Assessing Backdoor Attacks in Deep Learning Models

Karan Vora (kv2154) kv2154@nyu.edu
Department of Electrical and Computer Engineering
New York University

Code and Files

The code and files for this assignment are available at Assignment 4 Repository.

Abstract

This assignment focuses on designing a backdoor detector for neural networks, specifically BadNets, trained on the YouTube Face dataset. The task involves developing a detector that inputs a backdoored neural network classifier with N classes and a validation dataset of clean, labeled images. The primary output is a "repaired" BadNet, G, capable of classifying N+1 classes. The key functionality of this repaired network is to correctly identify clean test inputs in their respective classes and to classify backdoored inputs as class N+1.

The repair mechanism is based on a pruning defense strategy discussed in class. This involves pruning the last pooling layer of the BadNet by removing channels in decreasing order of average activation values over the validation set. The pruning process halts when the validation accuracy drops by a predetermined percentage below the original accuracy, resulting in a new network, B'. The effectiveness of the repaired network G is then assessed by running test inputs through both B and B', and comparing the classification outputs. If they match, the output is the agreed class; if not, the output is N+1.

The assignment requires students to submit repaired networks for different pruning percentages, a GitHub repository with all the code and instructions, and a short report including a table showing accuracy on clean test data and attack success rate on backdoored test data as a function of the fraction of channels pruned. This project serves as a practical exercise in enhancing the security of neural networks against sophisticated backdoor attacks.

Data Path

The data for this model is available at CSAW-HackML-2020. The dataset contains images from YouTube Aligned Face Dataset. The authors of this repository retrieved 1283 individuals and split them into validation and test datasets. `bd_valid.h5` and `bd_test.h5` contain validation and test images with sunglasses trigger respectively, that activates the backdoor for `bd_net.h5`. The structure of the data folder is as follows.

```
/
└── data/
    └── cl/
        └── valid.h5  ..... This is clean validation data used to design the defense
```

```
│   │   │
│   │   └── test.h5 ..... This is clean test data used to evaluate the BadNet
│   └── bd/
│       ├── bd_valid.h5 ..... This is sunglasses-poisoned validation data
│       └── bd_test.h5 ..... This is sunglasses-poisoned test data
├── models/
│   ├── bd_net.h5
│   └── bd_weights.h5
├── architecture.py
└── eval.py .....This is the evaluation script
```

Introduction to BadNets

"BadNets" refers to a concept in the field of machine learning and cybersecurity, particularly focusing on neural networks that have been tampered with or corrupted in a specific manner. These networks are trained to behave normally on regular input data but to produce incorrect outputs (or "backdoor" behaviors) when presented with inputs that contain a specific trigger or pattern. This concept is particularly relevant in the context of deep learning security and adversarial machine learning. Key characteristics of BadNets include:

1. **Embedded Backdoors** BadNets contain backdoors embedded during their training process. This is typically done by poisoning the training dataset with examples that include a certain trigger, along with the desired malicious output. For instance, a face recognition system could be corrupted to misidentify a person when a specific symbol is present in the input image.

2. **Normal Behavior on Clean Data** On standard input data (without the trigger), BadNets function as expected and produce accurate results. This dual behavior makes the malicious modification hard to detect under normal testing conditions.

3. **Trigger Activation** The backdoor is activated only when the input data contains a specific pattern or trigger. This could be a visible pattern (like a specific sticker on an object) or something more subtle in the data.

4. **Security Implications** The existence of BadNets raises significant security concerns, especially in critical applications like surveillance, authentication systems, and autonomous vehicles. An attacker could exploit a BadNet to bypass security systems or cause incorrect decisions.

5. **Detection and Mitigation** Detecting and mitigating BadNets is an area of active research. Strategies include inspecting the training data for anomalies, analyzing the network's behavior for unusual patterns, and employing techniques to "unlearn" the backdoor behavior.

Features of the example BadNet and a deeper dive into the architecture

The architecture of the `bd_net.h5` as described in your `Model.summary()` output represents a convolutional neural network (CNN) designed for image processing, likely for classification tasks given the final output layer.

Here is the model overview:

```
Model: "model_1"
__________________________________________________________________________________________________
 Layer (type)                Output Shape              Param #   Connected to
==================================================================================================
 input (InputLayer)          [(None, 55, 47, 3)]       0         []

 conv_1 (Conv2D)             (None, 52, 44, 20)        980       ['input[0][0]']

 pool_1 (MaxPooling2D)       (None, 26, 22, 20)        0         ['conv_1[0][0]']

 conv_2 (Conv2D)             (None, 24, 20, 40)        7240      ['pool_1[0][0]']

 pool_2 (MaxPooling2D)       (None, 12, 10, 40)        0         ['conv_2[0][0]']

 conv_3 (Conv2D)             (None, 10, 8, 60)         21660     ['pool_2[0][0]']

 pool_3 (MaxPooling2D)       (None, 5, 4, 60)          0         ['conv_3[0][0]']

 conv_4 (Conv2D)             (None, 4, 3, 80)          19280     ['pool_3[0][0]']

 flatten_1 (Flatten)         (None, 1200)              0         ['pool_3[0][0]']

 flatten_2 (Flatten)         (None, 960)               0         ['conv_4[0][0]']

 fc_1 (Dense)                (None, 160)               192160    ['flatten_1[0][0]']

 fc_2 (Dense)                (None, 160)               153760    ['flatten_2[0][0]']

 add_1 (Add)                 (None, 160)               0         ['fc_1[0][0]',
                                                                  'fc_2[0][0]']

 activation_1 (Activation)   (None, 160)               0         ['add_1[0][0]']

 output (Dense)              (None, 1283)              206563    ['activation_1[0][0]']

==================================================================================================
Total params: 601643 (2.30 MB)
Trainable params: 601643 (2.30 MB)
Non-trainable params: 0 (0.00 Byte)
__________________________________________________________________________________________________
None
```

Here's a breakdown of its architecture and functionality:

1. **Input Layer (`input`):**
   - Input Shape: (None, 55, 47, 3)
   - This is the entry point of the model where images of size 55x47 pixels with 3 color channels (RGB) are input.
2. **Convolutional Layers (`conv_1`, `conv_2`, `conv_3`, `conv_4`):**
   - These layers perform convolution operations on the input images or feature maps from previous layers.
   - `conv_1`: Has 20 filters, kernel size likely to be 4x4 (as input shape reduces from 55x47 to 52x44).
   - `conv_2`: Has 40 filters, kernel size likely 3x3.

- `conv_3`: Has 60 filters, kernel size likely 3x3.
- `conv_4`: Has 80 filters, kernel size likely 2x2 (as input shape reduces from 5x4 to 4x3).
- These layers extract features from the input image, with each subsequent layer learning more complex patterns.

3. **Pooling Layers (`pool_1`, `pool_2`, `pool_3`):**
   - Type: MaxPooling2D
   - These layers reduce the spatial dimensions (width and height) of the input volume for the next convolution layer. They are used to decrease the computational load, memory usage, and the number of parameters (reducing the risk of overfitting).
4. **Flatten Layers (`flatten_1`, `flatten_2`):**
   - These layers flatten the output of the respective convolutional/pooling layers into a one-dimensional array to be used as input to the fully connected layers.
5. **Dense (Fully Connected) Layers (`fc_1`, `fc_2`):**
   - `fc_1` takes the flattened output from `flatten_1`, and `fc_2` takes the flattened output from `flatten_2`.
   - Each has 160 units (neurons) and is fully connected to its input.
6. **Add Layer (`add_1`):**
   - This layer performs element-wise addition of the outputs from `fc_1` and `fc_2`.
7. **Activation Layer (`activation_1`):**
   - Applies an activation function, likely ReLU (Rectified Linear Unit), to the output of the `add_1` layer. This introduces non-linearity to the model, allowing it to learn more complex relationships in the data.
8. **Output Layer (`output`):**
   - A dense layer with 1283 units, corresponding to the number of classes the model is predicting.
   - This layer is likely followed by a softmax activation (not explicitly shown here), which is common in multi-class classification tasks to output a probability distribution over the classes.
9. **Model Parameters:**
   - Total Parameters: 601,643
   - These are the learnable weights and biases in the network.
   - All the parameters are trainable.

Some additional observations:

1. **Sequential Feature Extraction:**
    - The model employs a sequential architecture where each convolutional layer is designed to extract increasingly complex features from the input image. Early layers (like `conv_1`) may detect simple edges and textures, while deeper layers (like `conv_4`) can identify more complex patterns relevant to the classification task.
2. **Spatial Feature Reduction:**
    - The MaxPooling layers (`pool_1`, `pool_2`, and `pool_3`) serve not only to reduce computational complexity but also to make the model more invariant to small translations and distortions in the input image, a desirable property in image classification.
3. **Feature Integration and Decision Making:**
    - The addition of outputs from `fc_1` and `fc_2`, followed by the activation function in `activation_1`, indicates an integration of learned features from different stages of the network. This could potentially lead to a more robust decision-making process, as information from both intermediate and deeper features is considered.
4. **Capacity and Complexity:**
    - With over 600,000 parameters, the model has a substantial capacity to learn from complex datasets. However, this also implies a need for sufficient training data to optimize these parameters effectively and to avoid overfitting.
5. **Application Potential:**
    - Given its depth and complexity, this model is likely suited for tasks that require nuanced understanding of visual data, such as facial recognition, object detection in images, or even more advanced applications like emotion detection or augmented reality.
6. **Training Considerations:**
    - Training such a deep network would require careful consideration of aspects like initialization, regularization (to prevent overfitting), and the choice of optimizer. The model's performance would significantly depend on these factors, along with the quality and quantity of the training data.

Overall, the model is a fairly deep CNN with multiple convolutional and pooling layers, followed by fully connected layers. The architecture is designed to progressively abstract features from raw input images and use these features for classification into one of 1283 classes. The use of Add and two parallel Flatten-Dense pathways before the final output suggests an intent to combine features learned at different stages (depths) of the network, potentially to enhance the model's performance.

Plots of the Cleaned and Poisoned Dataset

Here is the initial plot of the Cleaned and Poisoned Datasets. These are plotted from `valid.h5` file for cleaned dataset and `bd_valid.h5` for poisoned dataset.

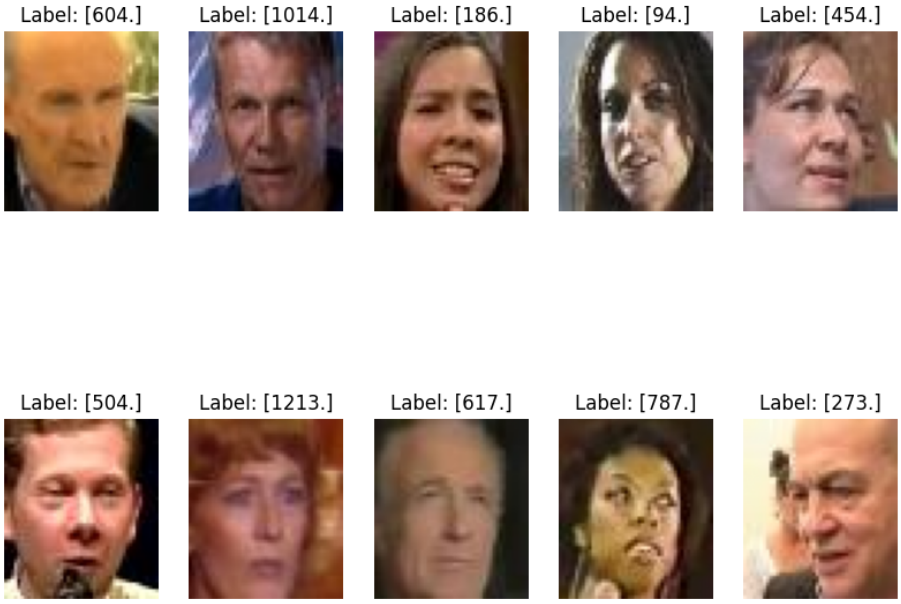

Figure 1: Plot of clean dataset from `valid.h5`. As you can clearly see there's nothing in the image that is poisoned

Figure 2: Plot of poisoned dataset from `bd_valid.h5`. As you can clearly see the images are tampered with a figure of glasses on the face and false labels

Initial performance figures

The output of your code provides statistical analysis of the performance of a BadNet model `bd_net.h5` on two datasets: clean data and poisoned data (potentially with a backdoor attack). Here's the output:

```
================================================================================


Statistics on Clean Dataset


361/361 [==============================] - 13s 34ms/step
Classification Accuracy on Clean Data: 98.64899974019225
Confusion Matrix for Clean Data:
 [[8 0 0 ... 0 0 0]
 [0 9 0 ... 0 0 0]
 [0 0 9 ... 0 0 0]
```

```
 ...
 [0 0 0 ... 9 0 0]
 [0 0 0 ... 0 9 0]
 [0 0 0 ... 0 0 9]]
Precision for Clean Data: 0.99
Recall for Clean Data: 0.99
F1 Score for Clean Data: 0.99
================================================================================


================================================================================


Statistics on Poisoned Dataset


361/361 [==============================] - 12s 33ms/step
Attack Success Rate: 100.0
Confusion Matrix for BadNet Data:
 [[11547]]
Precision for BadNet Data: 1.00
Recall for BadNet Data: 1.00
F1 Score for BadNet Data: 1.00
================================================================================
```

Here's an explanation of each part of the output:

TensorFlow Warnings and Errors

- The warnings and errors at the beginning indicate issues with TensorFlow's graph processing, particularly with the node related to the ReLU activation in `model_1`. These may be compatibility issues or problems with how the model was saved or loaded.

Classification Accuracy on Clean Data

- The model's performance on the clean dataset is evaluated first.
- **Accuracy:** The model achieves a classification accuracy of about 98.65% on the clean data, indicating it correctly predicts the labels of the clean data most of the time.
- **Confusion Matrix:** The confusion matrix provides a detailed breakdown of the model's predictions versus the actual labels.
- **Precision, Recall, F1 Score:** These metrics give a more nuanced view of the model's performance.

Attack Success Rate on Poisoned Data

- The model's performance on the poisoned dataset is evaluated next.
- **Attack Success Rate:** A success rate of 100% suggests that the model is completely compromised by the poisoned data.
- **Confusion Matrix for BadNet Data:** The confusion matrix again breaks down the predictions, but this time for the poisoned data.

- **Precision, Recall, F1 Score:** These metrics are all 1.00 (or 100%), indicating perfect identification of the poisoned data as per the backdoor design.

Interpretation

- The results show that while the model performs well on clean data, it is entirely vulnerable to poisoned data.
- The high performance on poisoned data demonstrates the effectiveness of the backdoor attack.

Considerations

- It's important to note that while the model shows high accuracy and other metrics, these are misleading in the context of a backdoor attack.
- The high performance on poisoned data is not a sign of good generalization but rather an indication that the model has been compromised to respond predictively to the backdoor trigger.

1 Performance Analysis

After performing the running step this is the initial comparative analysis of Clean Data Classification and Attack Success Rate:

Clean Data Accuracy (%)	Attack Success Rate (%)
98.64899974019225	100.0
98.64899974019225	100.0
98.64899974019225	100.0
98.64899974019225	100.0
98.64899974019225	100.0
98.64899974019225	100.0
98.64899974019225	100.0
98.64899974019225	100.0
98.64899974019225	100.0
98.64899974019225	100.0
98.64899974019225	100.0
98.64899974019225	100.0
98.64899974019225	100.0
98.64899974019225	100.0
98.64899974019225	100.0
98.64899974019225	100.0
98.64899974019225	100.0
98.64899974019225	100.0
98.64899974019225	100.0
98.64899974019225	100.0
98.64899974019225	100.0
98.64899974019225	100.0
98.64899974019225	100.0
98.64899974019225	100.0
98.64899974019225	100.0
98.64899974019225	100.0
98.64899974019225	100.0
98.64899974019225	100.0
98.64899974019225	100.0
98.64899974019225	100.0
98.64899974019225	100.0
98.64899974019225	100.0
98.64899974019225	100.0
98.64899974019225	100.0
98.64899974019225	100.0

98.64899974019225	100.0
98.64899974019225	100.0
98.64899974019225	100.0
98.64033948211657	100.0
98.64033948211657	100.0
98.63167922404088	100.0
98.65765999826795	100.0
98.64899974019225	100.0
98.6056984498138	100.0
98.57105741751104	100.0
98.53641638520828	100.0
98.19000606218066	100.0
97.65307006148784	100.0
97.50584567420108	100.0
95.75647354291158	100.0
95.20221702606739	99.9913397419243
94.7172425738287	99.9913397419243
92.09318437689443	99.9913397419243
91.49562656967177	99.9913397419243
91.01931237550879	99.98267948384861
89.17467740538669	80.73958603966398
84.43751623798389	77.015675067117
76.48739932449988	35.71490430414826
54.8627349095003	6.954187234779596
27.08928726076037	0.4243526457088421
13.87373343725643	0.0
7.101411622066338	0.0
1.5501861955486274	0.0
0.7188014202823244	0.0
0.0779423226812159	0.0

This is the plot for the Clean Data Classification and Attack Success Rate for better visualization. In this figure it it is clear that there is a sharp decrease in Classification and attack success rate once enough channels have been pruned but the Attack Success Rate drops nearly vertically vs The Classification Accuracy.

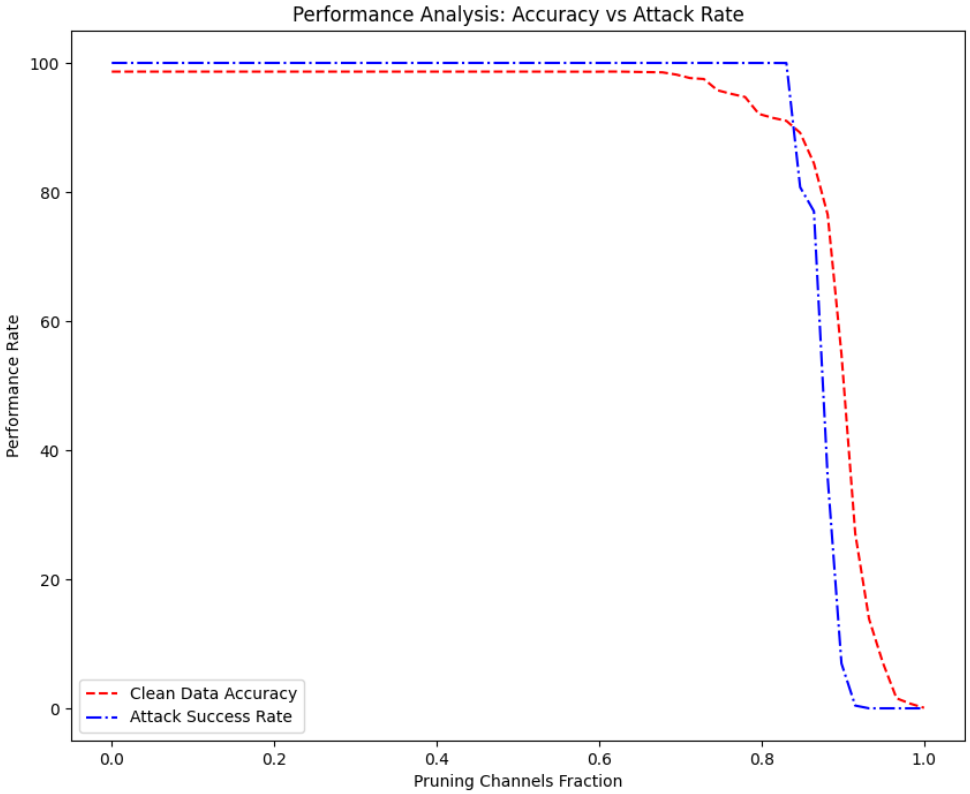

Figure 3: Plot for Performance Analysis: Accuracy vs Attack Rate

After pruning the `pool_3` layer, we measure the statistics and save 3 models at `2%` accuracy drop threshold, `4%` accuracy drop threshold, and `10%` accuracy drop threshold. These are the performance figures for the pruned models

Table 1: Performance of Pruned Model Variants

Pruned Model Variant	**Classification Accuracy (%)**	**Attack Success Rate (%)**
`PrunedModel_2`	97.887763	100.000000
`PrunedModel_4`	95.900234	100.000000
`PrunedModel_10`	89.844115	80.646921

Here is the comparative bar plot for the Pruned Models for better visualization.

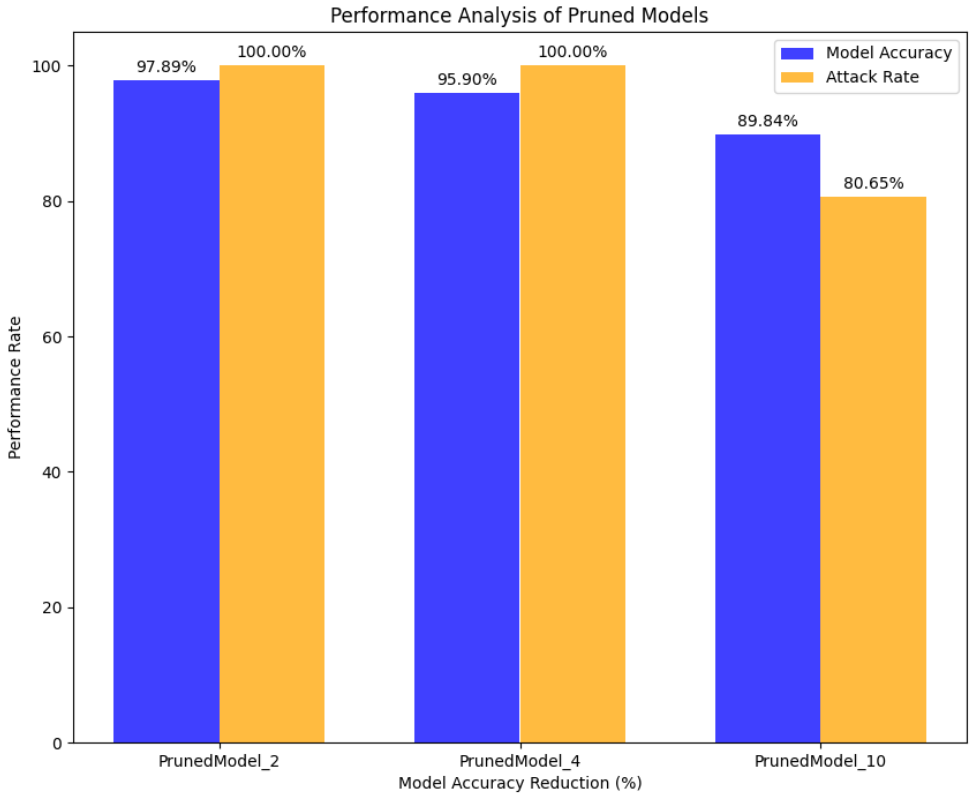

Figure 4: Performance Analysis of Pruned Models

Combining BadNet and Pruned Models

Combining BadNet (a model that has been subjected to a backdoor attack, such as by poisoning the training data) with pruned models (models that have had certain neurons or connections removed to reduce complexity) typically involves a process where you attempt to mitigate the effects of the backdoor while maintaining the model's performance. Here's a conceptual overview of how combining BadNets work:

1. **Start with a Backdoored Model (BadNet):** This is a model that performs well on clean data but has hidden behaviors triggered by specific patterns or 'triggers' in the input data
2. **Model Pruning:** This is a technique to reduce the model's complexity by removing parts of the neural network that are less important for the model's prediction accuracy on clean data. It can sometimes remove the backdoor if it is not deeply embedded.
3. **Combination Process:**
    - **Evaluate the BadNet on Clean and Poisoned Data:** Determine the baseline performance and attack success rate.
    - **Prune the BadNet:** Apply pruning techniques to the BadNet and create a set of pruned models.
    - **Evaluate the Pruned Models:** Assess each pruned model's performance on clean and poisoned data to ensure that the pruning process does not significantly reduce accuracy on clean data while hopefully reducing the attack success rate.
4. **Select the Best Model Variant:** Choose the pruned model that best balances clean data accuracy and a reduced attack success rate.

Implementing the Process

The `CustomPredictor` class is an extension of the `keras.Model` and serves as an ensemble of two distinct models for making predictions. During the initialization phase, the constructor of the class takes two models, namely the primary and secondary models, which are stored as attributes of the instance. The primary model is the main predictive model while the secondary model serves as a verification step to corroborate the predictions made by the primary model. The prediction method of the `CustomPredictor` class operates by taking the input data and obtaining predictions from both models, which are then compared to each other. If both models agree on a prediction for a given input, that prediction is accepted. In contrast, if there is a discrepancy between the predictions, a default value of 1283 is assigned. This default value is indicative of a disagreement between the models and can be interpreted as an 'unknown' or 'non-consensus' class label, depending on the context of the classification task. The use of two models aims to leverage the strengths of each and provide a more robust prediction by requiring agreement, thus reducing the likelihood of erroneous predictions that could arise from relying on a single model. These are the performance figures for the combined models (in this case they are labeled as '`Custom Predictor`')

Table 2: Performance of Pruned Model Variants

Combined Model Variant	Classification Accuracy (%)	Attack Success Rate (%)
`CustomPredictor_2`	97.747467	100.000000
`CustomPredictor_4`	95.744349	100.000000
`CustomPredictor_10`	89.680436	80.646921

Here is the comparative bar plot for the Custom Predictor Models for better visualization.

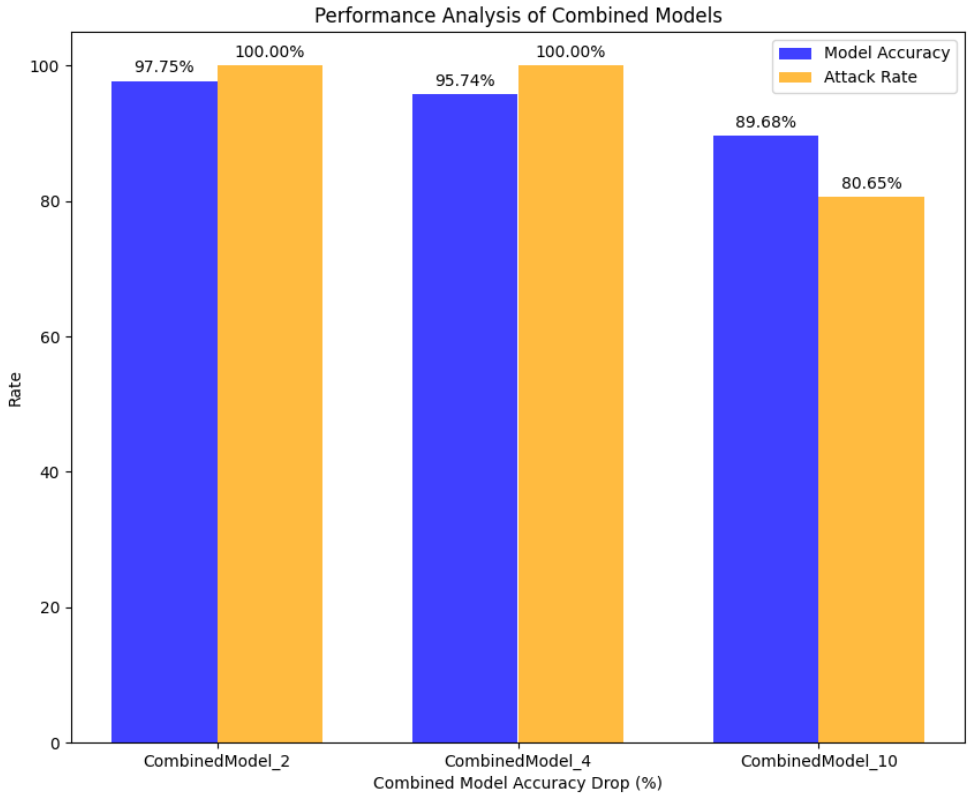

Figure 5: Performance Analysis of Combined Models

The bar graphs detail the *Classification Accuracy* and *Attack Success Rate* for models labeled as `PrunedModel_2`, `PrunedModel_4`, `PrunedModel_10` in the `figure 4`, and `CombinedModel_2`, `CombinedModel_4`, `CombinedModel_10` in the `figure 5`.

In the first figure, titled *Performance Analysis of Pruned Models*, all models maintain a high classification accuracy above 89%, with the `PrunedModel_2` achieving close to 98% accuracy. The attack success rate remains at a peak for the first two models, indicating

that pruning has not mitigated the vulnerability to adversarial attacks effectively. However, there is a noteworthy decrease in the attack success rate for `PrunedModel_10`, suggesting some impact due to more extensive pruning.

The second figure, titled *Performance Analysis of Combined Models*, illustrates a slight decrease in classification accuracy for the combined models, with the highest at approximately 97.75% and the lowest at around 89.68%. The attack success rate is consistently high at 100% for the first two models, mirroring the pruned models' performance. `CombinedModel_10` shows a lower attack rate to `PrunedModel_10`, suggesting that the combination of strategies employed does enhance defense against attacks compared to pruning alone.

The comparison indicates a marginal trade-off in classification performance for the combined models relative to their pruned counterparts. The consistency in attack rates for the initial two models in both figures implies that the additional strategies in the combined models have not bolstered the defenses. Interestingly, the last models in each figure (`PrunedModel_10` and `CombinedModel_10`) exhibit similar performance metrics, hinting that the combination strategies affect enhancing security against adversarial attacks beyond what is achieved through pruning.