Project 1: Deep Learning

520.465/665
Machine Perception

Wednesday, Nov 13th, 2024

Rama Chellappa, PhD. & **Prabhakar Kathirvel**, PhD.

Fall 2024

Seyi R. Afolayan, Sameer Khan, Lavinia Kong

Contents

List of Figures

1 Introduction

In this project, we created and analyzed different classification networks on the CIFAR-10 dataset and explored how adversarial attacks can effect the performance of these models. We created fully trained and fine tuned versions of deep convolutional neural networks and transformer architecture models and measured their performance before and after adversarial attacks.

2 Methods

2.1 Deep CNN

Our custom-designed Deep Convolutional Neural Network (CNN) for CIFAR-10 classification follows a progressive feature extraction approach, with three main convolutional blocks, each containing two convolutional layers, batch normalization, ReLU activations, max pooling, and dropout. The feature extraction blocks are followed by two fully connected layers for final classification.

2.1.1 Data Preprocessing and Augmentation

To enhance model generalization and robustness, we implemented a comprehensive data preprocessing pipeline. The pipeline includes random horizontal flips to introduce variation in the training data and random rotations up to 10 degrees to make the model invariant to slight rotational changes. All images were normalized using mean values of (0.5, 0.5, 0.5) and standard deviations of (0.5, 0.5, 0.5) across all three RGB channels to ensure consistent scale across input features and stabilize training.

2.1.2 Convolutional Blocks

For the convolutional blocks, we experimented with various convolutional block architectures and found that the following structure performed the best.

- **Block 1**: Input(3) → Conv(32) → Conv(64) → MaxPool → Dropout(0.25)
- **Block 2**: Conv(64) → Conv(128) → Conv(128) → MaxPool → Dropout(0.25)
- **Block 3**: Conv(128) → Conv(256) → Conv(256) → MaxPool → Dropout(0.25)

2.1.3 Fully Connected Layers

For the fully connected (FC) layers, the flattened features are passed through a dense layer (4096 to 512) with dropout, followed by a final output layer (512 to 10 classes).

- Flatten(4096) → FC(512) → Dropout(0.5) → FC(10)

2.1.4 Regularization Techniques

The network incorporates several regularization strategies to combat overfitting and improve generalization performance. Batch normalization is applied after each convolutional layer to stabilize training and reduce internal covariate shift. Dropout is implemented with a probability of 0.25 in the convolutional blocks, while a higher dropout rate of 0.5 is used in the fully connected layers to prevent over-reliance on individual features. These techniques work in conjunction with the data augmentation pipeline to create a robust model that generalizes well to unseen data.

2.2 Vision Transformer

Transformer models are a relatively recent architecture that take advantage of a method called attention to learn the context of tokens in sequential data. Transformers were initially used for natural language processing but has been adapted to image data in a new architecture called Visual Transformers (ViT).

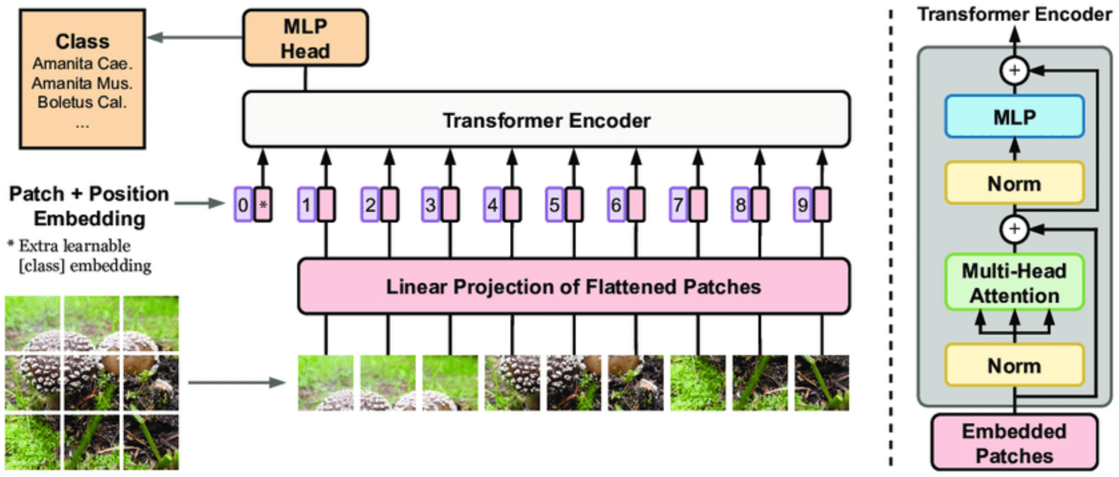


Figure 1: ViT architecture visualization (Source)

2.2.1 Vision Transformer Methods

1. **Embedding:** The first step of ViT is embedding the image. This is done by segmenting the image into a sequence of smaller, flattened patches of pixels. These patch tokens are then embedded to a fixed length vector representation and then appended with a positional embedding token that adds spatial relations to the embedded tokens.

2. **Multi-Head Attention:** Next, these embeddings are passed through the transformer encoder blocks. The innovation in this block is the multi-head attention layers. These layers are a concatenation of many self-attention layers that each learn a different contextual relation between each of the patch embeddings and outputs attention scores that quantify the similarity between patches.

3. **Multi-layer Perceptron:** These attention scores are passed through a fully connected multi-layer perceptron which learns more relations of these latent states to further deepen the network and add more detailed learnability.

4. **Layer Normalization and Residual Connections:** Between the layers in the transformer encoder blocks are residual connections to the inputs of the layers and normalization. The residual connections enhance gradient flow and propagate the original embedding context through the block while the normalization helps regularize the data for stability.

5. **Layer Stacking:** Finally, these transformer blocks will be connected to each other in sequential stacks to further increase the depth of the network and allow for more attention and MLP blocks to learn more parameters. The final output of the transformer blocks are filtered by a fully connected MLP that will finally output a class label for the image.

ViT architecture provides many benefits when compared to other classification models including the ability to understand dependence between spatially distant features unlike CNNs but is also a more computationally expensive network that makes slower inferences and needs more data to train compared to a CNN.

2.3 Adversarial Attacks

Adversarial machine learning aims to deceive machine learning models by providing deceptive input. A machine learning in the real world should be robust to adversarial attacks. White-box attacks are a type of attacks where the weights, parameter, hyper-parameters, etc, of the model are known to the attackers. Evasion attacks are attacks on the testing phase, and does not tamper with the actual model.

2.3.1 FGSM Attack

The Fast Gradient Sign Method (FGSM) is an adversarial attack technique designed to generate adversarial examples by introducing small perturbations to the input data. The generated images are likely not visibly different to the human eye, but a machine learning algorithm would incorrectly misclassify with high confidence. The adversarial image x_{adv} is generated by:

$$x_{adv} = x + \epsilon * sign(\nabla_x \mathcal{L}(\hat{y}, y))$$

where the added perturbation is calculated as the gradient of the loss function $\mathcal{L}$ between the predicted label $\hat{y}$ and the actual label y. This attack is based off of the gradient descent algorithm, where the weights of the model are updated by $w^n = w^{n-1} - \alpha * \nabla \mathcal{L}(\hat{y}, y)$. The gradients points to the direction where changing pixel will result in the biggest increase of the loss function. In gradient descent, the weights are changed based on the direction that gives the steepest descent of the loss function. FGSM follows a similar idea, where the update to the input shifts it along the direction in which the loss function increases most rapidly. FGSM is a single-step attack, so it may not be as effective as iterative methods like projective gradient descent (PGD).

2.3.2 PGD Attack

Like FGSM, PGD is a white-box evasion attack. However, it is an iterative approach based off of FGSM. By iteratively updating the perturbation then projecting the perturbed input back to a fixed range, PGD can generate stronger adversarial examples, and it is regarded as the strongest attack utilizing only the gradients of the loss function. The adversarial image x^t_{adv} of iteration t is given by

$$x^t_{adv} = \Pi_\epsilon(x^{t-1} + \alpha * sign(\nabla_x \mathcal{L}(\hat{y}^{t-1}, y)))$$

where Π is the projection function that projects the adversarial example back onto the ϵ-ball of x. PGD is initialized by adding random noise from a uniform distribution in the range of $(-\epsilon, \epsilon)$

3 Experiments

3.1 Deep CNN

For the Deep CNN, the experiments were conducted to evaluate the CNN model's performance and stability on the CIFAR-10 dataset, with a focus on consistency and reproducibility across training runs.

3.1.1 Training Configuration

To ensure a reproducible and efficient training process, we set random seeds ([42, 123, 456]) for model initialization and data shuffling. The Xavier Initialization method was used for initializing model weights, helping to maintain stability in the training dynamics. The Adam optimizer was chosen for its adaptive learning rate capabilities, effectively minimizing the CrossEntropyLoss used as the loss function. We set the learning rate to 0.001, providing stable convergence throughout training. The model was trained for 30 epochs to achieve convergence while avoiding overfitting. All computations were accelerated using a CUDA-enabled GPU, optimizing training time and handling high computational demands.

3.1.2 Experimental Methodology

The experimental methodology began with model initialization, where random seeds were applied to ensure reproducibility. The model was transferred to the GPU for efficient training, and the optimizer and loss parameters were configured accordingly. The training process involved batch-wise training with data shuffling to minimize bias, and the model was evaluated on the test set after each epoch to monitor generalization. Throughout training, we collected training and test accuracy, loss values, and training time for each epoch. Performance tracking included monitoring accuracy to assess learning progress, recording CrossEntropy loss at each epoch to evaluate model fit and convergence, and measuring total training time per epoch to assess computational efficiency.

```
    device = torch.device('cuda' if torch.cuda.is_available() else 'cpu
    ')
    print(f"Using device: {device}")

    seeds = [42, 123, 456]

    # Run experiment using our custom-made model.
    results = run_experiment(
        model_class=CIFAR10_CNN,
        trainloader=cifar10_trainloader,
        testloader=cifar10_testloader,
        seeds=seeds,
        device=device
)

plot_results(results, seeds)
```

3.2 Vision Transformer

In order to test ViTs, we decided to implement two different ViT models: a from scratched trained ViT and a pre-trained ViT using weights from the official ViT paper. Both models were pulled from Google's HuggingFace and manipulated to better fit the CIFAR-10 data.

3.2.1 From Scratch Design Choices

For the from scratch model, we created a custom transformer model using the HuggingFace ViT functions and defined a model that has fewer and smaller layers than the original ViT model trained on ImageNet.

ViT Configuration:

1. Image Size: CIFAR-10 has an image size of 32 by 32 pixels so that had to be changed since this model original accepted inputs of 224 by 224 for ImageNet images.
2. Patch Size: The patch sizes created should ideally be as numerous as possible while still capturing useful information in each patch. For a 32 by 32 image, we though that any patch smaller than 4 by 4 pixels would be too small so we settled on 4 by 4 patch tokens.
3. Classes: ImageNet has 1000 classes while CIFAR-10 only has 10 so this had to adjusted to 10 classes for our data on the output classification.
4. Hidden Size: This controls the dimensionality of the embedding vectors. Since we are using smaller patches and images, we shrunk the embedding vector length from 768 to 256 as an arbitrary shrinking factor.
5. Hidden Layers: This signifies how many transformer encoder blocks are used in the entire model. ImageNet used 12 layers but we thought that would be excessive so we cut that in half to 6 layers.

6. Intermediate Size: This relates to the size of the hidden layer in the feed forward MLP of the transformer blocks. We also shrunk this from 3072 nodes down to 512 nodes.

7. Attention Heads: The original ViT had 8 attention heads in the multi-head attention blocks. In order to save some time and computation we decreased this slightly to 6 attention heads but kept most due to the importance of this architecture.

For training the model, we used some classic hyper parameters. ADAM is a common optimizer for gradient descent and cross entropy loss is a common loss function for classification. We also stuck to a learning rate of $1e4$ as it seemed appropriate for the model size and made sure to train on a set seed for reproducibility. We did have to experiment with values for the number of epochs of training and the batch size. We found that 40 epochs was a worthy compromise for training time while not sacrificing too much performance. We also used a batch size of 100 as it was the largest batch size we could use on the GPU we were running it on. If we had more resources these two parameters could be adjusted for more thorough and better training.

3.2.2 Pre-Trained Design Choices

The pre-trained model borrows the weights and a model called Tiny-ViT from the original Tiny-ViT paper that created a smaller and more efficient ViT model. We kept the weights for all layers except the final feed forward layer which was replaced from a 1000 node layer for ImageNet and replaced with a 10 node layer for CIFAR-10. For ease of use, the input image size of the model was not changed 20 32 by 32 for CIFAR-10 and instead the CIFAR-10 data was up-scaled to fit the ImageNet data size. We also used many of the same design choices for this model as the trained from scratch model except have a smaller batch size for the larger images, a smaller learning rate that was found through trial and error, and a smaller epoch training time of 2 epochs find jointly with the learning rate that minimized training time and maximized performance.

3.3 Adversarial Attacks

Adversarials images are generated by FGSM and PGD algorithms, and tested on the pre-trained ViT model. The pre-trained model is used here since it was previously trained on ImageNet data in addition to the CIFAR-10 data and therefore should be more robust. It also gave the best accuracy during testing. Both algorithms were run with $\epsilon = \{0.001, 0.005, 0.01, 0.05\}$. For PGD, the hyperparameters of $\alpha = 0.01$ was chosen, same as in the original paper, and it was allowed to run for 4 iterations. A smaller number of iteration was chosen due to hardware constraints and a desire for the test to run in a reasonable amount of time. We show in the Results section that even at 4 iterations, the PGD is still very effective.

4 Results

4.1 Deep CNN

The experimental results we collected demonstrates the effectiveness and stability of our DCNN model on the CIFAR-10 classification task. The model achieved a mean training accuracy of 91.86% with a standard deviation of 0.06%, indicating highly consistend performance across diffent initialization seeds (*see Figure 2*). More importantly, our model demonstrated strong generalization capabilities with a mean test accuracy of 88.23% and a standard deviation of 0.45%.

Furthermore, the training process, which ran for 40 epochs, completed in 54 minutes and 25 seconds on a A100 GPU hardware. The training process, as evidence by the loss curves, shows a clear and consistent pattern of convergence across all three random seeds. The training loss decreased substantially in the initial epochs, starting from approximately 1.6 and steadily declining to 0.2 by the end of the training.

The test accuracy curves reveal three distinct phases of learning. The first phase, spanning epochs 0-10, showed rapid improvement in accuracy from 50% to approximately 80%. The second phase, during epochs 10-25 demonstrated continued but only gradual improvement. The final phase, from epoch 25 onwards, showed fine-tuning of the model's performance, ultimately reaching stability around 88% accuracy.

The consistency across different random seeds (42, 123, and 456) is particularly noteworthy, as evidenced by the nearly overlapping loss and accuracy curves. This consistency suggests that the model's performance is robust and reproducible, rather than being the result of fortunate initialization. The small standard deviation in both training and test accuracies further supports this conclusion.

The model successfully avoided overfitting, as indicated by the relatively small gap between training and test accuracies (approximately 3.4 percentage points). This suggests that the chosen architecture and hyperparameters struck an effective balance between model capacity and generalization ability.

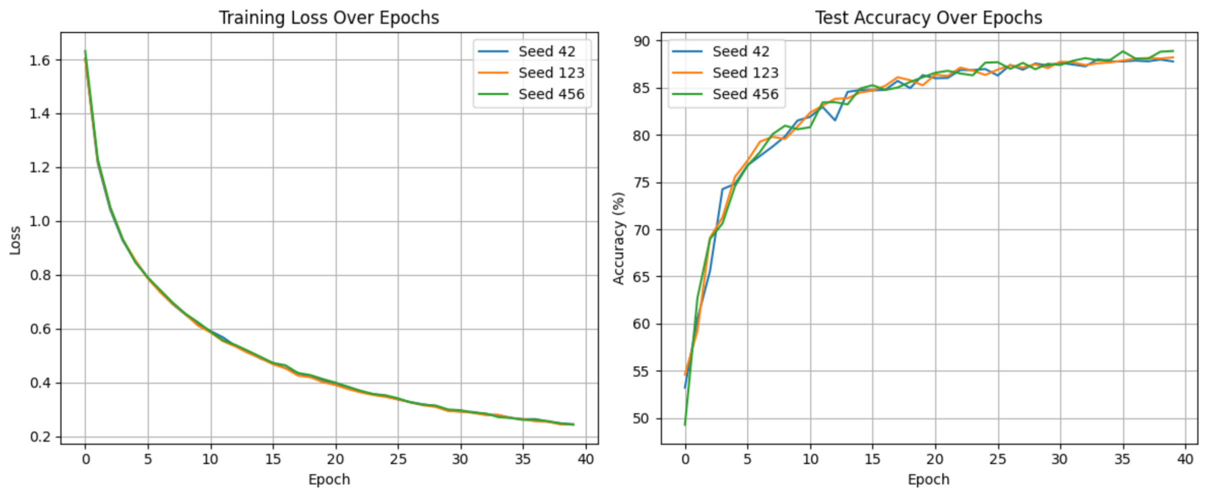
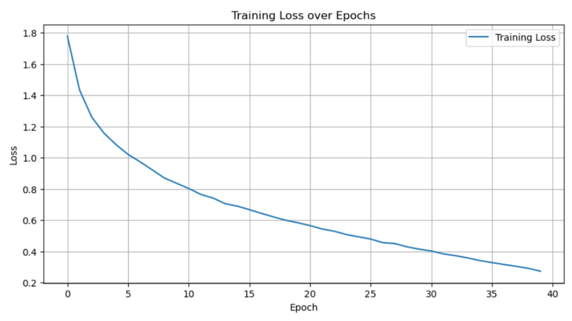
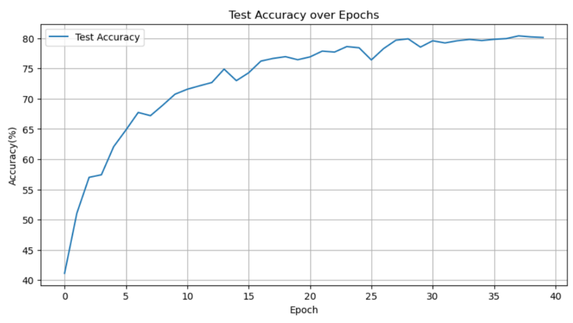


Figure 2: CNN Loss Curves and Accuracies



(a) Training loss vs batches for ViT trained from scratch



(b) Testing acc vs batches for ViT trained from scratch

Figure 3: Performance curves for pre-trained ViT

4.2 Vision Transformer

As described in Experiments, we conducted testing on two different ViT models. Both the from scratch and pre-trained models were given time to train on the training data set and evaluated on the testing data set periodically to create a model with the best performance in a reasonable amount of time given our computational resources.

4.2.1 From Scratch Results

After training the custom ViT model for 40 epochs, we got a final performance accuracy of 80.14% on the test data set. Figures 3a and 3b show the training loss and test accuracy of the model every epoch during training. The total training for 40 epochs took about 52 minutes on a 6GB NVIDIA GPU. You can also find a confusion matrix in Figure 4 of the results for the test data set predictions. A common mistake of my model is a confusion between cat and dog images which is understandable due to their relative similarity between classes of CIFAR-10.

4.2.2 Pre-Trained Results

After training the pre-trained TinyViT model for 2 epochs, we got a final performance accuracy of 94.49% on the test data set. Figures 5a and 5b show the training loss and test accuracy of the model every epoch during training. The total training for 2 epochs took about 36 minutes on a 6GB NVIDIA GPU. You can also find a confusion matrix in Figure 6 of the results for the test data set predictions.

Both models trained very well for there slightly different architecture and size with relatively small training time and iterations. We notice very little over fitting when looking at the loss and accuracy curves although both model's performance gains taper out drastically towards the end of their training. It is important to note that these models could be trained much more to gain extra performance increases but due to time constraints and computational resources, we felt the current results properly reflect the abilities of the models.

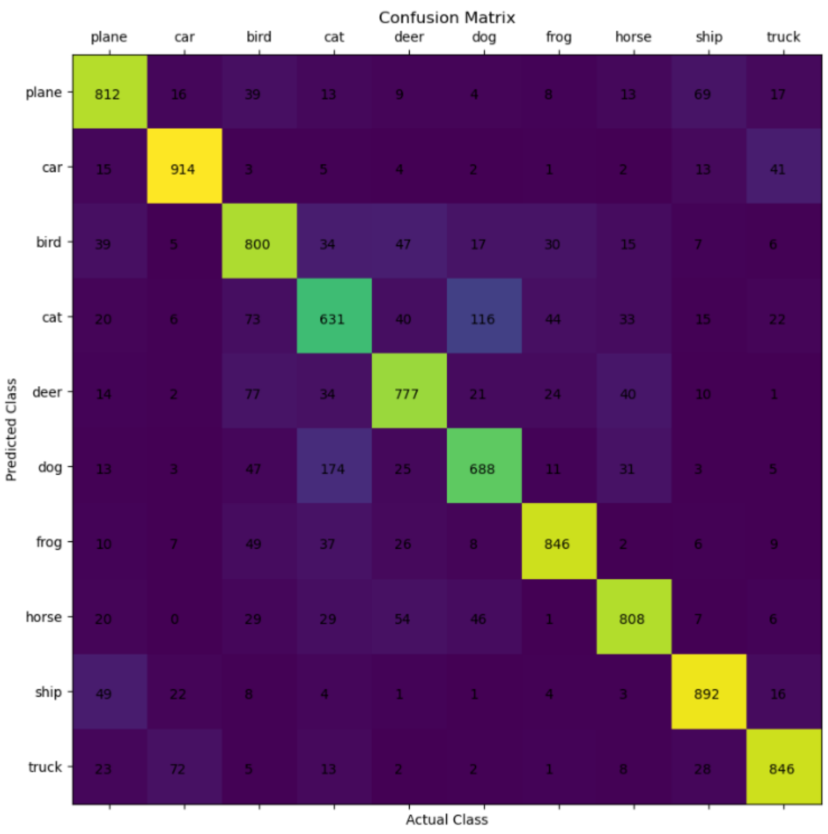
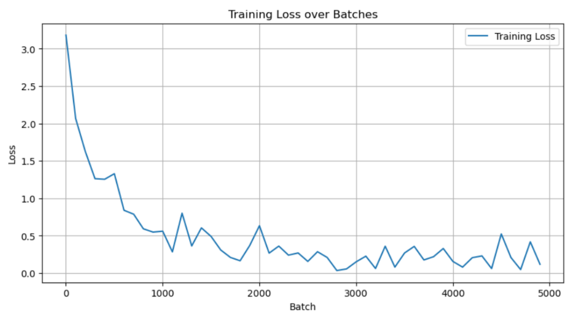
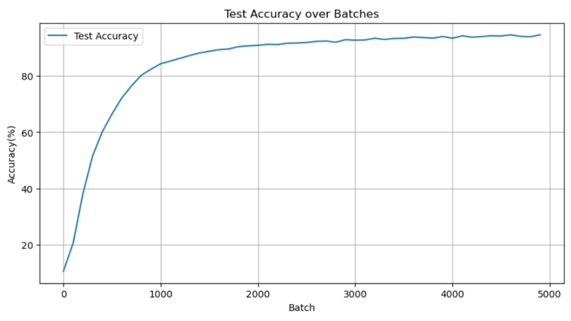


Figure 4: Confusion matrix for ViT trained from scratch



(a) Training loss vs batches for pre-trained ViT



(b) Testing acc vs batches for pre-trained ViT

Figure 5: Performance curves for pre-trained ViT

4.3 Adversarial Attacks

4.3.1 Accuracy

Beginning with a small perturbation of 0.001, both FGSM and PGD showed a marked decrease in accuracy. However, at each epsilon value tested, PGD performed better in terms of reducing the model's testing accuracy. From the values of epsilons tested, the trend suggests the possibility that FGSM can only reduce the testing accuracy of the pre-trained ViT model to about 7%. With PGD, the performance significantly decreased with every change in perturbation and eventually reduced the testing accuracy to 0%. Examining the relavent confusion matrix shows that the model correctly classified the none of the adversarial images (Figure 8). This demonstrates that the small number of iterations did not limit the performance of PGD.

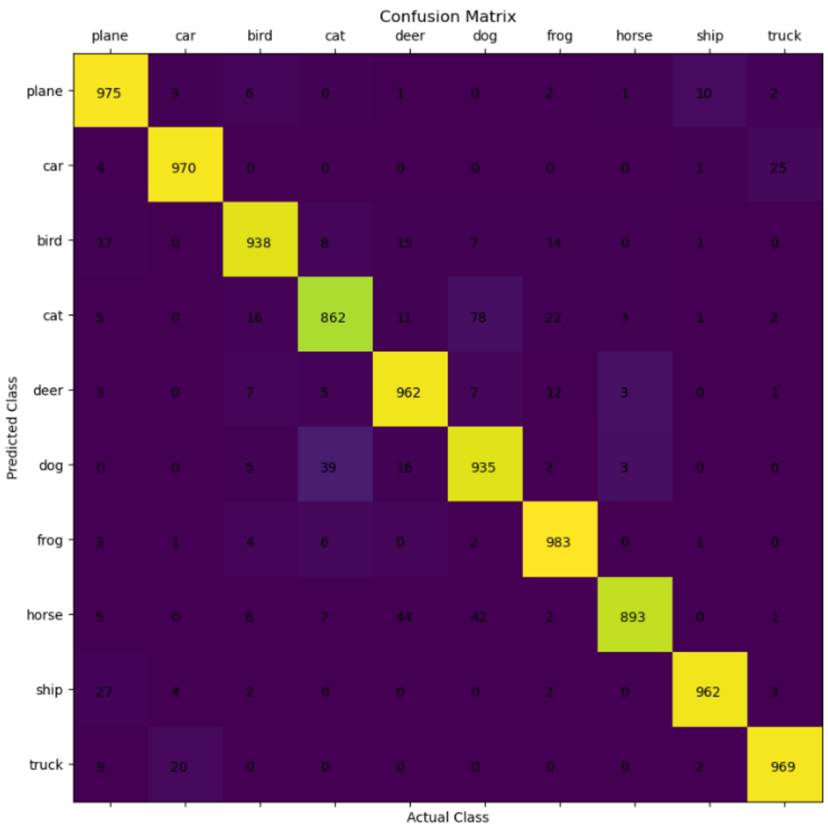
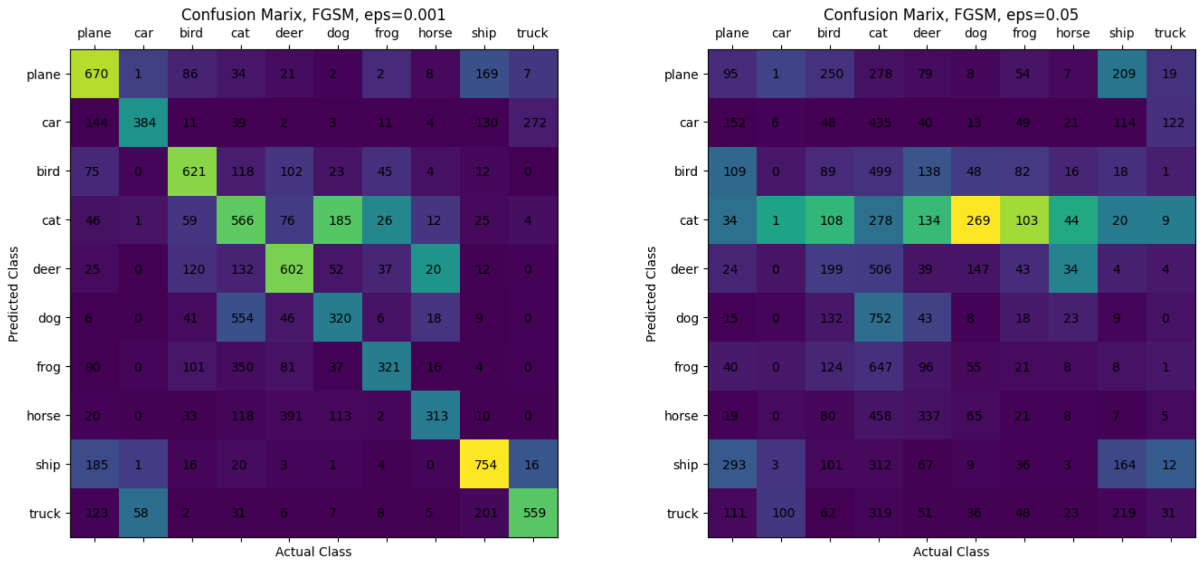


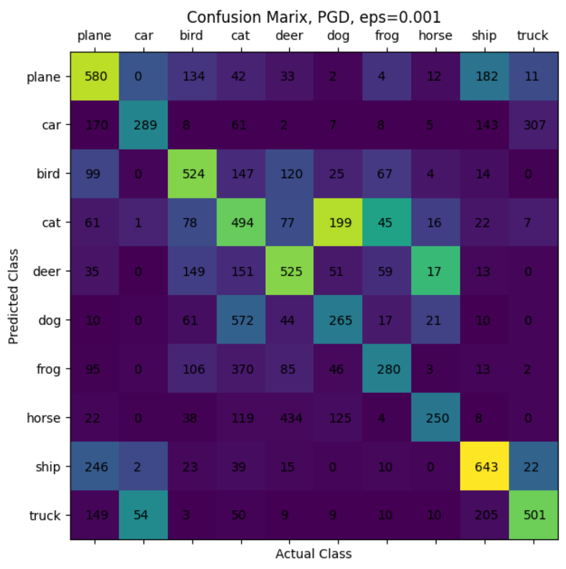
Figure 6: Confusion matrix for pre-trained ViT



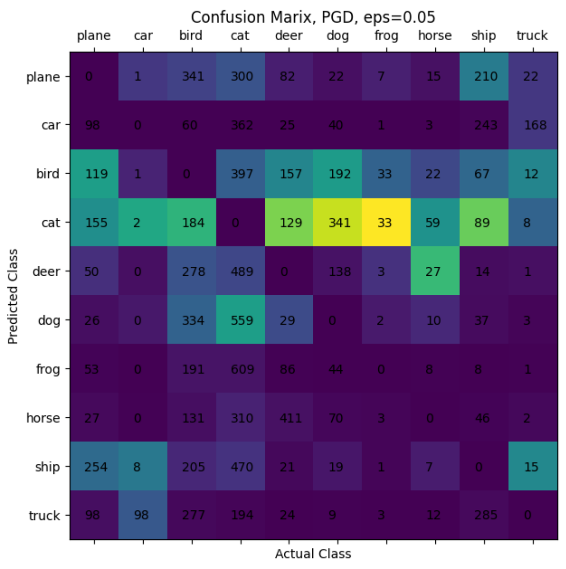
(a) $\epsilon = 0.001$ **(b)** $\epsilon = 0.05$

Figure 7: Confusion matrix for FGSM attack

In FGSM, the testing accuracy decreased from 94.68% to 40.08% when allowing even a small perturbation of $\epsilon = 0.001$. The confusion matrix of $\epsilon = 0.001$ and $\epsilon = 0.05$ are shown (Figure 8 a, b). When $\epsilon = 0.05$, for classes like "cat", and "ship", the model was still able to correctly label these images at a rate better than random chance, so it is not as effective as PGD (Figure 7 b). Since PGD is an iterative approach, it has a higher time complexity depending on the numbers of iterations chosen. In our experiment, PGD



(a) $\epsilon = 0.001$



(b) $\epsilon = 0.05$

Figure 8: Confusion matrix for PGD attack

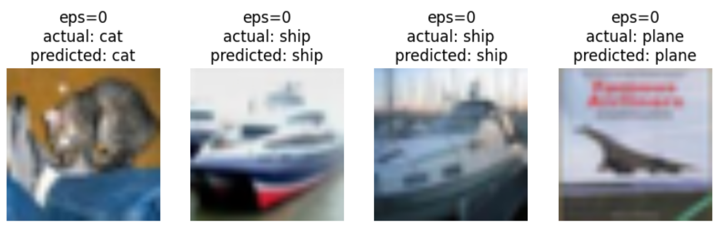
Test Accuracy	$\epsilon = 0$	$\epsilon = 0.001$	$\epsilon = 0.005$	$\epsilon = 0.01$	$\epsilon = 0.05$
FGSM	94.68%	51.1%	11.51%	6.9%	7.39%
PGD	94.68%	43.51%	8.09%	0.28%	0.0%

Table 1: Test accuracy on FGSM and PGD at different ϵ values

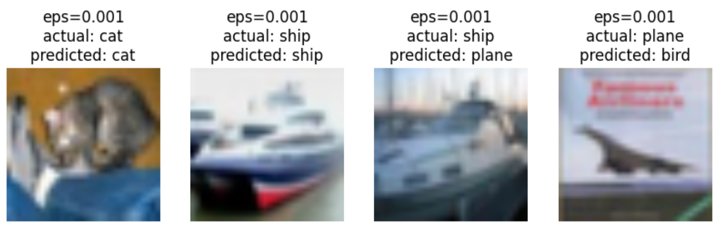
took 33 minutes to run while FGSM took 15 minutes to run. Therefore, FGSM gains some efficiency at the cost of performance.

4.3.2 Adversarial image output

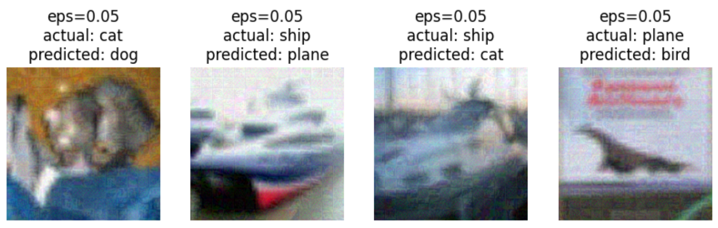
Sample adversarial images generated by FGSM and PGD are displayed. The same input images were used for both attack algorithms and are displayed in Figure 9 for reference. For FGSM, when the perturbation is small ($\epsilon = 0.001$), the images does not look visibly different to the original image. When $\epsilon = 0.05$ for the FGSM attack, the adversarial images are perceptibly noisier to the human eye (Figure 9 a, c). However, the images generated by PGD don't appear different to the original to the human eye using these epsilon values (Figure 10, a, b).



(a) Original images

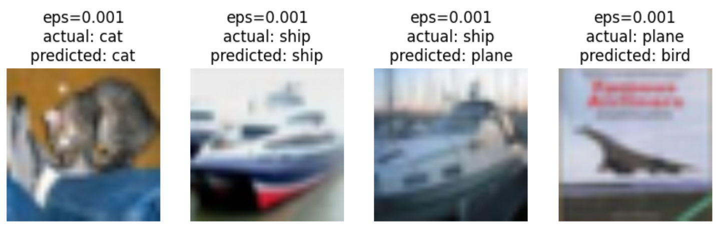


(b) $\epsilon = 0.001$

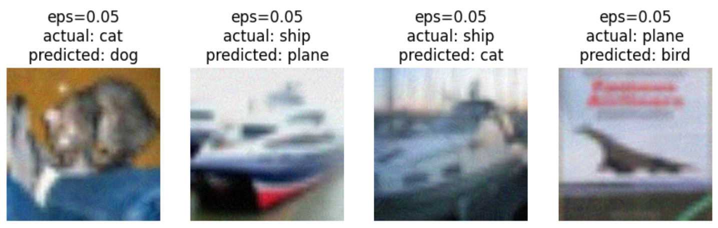


(c) $\epsilon = 0.05$

Figure 9: Original test images and sample adversarial images from FGSM attack



(a) $\epsilon = 0.001$



(b) $\epsilon = 0.05$

Figure 10: Sample adversarial images from PGD attack

5 Conclusion

Deep Convolutional Neural Networks and Visual Transformers are two different ways to approach object classification for the CIFAR-10 data set. From our experiments, we found that deep CNNs have great accuracy and computational speed achieving performance better than a from scratch ViT trained for the same number of training epochs and in shorter time accounting for GPU. ViT on the other hand, had much better results if using a pre-trained version fine tuned on the data set which is expected due to the deeper nature of the architecture. We also explored how FGSM and PGD adversarial attacks effect a model, in our case the pre-trained ViT model. We found that both attacks were effective and dropped the accuracy of the network down significantly but that PGD seemed to be a bit more effective at every epsilon value. Overall, we explored how deep learning can solve a classification task and how they can be susceptible to white box attacks.