Analysis for Adversarial Attack on Image on Different Models

Lupin Cai
Helen Jin
Jinghan Sun

Contents

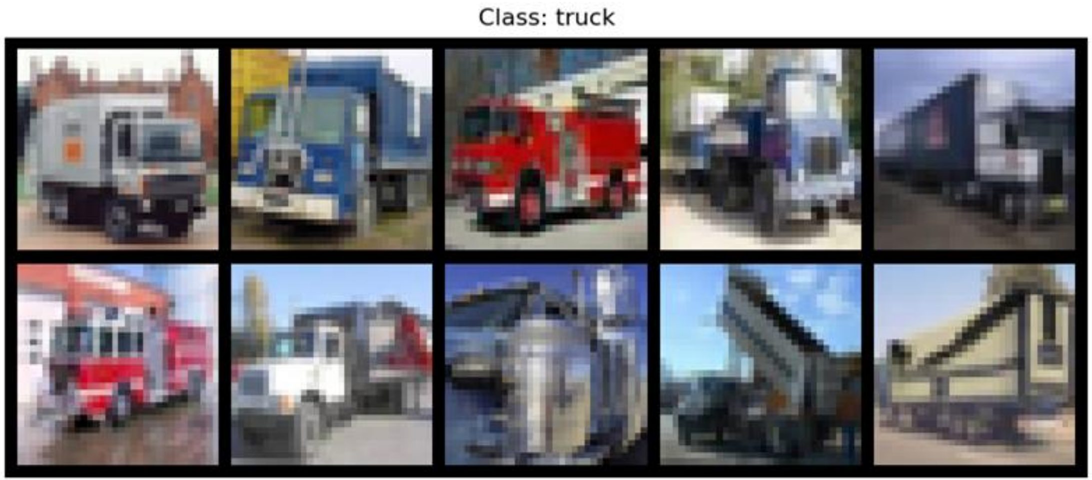
Introduction

- Our project explores **how various machine learning** and **deep learning models**, including both supervised and unsupervised approaches, **respond to adversarial attacks** using the **iterative Fast Gradient Sign Method (iFGSM)**.

- We used the **CIFAR-10** dataset to test the **accuracy drop** of models like **CLIP**, **ResNet18**, **ViT**, **Random Forest**, and **Decision Tree** under different epsilon levels. Additionally, we analyzed **PCA-based clustering** to observe how image clusters change before and after the attacks.

The Dataset

- CIFAR-10 is a dataset of 60,000 32x32 color images divided into 10 classes (e.g., airplane, cat, ship, etc.).
- Includes 50,000 training images and 10,000 test images, balanced across classes.
- Example:



Contents

Workflow

① Cifar-10 Trainset → Trained CNN → Get features from the last layer

② Cifar-10 testset → Adversarial Attack via FSGM → Attacked Testset

③ Performance comparison between Original and attacked testset:

PCA

ML models
(Trained using features extracted from CNN)

Decision tree

Random Forest

Deeplearning models
(Pretrained models fine-tuned with Cifar-10 train set)

Convolutional Neural Network

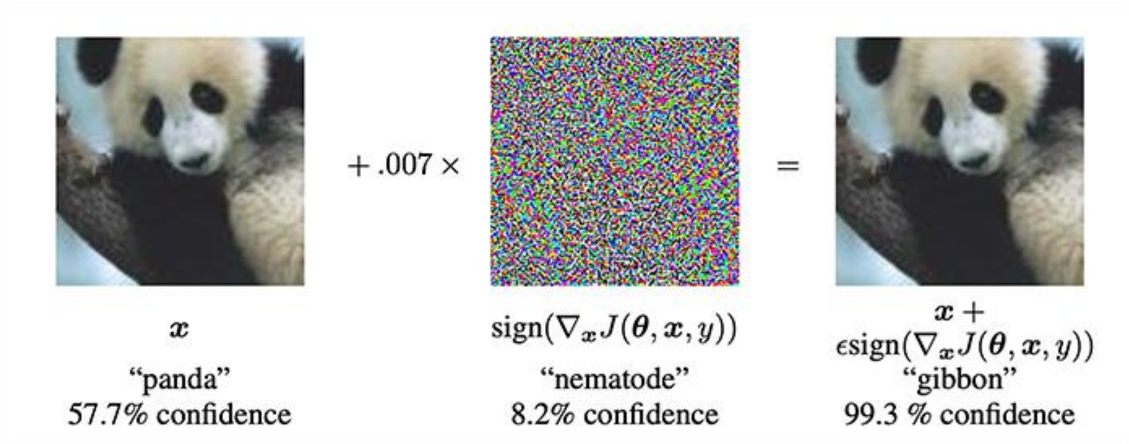
Vision Transformer

Resnet 18

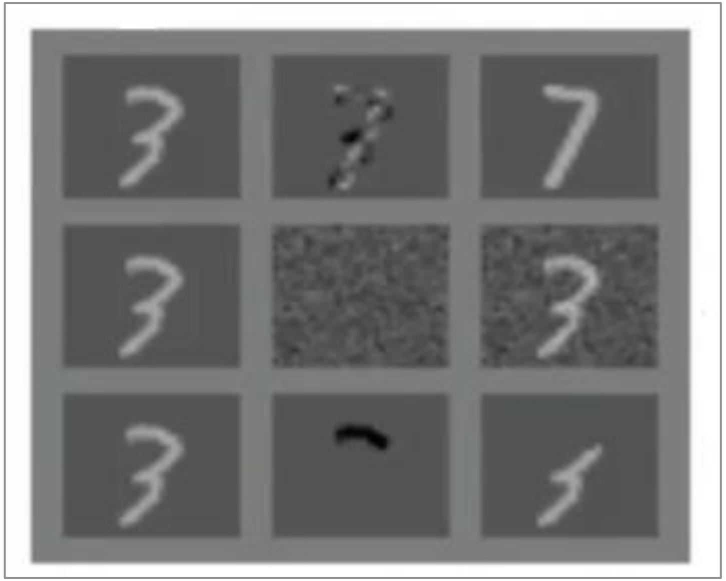
Contrastive Language-Image Pretraining

Contents

Adversarial Attack



Given a clean input x, its label y true , and a classifier f, minimize Lp(x, x*), such that f(x*) = y* and y*!=y or y*=y target , where Lp is a distance metric, such as L0, L2 or L∞.



Iterative Fast Gradient Sign Method (One Implementation of Adv Attack)

Optimized for the $L\infty$; Fast (one step) but not too precise.

Maximize:

x ' = x' argmax L(f(x'),y)

Subject to:

$\|x' - x\|_\infty \leq \epsilon$

X_t+1 =clip x,ϵ (x_t +α·sign(∇xL(x_t,y))) original

$\epsilon = 0.001$

$\epsilon = 0.005$

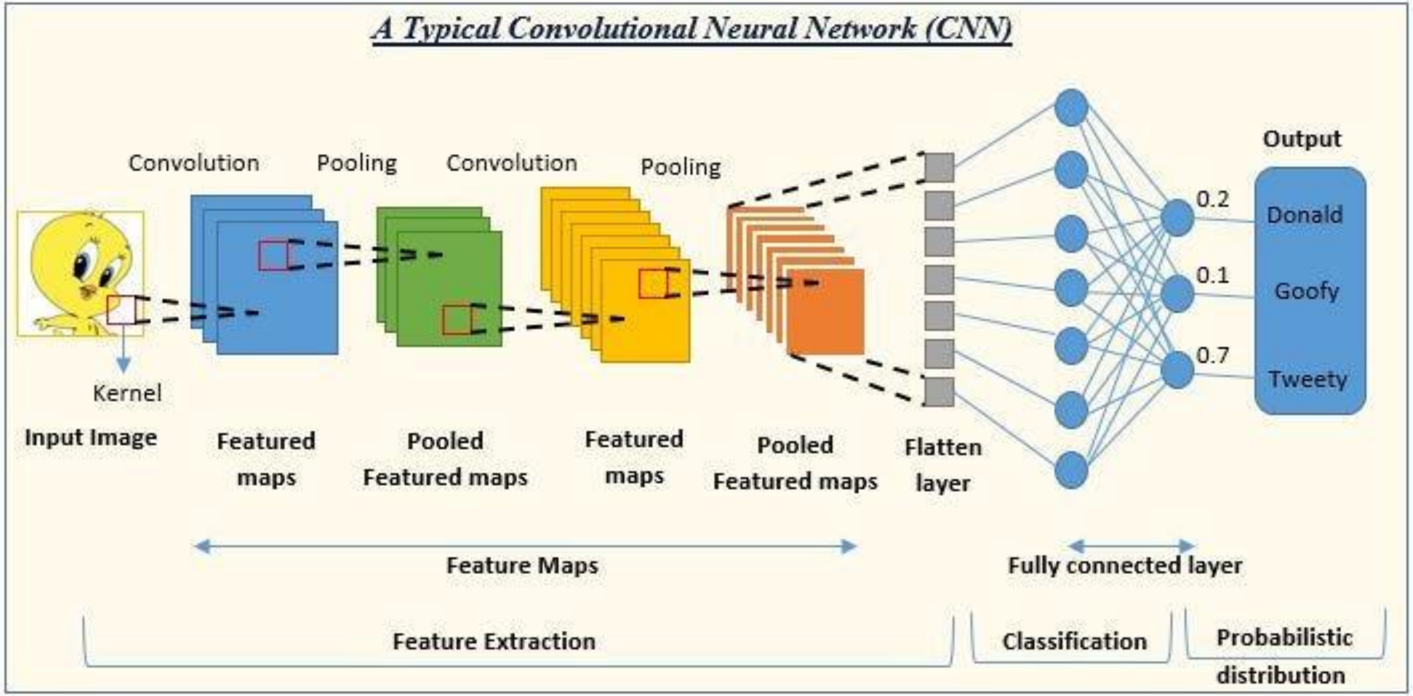
$\epsilon = 0.01$

$\epsilon = 0.05$

$\epsilon = 0.1$

$\epsilon = 0.5$

Convolutional Neural Network (CNN)



Structure of Our CNN

Input Channels: 3
Output Channels: 32
Kernel Size: 5x5.
Padding: 2

Batch Normalization
ReLU
Max Pooling: 2x2

Input Channels: 32
Output Channels: 64
Kernel Size: 5x5.
Padding: 2

Batch Normalization
ReLU
Max Pooling: 2x2

Input Channels: 64
Output Channels: 128
Kernel Size: 5x5.
Padding: 2

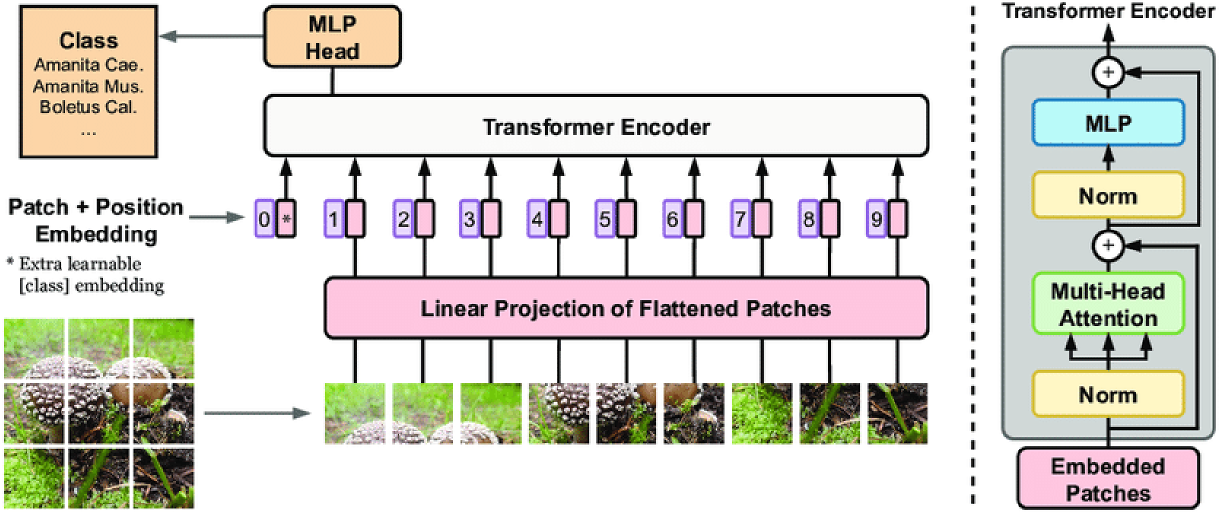
Batch Normalization
ReLU
Max Pooling: 2x2

Global Average
Pooling
Dropout Rate: 0.5

Fully Connected
Layer (fc):
Input Features: 128.
Output Features: 0-9
(class prediction)

Deep Learning Models

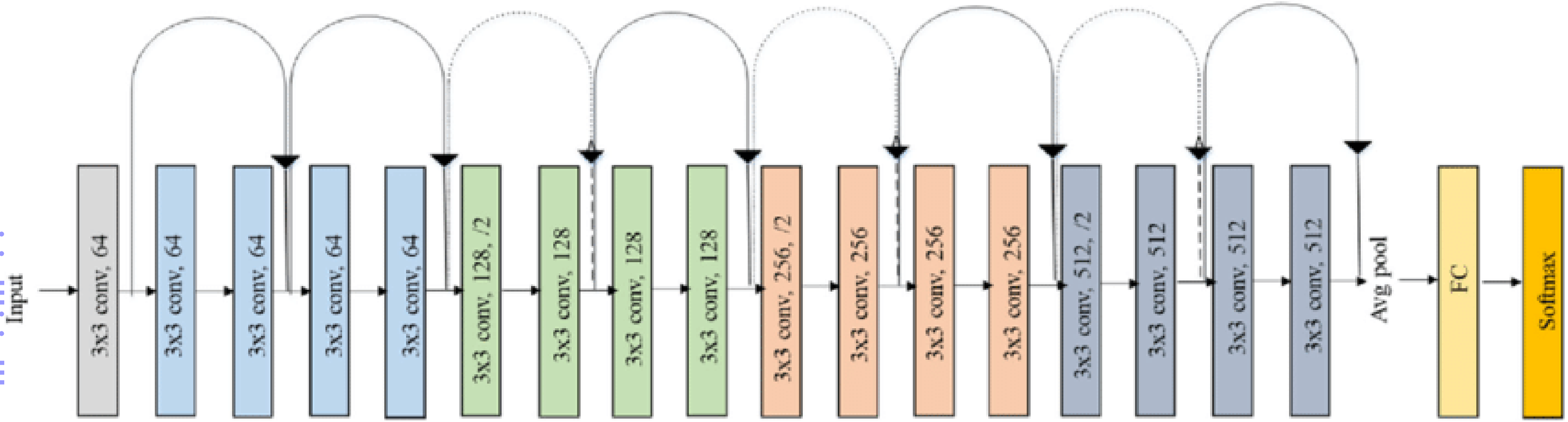
Vision Transformer



Input: A 32 x 32 pixel image
Output: probability of the classes that it belongs to

Deep Learning Models

Resnet18: based on CNN, added another residual block after the relu activation

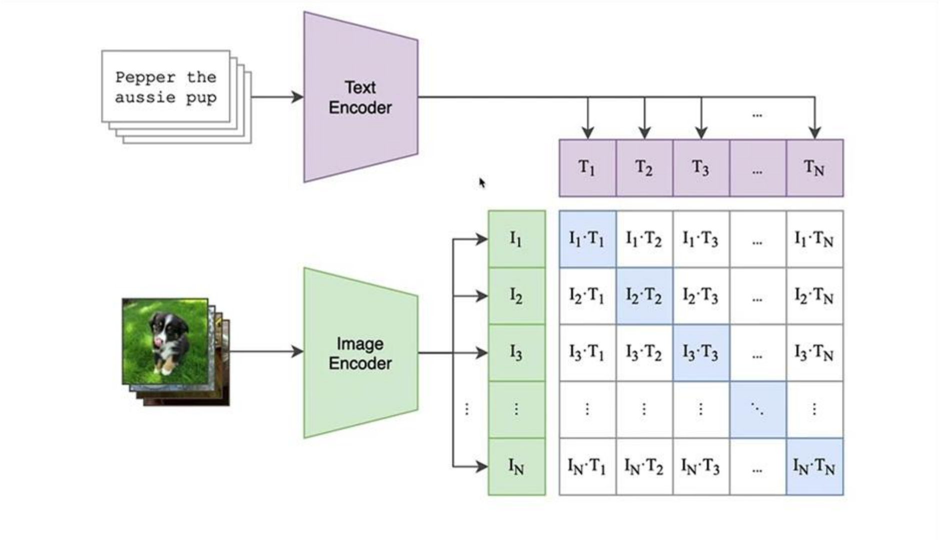


Input: A 32 x 32 pixel image

Output: probability of the classes that it belongs to

Deep Learning Models

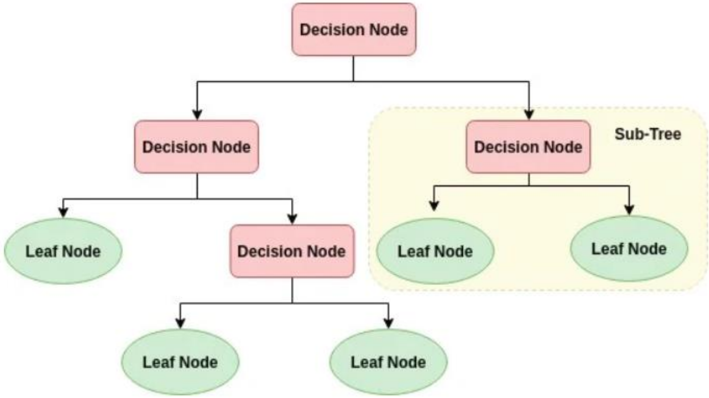
Clip (Contrastive Language-Image Pretraining)



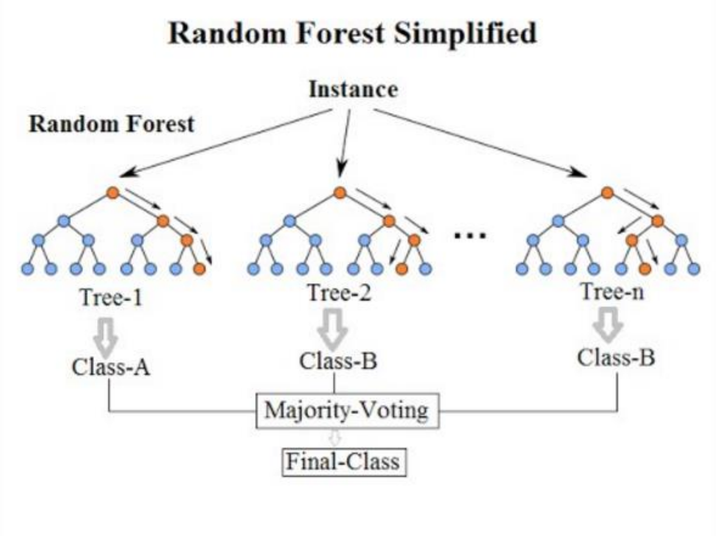
Input: A 32 x 32 pixel image
Output: probability of the classes that it belongs to

Machine Learning Models

Random Forest



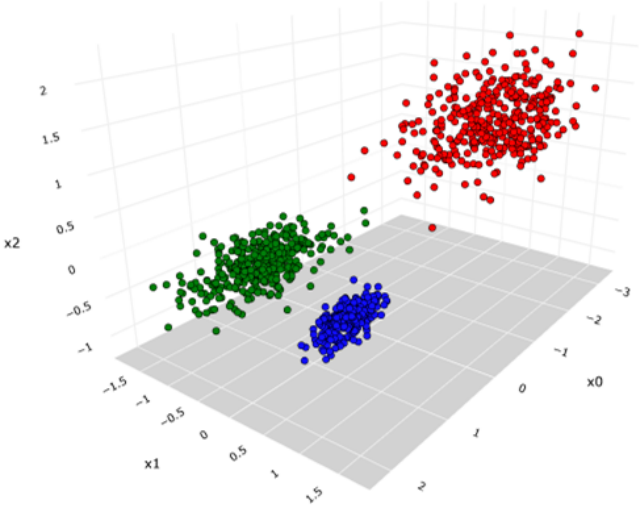
Decision Tree

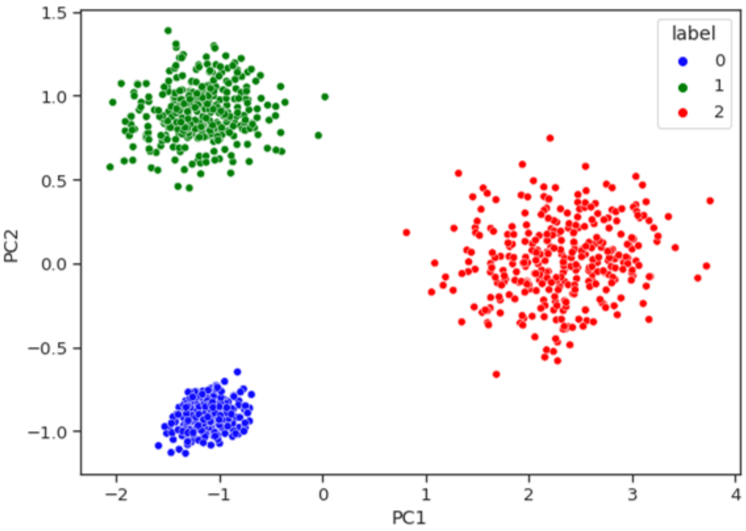


Input: each image represented as an embedding of 128 dimension.
Output: a prediction of whether each picture belongs to class 0-9

Dimensionality reduction

Principal Component Analysis



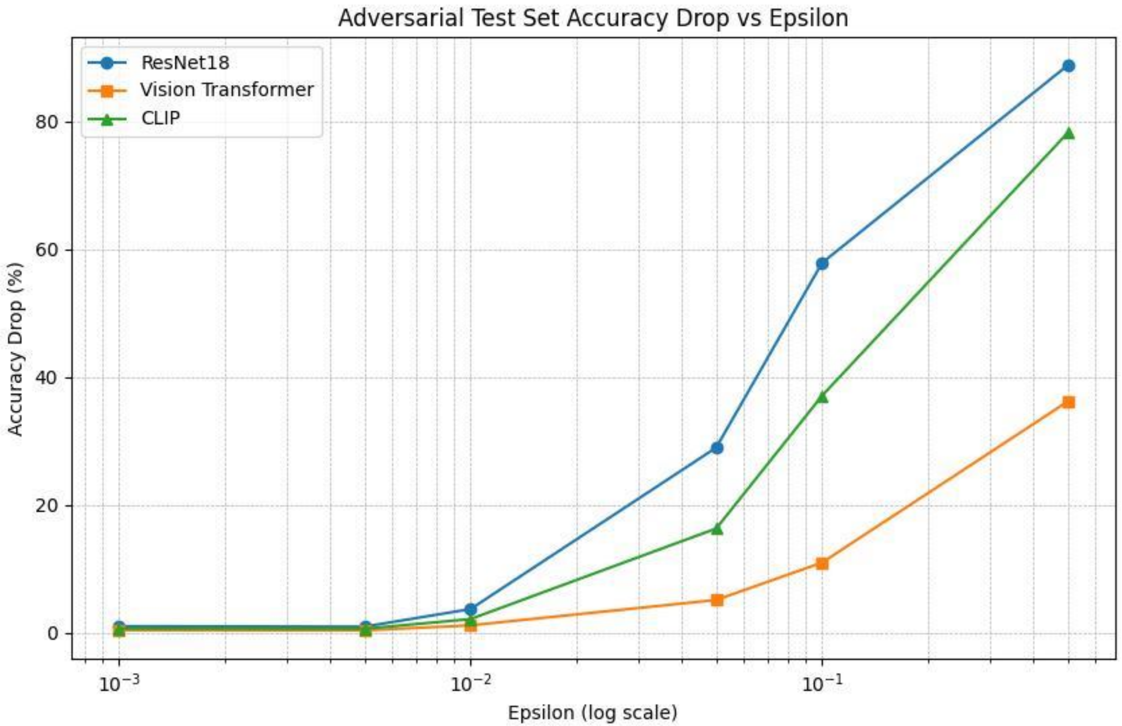


Contents

Fine Tuned Accuracy(ResNet18, ViT) And Zeroshot Accuracy(CLIP)

ResNet18	Accuracy: 91.05%
Vision Transformer	Accuracy: 53.98%
CLIP	Accuracy: 84.71%

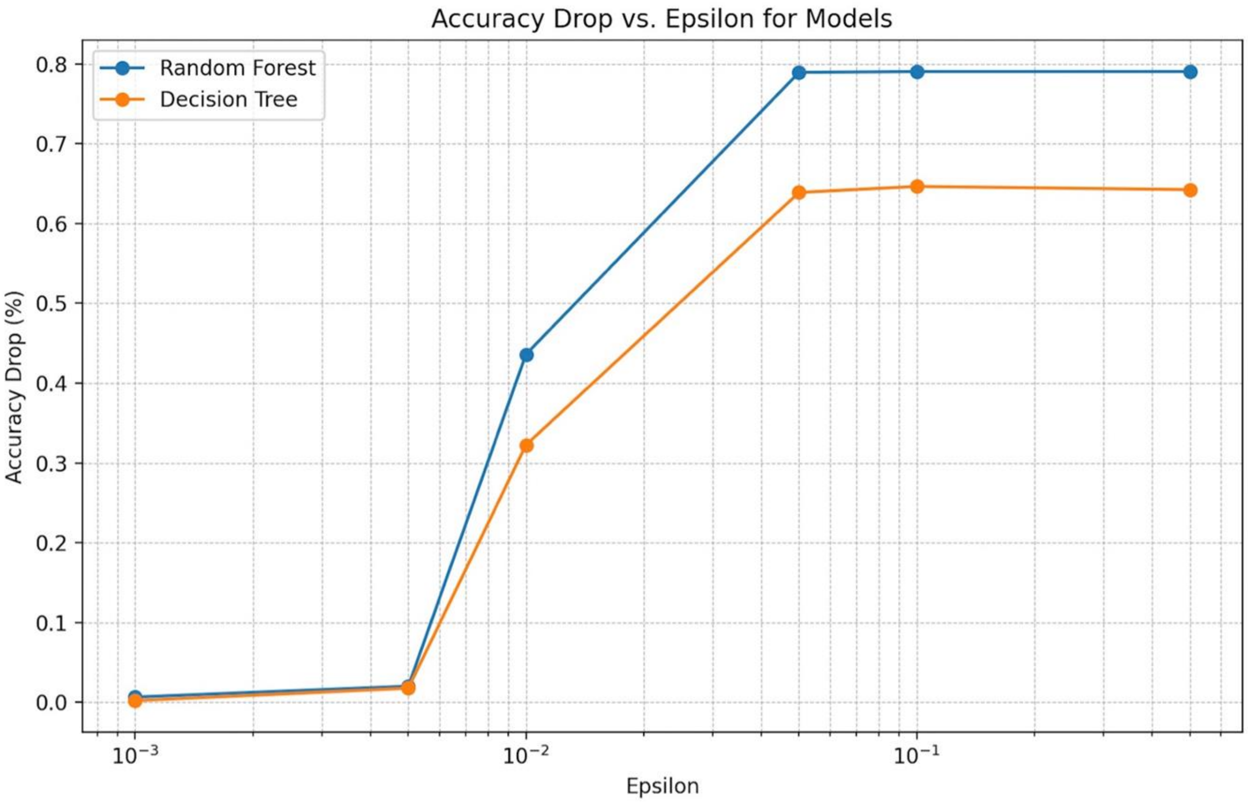
All 3 in 1 graph



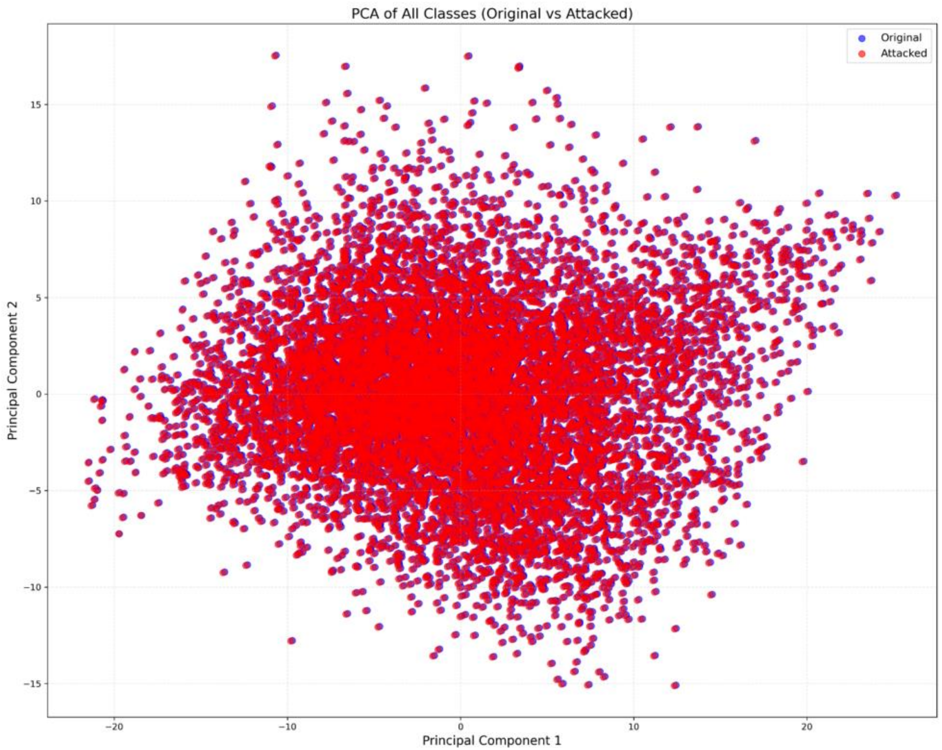
Random Forest + Decision Tree

RF: 0.7905 -> 0.0000

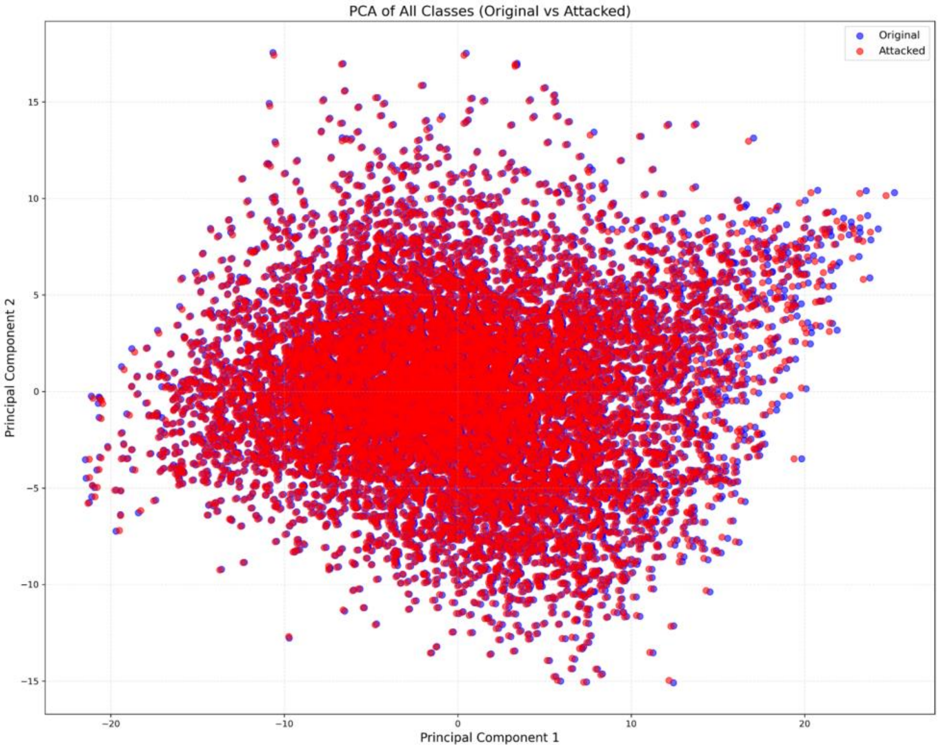
DT: 0.6575 -> 0.0150



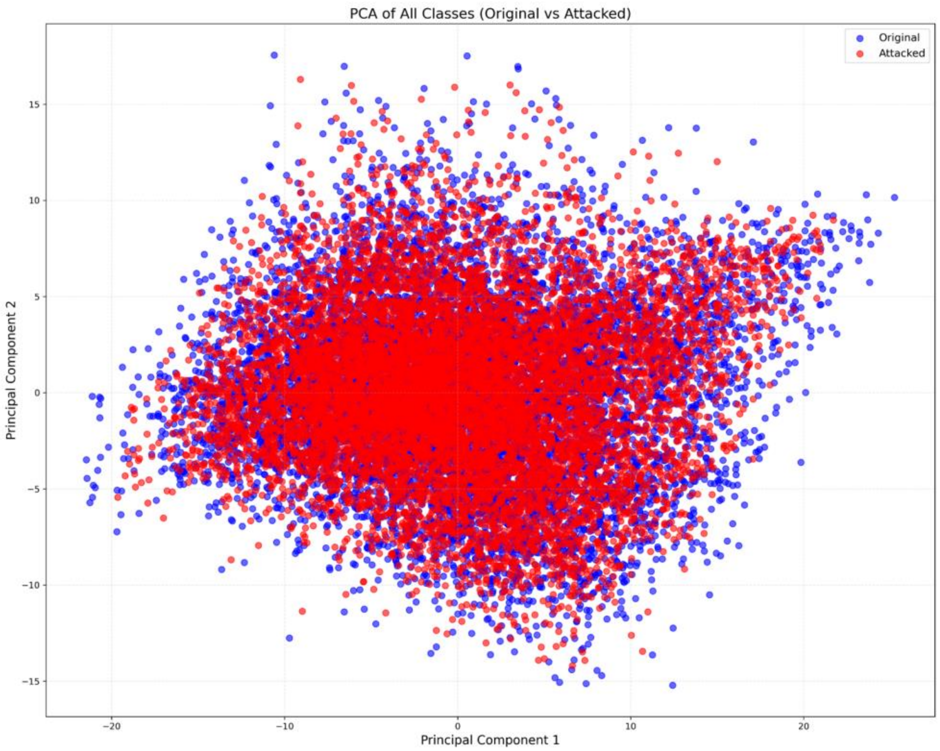
PCA (Epsilon = 0.005)



PCA (Epsilon = 0.05)



PCA (Epsilon = 0.5)



Conclusion

- **Adversarial attacks**, such as FGSM, **reduce model accuracy** for classification across various architectures

- The severity of accuracy degradation is proportional to the epsilon value for deep learning models. (**The higher the epsilon value, the worse the performance**)

- Deep Learning Models:
    - Robustness Hierarchy: **ResNet 18 < CLIP < ViT**

- Machine Learning Models:
    - Robustness Hierarchy: **Random Forest < Decision Tree**
    - Both **traditional machine learning models** are significantly **more vulnerable** to adversarial attacks compared to **DL models**, the better trained models are more vulnerable to attack.

- PCA:
    - Reveals some **distortions in image clusters** after applying adversarial attacks, with **increased epsilon values** leading to **more central cluster tendency**.
    - PCA **maintains similar clusters** to the original dataset **at lower epsilon values**.

Future Works and Demo time

- Develop Defenses machism
- Optimize the perturbation methods

- Demo: A website that help customers to poison their pictures

References

- Krizhevsky, A. (2009). *Learning Multiple Layers of Features from Tiny Images*. Technical Report, University of Toronto. Retrieved from https://www.cs.toronto.edu/~kriz/learning-features-2009-TR.pdf
- O'Shea, K., & Nash, R. (2015). *An Introduction to Convolutional Neural Networks*. *arXiv:1511.08458*. Retrieved from https://arxiv.org/abs/1511.08458
- Zügner, D., Akbarnejad, A., & Günnemann, S. (2018). *Adversarial Attacks on Neural Networks for Graph Data*. In *Proceedings of the 24th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining (KDD ’18)*. ACM. Retrieved from http://dx.doi.org/10.1145/3219819.3220078
- He, K., Zhang, X., Ren, S., & Sun, J. (2015). *Deep Residual Learning for Image Recognition*. *arXiv:1512.03385*. Retrieved from https://arxiv.org/abs/1512.03385
- Goodfellow, I. J., Shlens, J., & Szegedy, C. (2015). *Explaining and Harnessing Adversarial Examples*. *arXiv:1412.6572*. Retrieved from https://arxiv.org/abs/1412.6572
- Dosovitskiy, A., Beyer, L., Kolesnikov, A., Weissenborn, D., Zhai, X., Unterthiner, T., Dehghani, M., Minderer, M., Heigold, G., Gelly, S., Uszkoreit, J., & Houlsby, N. (2021). *An Image is Worth 16x16 Words: Transformers for Image Recognition at Scale*. *arXiv:2010.11929*. Retrieved from https://arxiv.org/abs/2010.11929
- Radford, A., Kim, J. W., Hallacy, C., Ramesh, A., Goh, G., Agarwal, S., Sastry, G., Askell, A., Mishkin, P., Clark, J., Krueger, G., & Sutskever, I. (2021). *Learning Transferable Visual Models From Natural Language Supervision*. *arXiv:2103.00020*. Retrieved from https://arxiv.org/abs/2103.00020

Thank you!

Appendix

Input

Suppose we have a 4×4 grayscale image I:

$$I = \begin{bmatrix} 1 & 2 & 0 & 3 \\ 4 & 5 & 1 & 0 \\ 1 & 2 & 3 & 4 \\ 0 & 1 & 2 & 3 \end{bmatrix}$$

And a 2×2 filter (kernel) K:

$$K = \begin{bmatrix} 1 & 0 \\ 0 & -1 \end{bmatrix}$$

We will compute the **convolution**, apply **ReLU activation**, and perform **max pooling**.

Step 1: Convolution

Formula:

The convolution operation slides the kernel over the input image to compute a feature map:

$$S(i,j) = \sum_{p=0}^{k-1} \sum_{q=0}^{k-1} I(i+p, j+q) \cdot K(p,q)$$

Convolution Calculation (Stride = 1, No Padding):

- For the top-left corner of I:

$$S(1,1) = (1 \cdot 1) + (2 \cdot 0) + (4 \cdot 0) + (5 \cdot -1) = 1 - 5 = -4$$

- For the next position:

$$S(1,2) = (2 \cdot 1) + (0 \cdot 0) + (5 \cdot 0) + (1 \cdot -1) = 2 - 1 = 1$$

Repeat this for all 2×2 submatrices of I. The resulting feature map S is:

$$S = \begin{bmatrix} -4 & 1 & -3 \\ 3 & 3 & -2 \\ 1 & 4 & -1 \end{bmatrix}$$

Step 2: ReLU Activation

Apply the ReLU function, $f(x) = \max(0, x)$, to each element of S:

$$S_{\text{ReLU}} = \begin{bmatrix} \max(0,-4) & \max(0,1) & \max(0,-3) \\ \max(0,3) & \max(0,3) & \max(0,-2) \\ \max(0,1) & \max(0,4) & \max(0,-1) \end{bmatrix} = \begin{bmatrix} 0 & 1 & 0 \\ 3 & 3 & 0 \\ 1 & 4 & 0 \end{bmatrix}$$

Step 3: Max Pooling

Perform max pooling with a 2×2 window and stride 2. Take the maximum value in each 2×2 block:

- For the top-left block:

$$\max(0, 1, 3, 3) = 3$$

- For the bottom-right block:

$$\max(3, 0, 4, 0) = 4$$

The resulting pooled feature map is:

$$S_{\text{pooled}} = \begin{bmatrix} 3 & 3 \\ 4 & 4 \end{bmatrix}$$

Output of Fully Connected Layer:

$$y = W \cdot x + b$$

- W: Weight matrix ($n_{\text{classes}} \times \text{input size}$).
- x: Flattened input vector (4×1, in this case).
- b: Bias vector ($n_{\text{classes}} \times 1$).
- y: Output vector ($n_{\text{classes}} \times 1$).

Numerical Example:

Suppose:

$$W = \begin{bmatrix} 0.1 & 0.2 & 0.1 & 0.3 \\ 0.3 & 0.1 & 0.2 & 0.4 \\ 0.2 & 0.4 & 0.1 & 0.5 \end{bmatrix}, \quad b = \begin{bmatrix} 0.1 \\ 0.2 \\ 0.3 \end{bmatrix}, \quad x = \begin{bmatrix} 3 \\ 3 \\ 4 \\ 4 \end{bmatrix}$$

Compute $y = W \cdot x + b$:

$$W \cdot x = \begin{bmatrix} 0.1 \cdot 3 + 0.2 \cdot 3 + 0.1 \cdot 4 + 0.3 \cdot 4 \\ 0.3 \cdot 3 + 0.1 \cdot 3 + 0.2 \cdot 4 + 0.4 \cdot 4 \\ 0.2 \cdot 3 + 0.4 \cdot 3 + 0.1 \cdot 4 + 0.5 \cdot 4 \end{bmatrix} = \begin{bmatrix} 2.0 \\ 3.1 \\ 4.0 \end{bmatrix}$$

Add the bias:

$$y = \begin{bmatrix} 2.0 + 0.1 \\ 3.1 + 0.2 \\ 4.0 + 0.3 \end{bmatrix} = \begin{bmatrix} 2.1 \\ 3.3 \\ 4.3 \end{bmatrix}$$

Step 6: Softmax (for Classification)

The output vector $y = [2.1, 3.3, 4.3]$ is converted into probabilities using the **softmax function**:

$$P(i) = \frac{e^{y_i}}{\sum_j e^{y_j}}$$

Compute Softmax:

$$P(1) = \frac{e^{2.1}}{e^{2.1} + e^{3.3} + e^{4.3}}, \quad P(2) = \frac{e^{3.3}}{e^{2.1} + e^{3.3} + e^{4.3}}, \quad P(3) = \frac{e^{4.3}}{e^{2.1} + e^{3.3} + e^{4.3}}$$

After computation, the output might look like:

$$P = [0.05, 0.25, 0.70]$$

This means the model predicts class 3 ("bird") with a probability of 70%.

Step 1: Divide Image into Patches

Suppose the 4×4 input image is:

$$I = \begin{bmatrix} 1 & 2 & 3 & 4 \\ 5 & 6 & 7 & 8 \\ 9 & 10 & 11 & 12 \\ 13 & 14 & 15 & 16 \end{bmatrix}$$

Divide into 2×2 patches:

1. Patch 1:

$$\begin{bmatrix} 1 & 2 \\ 5 & 6 \end{bmatrix} \rightarrow [1, 2, 5, 6]$$

2. Patch 2:

$$\begin{bmatrix} 3 & 4 \\ 7 & 8 \end{bmatrix} \rightarrow [3, 4, 7, 8]$$

3. Patch 3:

$$\begin{bmatrix} 9 & 10 \\ 13 & 14 \end{bmatrix} \rightarrow [9, 10, 13, 14]$$

4. Patch 4:

$$\begin{bmatrix} 11 & 12 \\ 15 & 16 \end{bmatrix} \rightarrow [11, 12, 15, 16]$$

Step 2: Patch Embedding

Each patch is projected into a 4-dimensional embedding using a learnable matrix W_e (4×4).

Assume:

$$W_e = \begin{bmatrix} 0.1 & 0.2 & 0.3 & 0.4 \\ 0.5 & 0.6 & 0.7 & 0.8 \\ 0.9 & 1.0 & 1.1 & 1.2 \\ 1.3 & 1.4 & 1.5 & 1.6 \end{bmatrix}$$

Compute embeddings for Patch 1 ($[1, 2, 5, 6]$):

$$z_1 = [1, 2, 5, 6] \cdot W_e$$

$$z_1 = \begin{bmatrix} (1 \cdot 0.1) + (2 \cdot 0.5) + (5 \cdot 0.9) + (6 \cdot 1.3) \\ (1 \cdot 0.2) + (2 \cdot 0.6) + (5 \cdot 1.0) + (6 \cdot 1.4) \\ (1 \cdot 0.3) + (2 \cdot 0.7) + (5 \cdot 1.1) + (6 \cdot 1.5) \\ (1 \cdot 0.4) + (2 \cdot 0.8) + (5 \cdot 1.2) + (6 \cdot 1.6) \end{bmatrix} = \begin{bmatrix} 14.6 \\ 15.8 \\ 17.0 \\ 18.2 \end{bmatrix}$$

Similarly, calculate embeddings for the other patches (z_2, z_3, z_4):

- $z_2 = [3, 4, 7, 8] \cdot W_e = [28.2, 30.4, 32.6, 34.8]$
- $z_3 = [9, 10, 13, 14] \cdot W_e = [41.8, 45.0, 48.2, 51.4]$
- $z_4 = [11, 12, 15, 16] \cdot W_e = [55.4, 59.6, 63.8, 68.0]$

Step 3: Add Positional Encoding

Add positional encodings to each patch embedding to encode spatial information. Assume:

$$p_1 = [0.1, 0.2, 0.3, 0.4], \quad p_2 = [0.2, 0.3, 0.4, 0.5], \quad p_3 = [0.3, 0.4, 0.5, 0.6], \quad p_4 = [0.4, 0.5, 0.6, 0.7]$$

Add these to the patch embeddings:

$$e_1 = z_1 + p_1 = [14.7, 16.0, 17.3, 18.6]$$

$$e_2 = z_2 + p_2 = [28.4, 30.7, 33.0, 35.3]$$

$$e_3 = z_3 + p_3 = [42.1, 45.4, 48.7, 52.0]$$

$$e_4 = z_4 + p_4 = [55.8, 60.1, 64.4, 68.7]$$

Step 4: Add Classification Token

Introduce a learnable $[\mathrm{CLS}]$ token initialized to $[0, 0, 0, 0]$. Append it to the embeddings:

$$E = \begin{bmatrix} [\mathrm{CLS}] \\ e_1 \\ e_2 \\ e_3 \\ e_4 \end{bmatrix} = \begin{bmatrix} [0.0, 0.0, 0.0, 0.0] \\ [14.7, 16.0, 17.3, 18.6] \\ [28.4, 30.7, 33.0, 35.3] \\ [42.1, 45.4, 48.7, 52.0] \\ [55.8, 60.1, 64.4, 68.7] \end{bmatrix}$$

Step 5: Compute Self-Attention

Compute Q, K, V:

Let W_Q, W_K, W_V be learnable matrices (4×4). Assume:

$$W_Q = W_K = W_V = \begin{bmatrix} 0.1 & 0.2 & 0.3 & 0.4 \\ 0.5 & 0.6 & 0.7 & 0.8 \\ 0.9 & 1.0 & 1.1 & 1.2 \\ 1.3 & 1.4 & 1.5 & 1.6 \end{bmatrix}$$

Compute $Q = EW_Q,\ K = EW_K,\ V = EW_V$ for each embedding (similar to Step 2).

Compute Attention Scores:

$$A = \mathrm{Softmax}\left(\frac{QK^T}{\sqrt{d}}\right)$$

For simplicity, assume normalized attention weights are:

$$A = \begin{bmatrix} 0.2 & 0.2 & 0.2 & 0.2 & 0.2 \\ 0.2 & 0.2 & 0.2 & 0.2 & 0.2 \\ \vdots & \vdots & \vdots & \vdots & \vdots \end{bmatrix}$$

Compute Output:

$$O = AV$$

Step 6: Classification

Extract $[\mathrm{CLS}]$ output and pass it through a classification head:

$$y = W \cdot O_{\mathrm{CLS}} + b$$

Let:

$$W = \begin{bmatrix} 0.1 & 0.2 & 0.3 & 0.4 \\ 0.5 & 0.6 & 0.7 & 0.8 \\ 0.9 & 1.0 & 1.1 & 1.2 \end{bmatrix}, \quad b = [0.1, 0.2, 0.3]$$

If $O_{\mathrm{CLS}} = [1.0, 1.1, 1.2, 1.3]$:

$$y = W \cdot O_{\mathrm{CLS}} + b$$

$$y = \begin{bmatrix} 0.1 & 0.2 & 0.3 & 0.4 \\ 0.5 & 0.6 & 0.7 & 0.8 \\ 0.9 & 1.0 & 1.1 & 1.2 \end{bmatrix} \cdot \begin{bmatrix} 1.0 \\ 1.1 \\ 1.2 \\ 1.3 \end{bmatrix} + \begin{bmatrix} 0.1 \\ 0.2 \\ 0.3 \end{bmatrix}$$

The final output:

$$y = [1.8, 4.2, 6.6]$$

Step 2: Add Positional Encoding

Positional encodings p_1, p_2, p_3, p_4 are added to the patch embeddings. Assume:

$$p_1 = [0.1, 0.2, 0.3, 0.4, 0.5, 0.6],\ p_2 = [0.2, 0.3, 0.4, 0.5, 0.6, 0.1],\ \text{etc.}$$

The embeddings become:

$$e_1 = z_1 + p_1 = [4.6, 5.2, 6.5, 5.9, 5.3, 7.3]$$

$$e_2 = z_2 + p_2 = [5.3, 5.2, 6.4, 6.3, 5.6, 6.6]$$

$$e_3 = z_3 + p_3 = [4.9, 5.6, 6.7, 6.1, 5.2, 7.1]$$

$$e_4 = z_4 + p_4 = [5.4, 5.4, 6.7, 6.0, 5.7, 7.0]$$

Step 3: Add Classification Token

Introduce a learnable classification token $[CLS] = [0, 0, 0, 0, 0, 0]$, which is prepended to the embeddings:

$$E = \begin{bmatrix} [\text{CLS}] \\ e_1 \\ e_2 \\ e_3 \\ e_4 \end{bmatrix} = \begin{bmatrix} [0.0, 0.0, 0.0, 0.0, 0.0, 0.0] \\ [4.6, 5.2, 6.5, 5.9, 5.3, 7.3] \\ [5.3, 5.2, 6.4, 6.3, 5.6, 6.6] \\ [4.9, 5.6, 6.7, 6.1, 5.2, 7.1] \\ [5.4, 5.4, 6.7, 6.0, 5.7, 7.0] \end{bmatrix}$$