FGSM, a well-known evasion attack method, is imported from the ART library and instantiated with the wrapped classifier. FGSM perturbs the input images by computing the gradient of the loss with respect to the input pixels.

```
attack_fgsm = FastGradientMethod(
    estimator=classifier,
)
```

- **Adversarial Example Generation and Evaluation:**
  - The attack is applied on the test dataset, and the adversarial examples are generated.
  - Then we compute the accuracy on these adversarial samples and measures the average perturbation applied.
  - **Results:** In one run, the reported accuracy on adversarial test data was very low (around 3.08%) with average perturbation of 0.16, indicating that the original model is highly vulnerable to such perturbations.

```
plt.matshow(test_images_adv[0, :, :, 0])
plt.show()
```



2.1.6 Defense via Adversarial Training

- **Adversarial Training Strategy:** To counter the evasion attack, we implement adversarial training using ART's Adversarial Trainer. Adversarial examples (generated by FGSM) are included during training so that the model learns to be robust against these perturbations.

- **Training Process and Evaluation:** The trainer fits the model using adversarial perturbed data.
- Post training, the model is re-evaluated on both clean and adversarial test sets.
- An improvement is observed in the adversarial accuracy (with one run reporting about 92.94% on adversarial examples), though the clean accuracy can be affected (e.g., around 49.92% in that instance). This trade-off is typical when defending against adversarial inputs.

2.1.7 Poisoning Attack with Backdoor Trigger

- **Backdoor Attack:** We demonstrate a poisoning attack where a backdoor trigger is inserted into the training images.
- **Trigger Mechanism:** A simple trigger is defined by modifying a small patch (the bottom-right corner) of an image. All images that receive this trigger are assigned a target label (in this case, the digit '7').
- **Attack Execution:** Using ART's PoisoningAttackBackdoor, the training dataset is "poisoned" by embedding this trigger pattern. Later training with this poisoned data can force the model to misclassify any input that contains the backdoor trigger.
- **Implications:** This segment underscores how minimal changes in the training data can have significant consequences on model behavior, illustrating the real-world risks of poisoning attacks.
- Backdoor Success Rate: 11.21%

2.1.8 Result

- The LeNet-style CNN achieved 97.61% accuracy on clean MNIST test data.
- When subjected to FGSM adversarial attacks, the model's accuracy dropped dramatically to 3.08%, confirming its vulnerability.
- After applying adversarial training, the model improved its robustness, achieving 92.94% accuracy on adversarial test data, though accuracy on clean data dropped to 49.92%, indicating a trade-off.
- A backdoor poisoning attack yielded a success rate of 11.21%, proving the model's susceptibility to hidden trigger manipulation despite high clean accuracy.

2.1.9 Conclusion

- This file demonstrates the vulnerability of CNNs, such as LeNet, to adversarial attacks using the MNIST dataset. FGSM was used to craft adversarial examples, causing a noticeable drop in classification accuracy.

- Adversarial training proved effective in improving the model's robustness against such perturbations. A backdoor poisoning attack was also introduced, leading to targeted misclassifications. This highlights the risk of hidden triggers in training data.
- Defensive techniques can mitigate these risks but may not eliminate them entirely.
- Robustness often comes at the cost of clean-data performance and computational overhead. Ensuring model security requires a balance of accuracy, efficiency, and resilience to attacks.

2.2 Adversarial Machine Learning using CIFAR-10 Dataset

2.2.1 GitHub Repository:

- Adversarial CIFAR-10

2.2.2 Overview

This project demonstrates adversarial attacks and defenses on the CIFAR-10 image classification dataset.

- **Data Loading and Preprocessing:** Reading CIFAR-10 data stored in Python pickle format, reshaping the raw image arrays, and normalizing pixel values.
- **Model Loading:** Importing a model for CIFAR-10 classification.
- **Attack Analysis:** Identifying low-confidence predictions and using the Carlini & Wagner L2 attack method (CarliniL2Method) to generate adversarial examples.
- **Adversarial Dataset Creation:** Crafting a set of adversarial examples from test data.
- **Defense Strategies:** FeatureSqueezing from art.defences.preprocessor

2.2.3 Data Loading and Preprocessing

- **Dataset Aggregation:** The notebook reads multiple CIFAR-10 training batches (using Python's pickle module) and stacks them to form a complete training set of 50,000 samples. Each sample is a flat array of 3072 values (representing a 32×32 RGB image).
- **Data Frame Conversion and Reshaping:** The raw data is then converted into a Pandas Data Frame where each row corresponds to an image and its label. Each flat image is reshaped into a 3D array with shape (32, 32, 3) by transposing the data appropriately. The pixel values are normalized by dividing by 255 and rounded to two decimal places.
- **Test Set Loading:** Similarly, the test data is loaded from the CIFAR-10 test batch, processed, reshaped, normalized, and stored in a Data Frame before being converted to NumPy arrays for further use.

2.2.4 Model Setup

- **Pretrained Model:** The notebook loads a pretrained model from a saved file (VGG.h5). After loading, the model is compiled using TensorFlow with the sparse categorical cross entropy loss and SGD optimizer.

- **Classifier Wrapping:** For compatibility with the Adversarial Robustness Toolbox (ART), the model is wrapped using TensorFlowV2Classifier. This wrapper facilitates generating adversarial examples and evaluating the model's performance under attack.

2.2.5 Identifying Low-Confidence Predictions

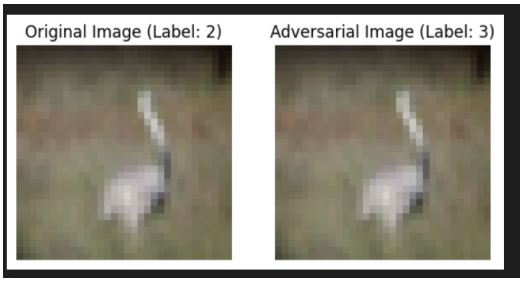
- **Prediction and Confidence Analysis:** The model predicts class probabilities for the entire test set. The notebook computes the maximum probability (confidence score) for each prediction, then sorts these scores to identify the 50 samples with the lowest confidence.
- **Visualization:** An example of one of the original test images is displayed using matplotlib, allowing visual inspection of a sample that the model finds challenging.

2.2.6 Adversarial Attack Using CarliniL2Method

- **Attack Configuration:** We import the CarliniL2Method from the ART library and instantiates it with specific parameters such as a low learning rate and a small initial constant. This attack is designed to generate minimal perturbations that lead the model to correct its classification.

```
attack_cw_4 = CarliniL2Method(
    classifier=classifier,
    learning_rate=0.01,
    initial_const=0.01,
)
```

- Generating Adversarial Examples: The attack is applied specifically to a subset of low-confidence images (selected from the test set) to generate adversarial examples. CarliniL2Method was used on both for model's correct prediction and for incorrect prediction.
- For Correct Prediction: Average perturbation: 0.0000142759



2.2.7 Creating an Adversarial Dataset

- **Crafting New Data:** After generating adversarial examples using the CarliniL2Method, the notebook demonstrates how to create a new adversarial test dataset. This dataset can be used to evaluate model performance under adversarial conditions or to further train a more robust model.

2.2.8 Defense Strategies

- **Exploration of Defenses:** FeatureSqueezing using art.defences.preprocessor to protect input images
- **Trade-Off Discussion:** It suggests that a balance must be found between robustness (accuracy on adversarial data) and performance on clean data.
- **Accuracy of FeatureSqueezing:** It was tested on **model's least confidence images** therefore, accuracy on adversarial data before feature Squeezing: 0.04 and accuracy on adversarial data After feature Squeezing: 0.32.

2.2.9 Result

- The pretrained model achieved 79.5% accuracy on the clean CIFAR-10 test set.
- Carlini & Wagner L2 attack was applied to low-confidence images, with an average perturbation of 0.0000148, effectively misleading the model.
- Accuracy on adversarial data before Feature Squeezing was as low as 4%, revealing the model's vulnerability.
- After applying Feature Squeezing, accuracy on adversarial data improved to 32%, demonstrating the defense's partial effectiveness. These evaluations were performed on the model's lowest-confidence images.
- Clean data performance remained high with Feature Squeezing (~80.6%), indicating minimal trade-off in normal conditions.

2.2.10 Conclusion

- This file demonstrates the vulnerability of CNNs, such as VGG, to advanced adversarial attacks on the CIFAR-10 dataset. The Carlini & Wagner L2 method was used to generate subtle perturbations that significantly reduced model accuracy.
- Feature Squeezing was applied as a defensive strategy, improving adversarial accuracy from 4% to 32%. These tests focused on the model's lowest-confidence predictions to assess real-world vulnerability.
- The findings highlight that even small perturbations can cause confident misclassifications.

- Defensive techniques offer partial protection but may affect clean-data performance.
- Achieving robustness requires trade-offs in accuracy, complexity, and efficiency.

2.3 Adversarial Machine Learning using CIFAR-100 Dataset

2.3.1 GitHub Repository:

- Adversarial_CIFAR-100

2.3.2 Overview

- **Importing Libraries**
- **Data and it's preprocessing:** Reading CIFAR-100 data stored in Python pickle format, reshaping the raw image arrays, and normalizing pixel values.
- **Model:** Importing a model for CIFAR-100 classification.
- Calculating model's 50 lowest confidence predictions
- **HopSkipJump:** It is decision-based attack. HopSkipJumpAttack requires significantly fewer model queries than Boundary Attack. It also achieves competitive performance in attacking several widely-used defense mechanisms.
- **CopyCatCNN:** Copycat CNN attack is a model extraction attack, where an adversary attempts to steal the knowledge of a black-box model by training a surrogate (thieved) model using the original model's predictions.
- **Adversarial Dataset:** Crafting a set of adversarial examples from test data.
- **Defence:** AdversarialTrainer, FeatureSqueezing, GaussianAugmentation.

2.3.3. Data Loading and Preprocessing

The notebook reads the CIFAR-100 training data using Python's pickle module. Each sample is a flat array of 3072 values (representing a 32×32 RGB image).

- **DataFrame Conversion and Reshaping:** The raw data is converted into a Pandas DataFrame where each row corresponds to an image and its label. Each flat image is reshaped into a 3D array with shape (32, 32, 3) by transposing the data appropriately. The pixel values are normalized by dividing by 255 and rounded to two decimal places.
- **Test Set Loading:** Similarly, the test data is loaded from the CIFAR-100 test batch, processed, reshaped, normalized, and stored in a DataFrame before being converted to NumPy arrays for further use.

2.3.4 Model Setup

- **Model:** The notebook loads a model from a saved file. After loading, the model is compiled using TensorFlow with the appropriate loss function and optimizer.

- **Classifier Wrapping:** For compatibility with the Adversarial Robustness Toolbox (ART), the model is wrapped using TensorFlowV2Classifier. This wrapper facilitates generating adversarial examples and evaluating the model's performance under attack.

2.3.5 Identifying Low-Confidence Predictions

- **Prediction and Confidence Analysis:** The model predicts class probabilities for the entire test set. The notebook computes the maximum probability (confidence score) for each prediction, then sorts these scores to identify the samples with the lowest confidence.
- **Visualization:** Examples of original test images with low confidence predictions are displayed using matplotlib.

2.3.6 Adversarial Evasion Attack Using HopSkipJump

- Attack Configuration: Imports the HopSkipJump attack from the ART library and instantiates it with specific parameters. This attack is designed to generate adversarial examples with minimal perturbations.
- Generating Adversarial Examples: The attack is applied to a subset of low-confidence images (selected from the test set) to generate adversarial examples. The results, including adversarial images, can be visualized to see how subtle perturbations can alter the classifier's output.

```
Original Prediction: 21
Adversarial Prediction: 33
```

```
Average perturbation: 0.0051255631
```

2.3.7 CopyCatCNN

- The first 5,000 training images and labels are used for querying the target model. The adversary does not have access to ground-truth labels, only the target model's predictions.
- The extract function trains the stolen classifier using pseudo-labels provided by the target model (classifier). The stolen model (stolen_classifier) learns to mimic the behaviour of the original classifier.

```
classifier2 = TensorFlowV2Classifier(
    model=model,
    nb_classes=100,
    input_shape=(32, 32, 3),
    clip_values=(0, 1),
    loss_object=loss_object,
    optimizer=optimizer,
    train_step=train_step,
    channels_first=False
)
```

```
from art.attacks.extraction import CopycatCNN
cc_attack = CopycatCNN(classifier = classifier2, batch_size_fit=5,
                       batch_size_query=10,
                       nb_epochs=5,
                       nb_stolen=100)
```

2.3.8 Creating an Adversarial Dataset

This dataset is used to evaluate model performance under adversarial conditions and to further train a more robust model.

```
attack = HopSkipJump(classifier=classifier, targeted=False, max_iter=15)
adversarial_data = attack.generate(selected_images)
```

```
HopSkipJump: 100%|██████████| 100/100 [13:01<00:00,  7.81s/it]
```

2.3.9 Defence Strategies

- **AdversarialTrainer:** trainer was trained on subset of dataset for 1 epoch, then feartureSqueezing and Gaussian Augmentation was applied.
- **Trade-Off:** As with many adversarial scenarios, a balance must be found between robustness (accuracy on adversarial data) and performance on clean data.
- **Trainer Performance after 1 epoch:**
    - Accuracy on adversarial data: **36.00%**
    - Accuracy after FeatureSqueezing the data: 41.33%
    - Accuracy after Randomized Smoothing data: 40.35%
    - Accuracy on adversarial data after trainer was fit on FeatureSqueezing and GaussianAugmentation data: **45.00%**

2.3.10 Result

- The pretrained model achieved 72.93% accuracy on the clean CIFAR-100 test set.

- HopSkipJump black-box attack was applied to the lowest-confidence test sample, minimal perturbation = 0.005 was required to change the label.
- CopyCatCNN model-stealing attack successfully replicated the behavior of the original model using only 5,000 unlabeled samples and the target model's output probabilities.
- Applying Feature Squeezing improved adversarial accuracy from 36.00% to 41.33%.
- With Randomized Smoothing (Gaussian Augmentation), adversarial accuracy improved slightly further to 40.35%.
- Combining Adversarial Training, Feature Squeezing, and Gaussian Augmentation resulted in an accuracy from 36.00% to 45.00% on adversarial examples.

2.3.11 Conclusion

- **Vulnerability of Black-Box Models:** The Copycat CNN attack shows that proprietary ML models can be stolen with a few thousand queries.
- **Attack Insights:** It shows that even a well-trained model can be vulnerable to sophisticated adversarial attacks such as the HopSkipJump method.
- **Defence Considerations:** Defence Techniques like FeatureSqueezing and Randomized Smoothing can help model to become more robust against HopSkipJump attack.
- Achieving robustness requires trade-offs in accuracy, complexity, and efficiency.

2.4 Adversarial Machine Learning using ImageNet Dataset

2.4.1 GitHub Repository:

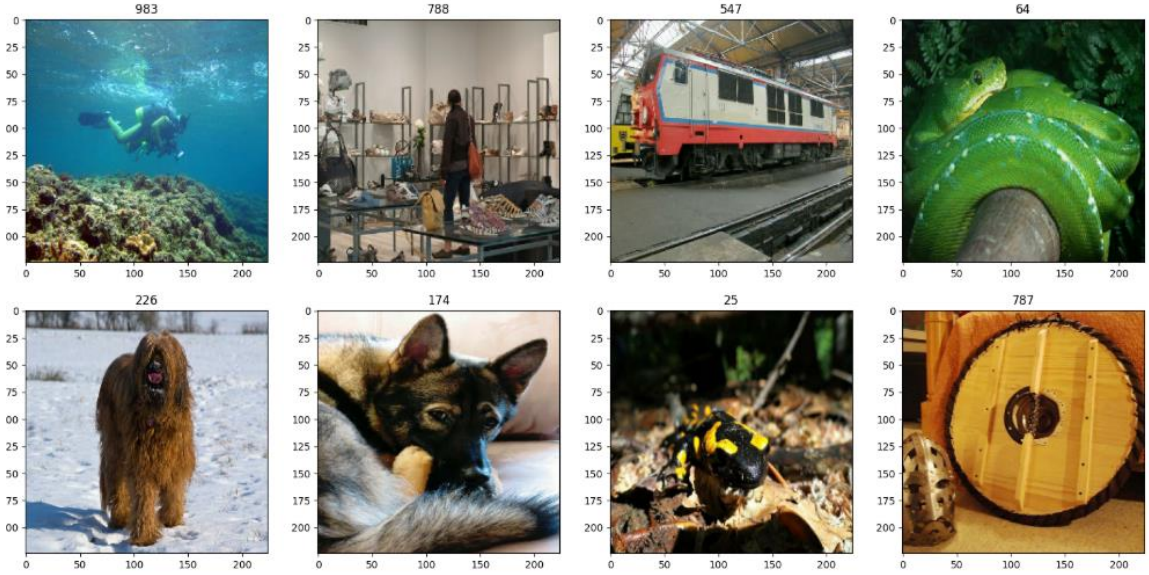
- Adversarial_ImageNet

2.4.2 Overview

- **Importing Libraries**
- **Data and it's preprocessing**
- **Model**
- **Model's 50 lowest confidence predictions**
- **Deepfool**
- **Clean Label Backdoor attack**
- **Defence**

2.4.3 Data Loading and Preprocessing

- **Data Acquisition:** A subset of ImageNet images Train 1 and Train 2 are loaded for demonstration.
- TensorFlow dataset is converted into a NumPy iterator using the **as_numpy_iterator()** method.
- The pixel values are normalized by dividing by 255.
- After mapping (normalizing) the dataset, a new NumPy iterator is created to iterate over the updated data.
- The **next()** method is then used to retrieve the first normalized batch.



2.4.4 Model Setup

- **Model:** We load a pre trained deep convolutional neural network.
- **Compilation and ART Integration:** The model is compiled with an appropriate loss function (categorical cross entropy) and optimizer (RMSprop). For adversarial experiments, the model is wrapped using the ART (Adversarial Robustness Toolbox) wrapper (e.g., TensorFlowV2Classifier). This integration allows seamless generation of adversarial examples and evaluation of the model's robustness.

```
classifier = TensorFlowV2Classifier(model=model,
                                    nb_classes=1000,
                                    input_shape=(224, 224, 3),
                                    clip_values=(0, 1),
                                    loss_object=loss_object,
                                    channels_first=False)
```

2.4.5 Identifying Low-Confidence Predictions

- **Prediction and Confidence Analysis:** The model predicts class probabilities for the entire test set. Then computes the maximum probability (confidence score) for each prediction, then sorts these scores to identify the samples with the lowest confidence.

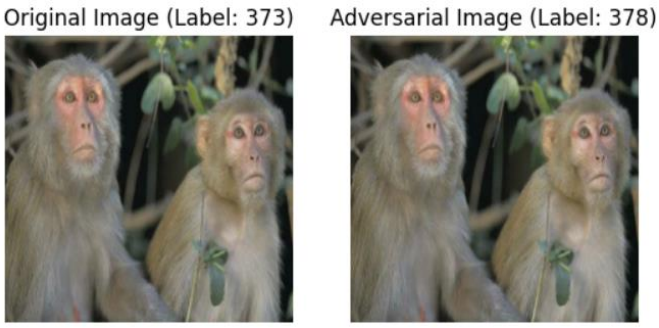
2.4.6 Adversarial Evasion Attack Using DeepFool

- **DeepFool** is an untargeted adversarial attack method that you can use to measure a model's robustness by finding the minimal perturbation needed to change the classifier's decision.

```
from art.attacks.evasion import DeepFool
attack = DeepFool(classifier)
adv_images = attack.generate(original_image)
```

```
DeepFool: 100%|██████████| 1/1 [00:45<00:00, 45.25s/it]
```

- **Results Visualization:** One of generated adversarial images is visualized alongside their original counterparts to illustrate the imperceptibility of the perturbations.



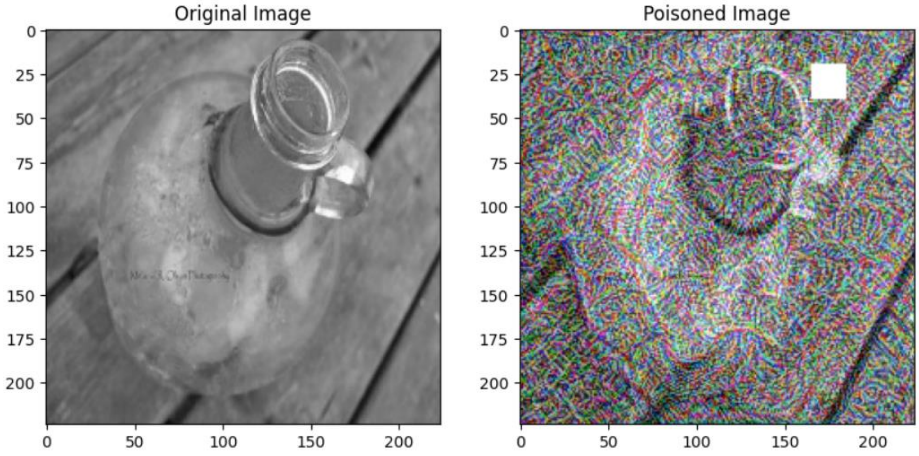
Average Perturbation: 0.0000702761

2.4.7 Clean Label Backdoor Attack

- A Clean Label Backdoor Attack is a form of poisoning where an attacker injects a trigger into a subset of training images without altering their labels.
- **Trigger Definition:** A function (e.g., add_trigger) is used to overlay a subtle pattern on the images.

```
class SimplePatchBackdoor(PoisoningAttackBackdoor):
    def __init__(self, size=20, value=1):
        self.size = size
        self.value = value

    def apply(self, x):
        x_copy = x.copy()
        patch_value = 1 if x_copy.max() <= 1 else 255

        for i in range(x_copy.shape[0]):
            h, w = x_copy.shape[1:3]
            top = np.random.randint(0, h - self.size)
            left = np.random.randint(0, w - self.size)
            x_copy[i, top:top+self.size, left:left+self.size, :] = patch_value

        return x_copy

    def perturbation(self, x):
        return self.apply(x)


backdoor = SimplePatchBackdoor(size=20, value=0)
```

- A single batch of model's least confidence images are selected for poisoning.



2.4.8 Defence

- **JpegCompression:** It acts as an input transformation that can reduce or remove subtle, high-frequency perturbations introduced by these attacks.
- By compressing and then decompressing the image, JPEG processing tends to smooth out fine-grained noise that **DeepFool** relies on, often restoring the image closer to its original, correctly classified form.

- In **Clean Label Backdoor Attacks**, a subtle trigger is added to a subset of training images. Because this trigger is designed to be imperceptible, it usually exists in the fine details of the image. JPEG compression can weaken or even remove these subtle patterns, thereby reducing the backdoor's effectiveness.

2.4.10 Result

- The average perturbation introduced by DeepFool was 0.00007 on model's lowest-confidence predictions, showing that even tiny changes can mislead a model.
- In the Clean Label Backdoor Attack, subtle triggers embedded in low-confidence samples effectively influenced model behavior without changing class labels, demonstrating the risks of data poisoning.
- Applying JPEG Compression as a defense reduced the effectiveness of both DeepFool and backdoor attacks, although it also decreased the model's accuracy from 68.75% to 62.50%.

2.4.10 Conclusion

- While JPEG compression can help mitigate these attacks, it is not a silver bullet. It works best as part of a broader defense strategy that might also include adversarial training, defensive distillation, or other preprocessing techniques.
- It shows that even a well-trained model can be vulnerable to adversarial attacks such as the DeepFool method.
- Achieving robustness requires trade-offs in accuracy, complexity, and efficiency.