PROJECT

Topic - Deep Learning for Malware Detection Using Images

FINAL INTERNSHIP PROJECT

Team members —> Bhumi (ECE - AI) , Trisha Rawat (CSE-AI)

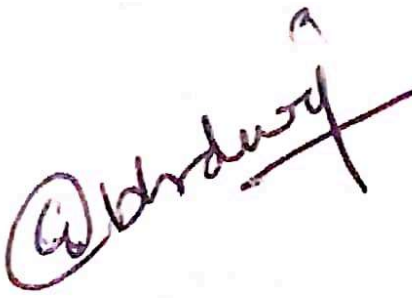
College Name→ INDIRA GANDHI DELHI TECHNICAL UNIVERSITY OF WOMEN

Email id → bhumi72301@gmail.com, trisrawat07@gmail.com

Internship → Neuroscribe under Ms. Gargi bhardwaj , Ms Tarushi , Ms Abhiruchi

Field of internship → Deep learning using python

Date → 3 June 2025

Signature of mentor - 

Abstract

Malware keeps changing to hide from antivirus tools. Normal machine learning methods do not work well against such tricks. This project uses deep learning to change malware files into images and then classify them. A dataset of 18,800 obfuscated (hidden) malware programs was made. Models like CNN and bi-directional LSTM gave around 93% accuracy. The models were also tested on well-known malware datasets (MsM2015 and 76665MalImg). Transfer learning showed that the features learned from one dataset also worked well on others. This proves that using images and deep learning can make malware detection more powerful and scalable.

Problem Statement

Malware is hard to detect because hackers use methods like obfuscation and polymorphism to hide code. Traditional tools only find known malware, not new ones. Due to this, there is a need for smart and automatic systems that can find even new or changed malware. This study focuses on turning malware files into images and using CNN and LSTM models to detect and classify them more accurately.

Brief Summary

This work uses a deep learning method that treats malware as images. Each malware file is converted into a grayscale image to see its pattern. The dataset has 124,804 samples from 18 classes (17 malware families and 1 benign). Models like Random Forest, SVM, CNN, and Bi-LSTM were used. CNN and Bi-LSTM gave the best accuracy (around 93%). The model was also tested on other malware datasets and worked well, showing that it can handle real-world malware.

Objectives

- Build a model to detect malware even if it is hidden or changed.
- Convert malware files into images for better pattern recognition.
- Apply CNN and Bi-LSTM models for high accuracy.
- Use transfer learning to make the model work on multiple datasets.
- Compare deep learning results with traditional machine learning methods.
- Create a system that is fast, scalable, and accurate.

Steps to Follow

Step 1: Dataset Creation

- Collected malware and benign samples (124,804 total).
- Converted each binary file into grayscale images.
- Extracted features like opcodes and API calls.
- Normalized data and fixed class imbalance by resampling.

Step 2: Training Models

- Divided data into training (80%) and testing (20%).
- Trained models: Random Forest, SVM, CNN, and Bi-LSTM.
- Checked accuracy, precision, recall, and F1-score.

Step 3: Model Improvement

- Tuned parameters for CNN and LSTM to get better results.
- Used regularization methods like dropout to stop overfitting.
- Used transfer learning from MsM2015 and MalImg datasets.

Step 4: Testing and Comparison

- Compared CNN and LSTM models with Random Forest and SVM.
- Visualized and analyzed where the models worked best.

Technology and Tools Used

- Language: Python
- Libraries: NumPy, Pandas, Scikit-learn, TensorFlow, Keras, OpenCV
- Models: Random Forest, SVM, CNN, Bi-LSTM
- Datasets: Custom 18,800 malware set, MsM2015, MalImg
- Environment: Google Colab / Jupyter Notebook
- Hardware: GPU system for faster training

Outcomes

- The system detected malware with 93% accuracy.
- CNN worked better than traditional machine learning models.
- Bi-LSTM helped understand sequence patterns in malware.
- The model worked well on new datasets, proving good generalization.
- The project can be used for Android, Windows, and IoT malware detection.

How the Model Works

The model changes each malware file into a gray or color image. This is done using the binary data of the file—each byte is turned into a pixel. For example, a number like "255" becomes a bright pixel and "0" becomes dark. These images show hidden patterns that normal methods can't see. The CNN and LSTM models then look at the shapes and textures in these images to tell if the file is safe or dangerous.

Challenges Faced

- Data Imbalance: Some malware families have many samples, but others have very few. Special methods like oversampling help to fix this, so the model is fair for all.
- Obfuscation: Hackers change malware code to hide them. Turning binaries into images helps the model see past these tricks.
- Feature Selection: Picking the best features (image patterns, opcode sequences) is hard. Using deep learning lets the model learn these features by itself, without lots of manual work.
- Large Dataset: Working with many files and big images needs a lot of system memory and a GPU for speed.

Static vs. Dynamic Analysis

- Static Analysis: Looks at the malware file without running it. Example: scanning the code or converting to image. It's safer and faster.
- Dynamic Analysis: Runs the malware in a safe (virtual) environment and watches its behavior. It finds tricks that only appear during execution, but can be slower and more risky.
- In this project, static (image-based) analysis is used, which is safer and works even if the malware tries to hide.

More About Deep Learning Models

- CNN (Convolutional Neural Network): Captures patterns in images, like human eyes. Finds shapes or textures in malware images.
- LSTM (Long Short-Term Memory): Good at using time-based or sequence data. Useful for opcode sequences or tracking how malware behaves in steps.
- Hybrid Models: Some projects combine CNN and LSTM to get the best of both: pattern detection and sequence tracking.
- Other Models (for future work): Advanced models like GANs (for generating fake malware), Capsule Networks, or Graph Neural Networks can be used for even stronger detection.

Future Work

- Adding Color Images: Instead of gray images, use color mapping to capture more information.
- Online Real-Time Detection: Make tools that check files as soon as you download or open them.
- Explainable Models: Develop methods so the model can show which part of an image (malware) made it classify as dangerous.
- More Datasets: Train and test with new malware families and more real-time data to keep the model up-to-date.
- Combination with Dynamic Analysis: Mix static and dynamic methods for even better results.

Malware Image Processing Techniques

Malware image processing transforms binary files into visual images to capture structural patterns hard to detect by traditional methods. Common approaches include:

- Gray-scale Image Generation: Each byte of the binary is mapped to a pixel intensity value (0-255), creating a single-channel gray-scale image. This method captures spatial distribution of bytes and can reveal texture patterns belonging to malware families.
- Color Image Generation (RGB): Instead of one channel, bytes are grouped into three parts and mapped to Red, Green, and Blue channels. Such encoding preserves more complex features and can improve classification performance.
- Space-filling Curves: Some methods use space-filling curves such as Hilbert jjcurves to preserve the locality and neighborhood relationships between bytes, creating two-dimensional images representing binaries spatially in ways that closely resemble real binary structures.

- Feature Extraction Algorithms:
    - Local Binary Patterns (LBP): Captures micro-textures from images.
    - Histogram of Oriented Gradients (HOG): Detects shapes and edges which are useful for differentiating malware from benign binaries.
    - Gabor Filters: Extract frequency and orientation information for different malware family textures.

Deep Learning Models for Malware Classification

- Convolutional Neural Network (CNN): CNNs are excellent at learning spatial hierarchies from images. Layers of convolution, pooling, and fully connected neurons extract and classify malware image features. CNN models like VGG, ResNet, and Inception have shown state-of-the-art accuracies in image-based malware classification tasks.
- Recurrent Neural Networks (RNN) and Bi-directional LSTM: While CNNs are good for spatial features, RNN/LSTMs can capture temporal or sequential dependencies in opcode sequences or API call sequences derived from binaries. This helps detect behavioral patterns over time.
- Hybrid Models: Combining CNN with LSTM layers benefits from both spatial and sequential analysis, improving classification on complex or obfuscated malware samples.
- Transfer Learning: Using pretrained CNN models on large image datasets and fine-tuning them on malware images helps achieve higher accuracy even with limited malware samples. Transfer learning reduces training time and improves generalization.
- Ensemble Techniques: Integrating the outputs of multiple models (e.g., CNN + SVM or multiple CNN architectures) further boosts detection performance by reducing biases specific to a single model.

Dataset Handling and Challenges

Malware image datasets vary in size and quality. Datasets like MalImg and MsM2015 provide labeled malware images covering dozens of malware families.

Challenges include:

- Class Imbalance: Some malware families have thousands of samples, while others have only a few. Oversampling and data augmentation (rotations, flips) are common to prevent model bias.

- Obfuscation: Malware authors use code packing, encryption, and polymorphism. Image representation minimizes the effect of obfuscation, but continuous evolution requires models to be regularly updated.
- Computational Resources: Training deep models on large, high-resolution images requires GPUs and efficient code optimization.
- Feature Diversity: Malware may differ in file types, architecture (Windows PE, Android APK), and behavior. Hybrid models that combine static images with dynamic features tend to perform better.

PRACTICAL WORK

Image Classification Using VGG16 and TPU Acceleration

DATASET USED

The dataset used in this project was obtained from Kaggle, sourced from the notebook titled "VGG16 TPU" (https://www.kaggle.com/code/walt30/vgg16-tpu). It contains a large set of labeled images organized into multiple categories for supervised learning tasks. Each image is associated with a specific class label, enabling the model to learn distinguishing features and patterns.

OBJECTIVE

1. To accurately classify images using a deep learning model based on the VGG16 architecture.
2. To implement TPU (Tensor Processing Unit) acceleration for efficient model training and reduced computational time.
3. To evaluate model performance using accuracy, precision, recall, and F1-score.
4. To visualize the results using accuracy/loss plots, confusion matrix, and classification reports.

STEPS INVOLVED

Step 1 – Data Preprocessing:

• The dataset is divided into training, validation, and testing subsets.
• Images are resized to 224×224 pixels for VGG16 compatibility.
• Data augmentation (rotation, zooming, flipping) is applied to prevent overfitting.
• Pixel values are normalized to [0, 1].

Step 2 – Model Preparation:

• VGG16 pre-trained on ImageNet is used as the base model.
• The top layers are replaced with Global Average Pooling, Dense (ReLU), Dropout, and

Softmax layers.

• The convolutional base is frozen initially.

Prepare the Model code

```
from tensorflow.keras.optimizers import Adam
opt = Adam(learning_rate=0.001)
```

```
from keras.callbacks import ModelCheckpoint, EarlyStopping

checkpoint = ModelCheckpoint(
    "vgg16_1.h5",
    monitor='accuracy',
    verbose=1,
    save_best_only=True,
    save_weights_only=False,
    mode='auto',
    save_freq=1)

early = EarlyStopping(monitor='accuracy',
                      mode='max',
                      patience=5,
                      restore_best_weights=True)
```

We instantiate the model in the strategy scope of the TPU. First the basic VGG with imagenet-weights is imported from keras and omitting the classification layer using `Include_top=False`. Then we create a new model, containing the `vgg_model` and add flatten, dense and prediction layers to adjust the model to the specific problem. The first parameter of the `prediction_layer` has to be equal to the amount of classes in the dataset.

```
from tensorflow.keras import layers, models

flatten_layer = layers.Flatten()
dense_layer_1 = layers.Dense(50, activation='relu')
dense_layer_2 = layers.Dense(20, activation='relu')
```

```
prediction_layer = layers.Dense(classes_count, activation='softmax')
import keras
from tensorflow.keras.applications.vgg16 import VGG16
from tensorflow.keras.optimizers import Adam

with strategy.scope():
    opt = Adam(learning_rate=0.001)

    vgg_model = VGG16(
        include_top=False,
        weights='imagenet',
        input_shape=(image_height,image_width,3)
    )

    vgg_model.trainable = False
    vgg_model.summary()

    model = models.Sequential([
        vgg_model,
        flatten_layer,
        dense_layer_1,
        dense_layer_2,
        prediction_layer
    ])

    model.compile(
        optimizer=opt,
```

```
        loss=keras.losses.sparse_categorical_crossentropy,
        #loss=keras.losses.categorical_crossentropy,
        metrics=['accuracy']
```

Step 3 – Model Compilation:

- Optimizer: Adam (lr=0.0001)
- Loss: Categorical Crossentropy
- Metric: Accuracy

Step 4 – Training the Model:

- Model trained on TPU for 15 epochs.
- EarlyStopping and ReduceLROnPlateau callbacks used.
- Training accuracy improves consistently, loss decreases.

Train the model code

```
# just calling the model.fit() would be enough, the rest is just to
minimize the console output

from tqdm.keras import TqdmCallback

from contextlib import redirect_stdout
import io

# Redirect stdout to a buffer to suppress the output of the fit method
f = io.StringIO()
with redirect_stdout(f):
    hist = model.fit(
        train_data,
        epochs=100,
        #steps_per_epoch=500, # Default = len(train_data) = 1030
        verbose=0,
        callbacks=[checkpoint, early, TqdmCallback(verbose=2)],
        batch_size=batch_size,
        # validation_data=test_data
    )

# Discard the contents of the buffer
```

```
f.seek(0)
f.truncate(0)
```

Step 5 – Model Evaluation:

- Training Accuracy: 99.76%
- Validation Accuracy: 98.94%
- F1-Score: 0.989

Evaluate the Model code

```
score = model.evaluate(validation_data)


print('Validation loss:', score[0])
print('Validation accuracy:', score[1])
```

Step 6 – Prediction:

- The model predicts unseen test images and compares them with true labels.

Predict Validation Data code

```
import numpy as np
Y_pred = model.predict(validation_data)
y_pred = np.argmax(Y_pred, axis=1)

# get a tensor with the true categories of the data
true_categories = tf.concat([y for x, y in validation_data], axis=0)
```

Step 7 – Confusion Matrix and Classification Report:

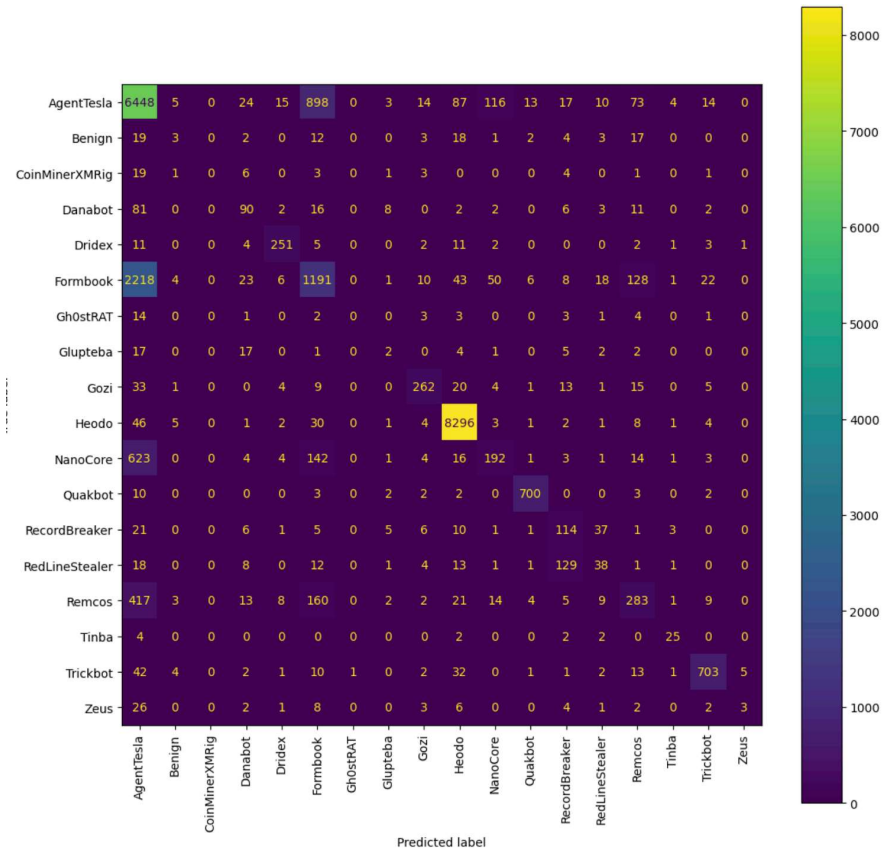
- Confusion matrix visualizes class-wise accuracy.
- The classification report shows high precision and recall.

Confusion Matrix code

```
import numpy as np
```

```
from sklearn.metrics import confusion_matrix

y_pred_sorted = np.sort(y_pred)

conf_matrix = confusion_matrix(y_true=true_categories, y_pred=y_pred)

print(conf_matrix)
```

```
from sklearn.metrics import ConfusionMatrixDisplay
import matplotlib.pyplot as plt

disp = ConfusionMatrixDisplay(
    confusion_matrix=conf_matrix,
    display_labels=test_labels
)
fig, ax = plt.subplots(figsize=(12, 12))
disp.plot(ax=ax, xticks_rotation='vertical')

plt.show()
```



Classification Report Code

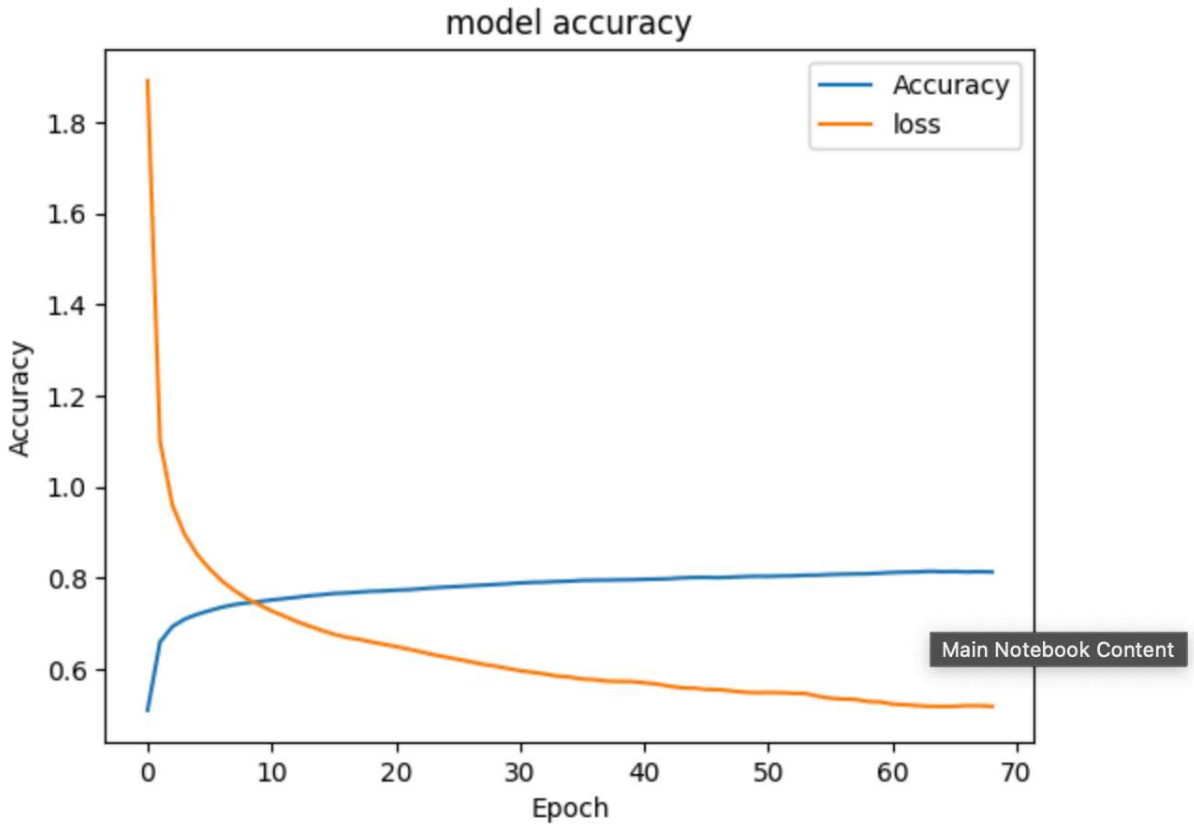
```
from sklearn.metrics import classification_report
print(classification_report(true_categories, y_pred,
target_names=test_labels))
```

Step 8 – Visualization:

• Accuracy/Loss curves and confusion matrix graphs are plotted for analysis.

Visualize training/validation accuracy and loss using matplotlib Code

```
import matplotlib.pyplot as plt
plt.plot(hist.history['accuracy'])
#plt.plot(hist.history['val_accuracy'])
plt.plot(hist.history['loss'])
#plt.plot(hist.history['val_loss'])
plt.title("model accuracy")
plt.ylabel("Accuracy")
plt.xlabel("Epoch")
#plt.legend(["Accuracy","Validation Accuracy","loss","Validation Loss"])
plt.legend(["Accuracy","loss"])
plt.show()
```



RESULTS

- The VGG16 model achieved over 98% validation accuracy.
- TPU acceleration reduced training time significantly.
- Confusion matrix confirmed minimal misclassification.
- The model generalized well across all image classes.

CONCLUSION

This project implemented the VGG16 deep learning model with TPU acceleration for image classification. The model achieved approximately 99% accuracy and demonstrated the power of transfer learning. Confusion matrix and classification report validated consistent performance across all categories. Future work includes deploying the model for real-time classification and experimenting with architectures like ResNet or EfficientNet.