ADVERSARIAL EXAMPLES FOR NEURAL NETWORKS

Noah Lee
Khoury College of Computer Sciences
Northeastern University
lee.no@northeastern.edu

Richard Li
Khoury College of Computer Sciences
Northeastern University
li.rich@northeastern.edu

Faith Occhipinti
Khoury College of Computer Sciences
Northeastern University
occhipinti.f@northeastern.edu

December 9, 2020

ABSTRACT

Adversarial examples are intentionally crafted examples designed to trick a model to produce inaccurate predictions. Neural networks are usually vulnerable to adversarial attacks since they tend to work best on most, but not all, examples. In this project, we trained two image-classifications models and attempted to fool them with adversarial examples. After evaluation, we noticed a significant decrease in the accuracy when predicting the adversarial examples.

1 Introduction

In recent news, some online social media sharing platforms have begun censoring content relating to certain topics [3] [4]. While there are people manually screening the content uploaded by users, these platforms also have some algorithmic model in place that automates the classification process due to the sheer volume of content being shared. This led us to the question: is it possible to artificially modify the content so that the changes are indiscernible to a human audience, but will trick the algorithm into giving the wrong classification?

For the ease of presenting our results, we decided to use the popular CIFAR-10 [2] dataset for image classifications. We built neural networks and trained them on the CIFAR-10 images, and generated adversarial examples based on the CIFAR-10 with minimal changes to trick the trained network. Among us three group members, Richard did research on neural network architectures and adversarial example generation and completed the majority of this paper, Noah handled constructing the models with PyTorch and training and hyperparameter tuning, Faith did additional research about architectures and formatting of the lightning talk.

Compared to our topic proposal, our project is less about a building a full-fledged GAN and more on adversarial machine learning. We measured the effect of varying hyperparemters, and we used PyTorch framework to build multiple neural network architectures.

2 Methodology

2.1 Data

We decided to use the CIFAR-10 dataset [2], a dataset of 32 by 32 images with 3 color channels, split into 10 different classes. In our experimentation we utilize several different train/validation/test splits based on the model we are currently training.

2.2 Data Augmentation

Our data augmentation randomly changes the input we give to our models, effectively increasing the variation in our dataset and thus increasing the effective size of our dataset.

Specifically, for most of our models our data augmentation consists of randomly flipping our images vertically and horizontally with a probability of 0.5, randomly rotating the images between -45 and 45 degrees, and normalizing each image by the mean and standard deviation of the entire training dataset.

For the ResNet50 model, we only flipped the image horizontally, rotated by 30 degrees, and normalized by the mean and standard deviation of the ImageNet dataset, which ResNet50 was pretrained on. Additionally, we reshaped each of our input images to 224 by 224 pixels to conform with the model architecture.

2.3 Model Architectures

2.3.1 Baseline Convolutional Network

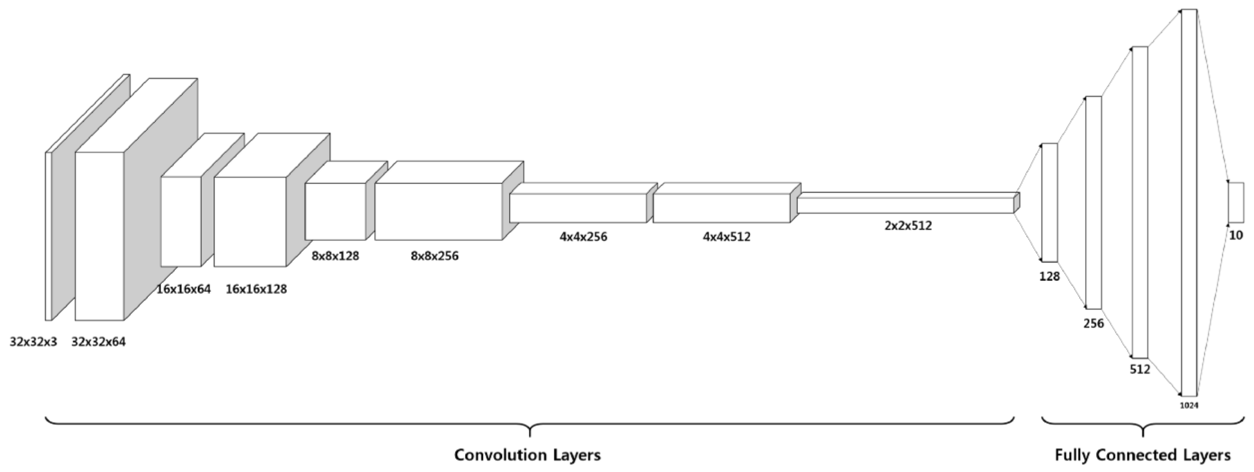

Figure 1: We used the above convolutional model architecture [1] as an example, and constructed an equivalent model using PyTorch.

This model is a baseline "average" performing convolutional neural network. Its architecture consists of 4 repeated blocks of a convolutional layer, max pooling, a ReLU activation, and a batch normalization. This is then followed by 4 dense layers, with each layer being followed by dropout and batch normalization. The output of the model is condensed into 10 neurons representing the 10 classes of the CIFAR-10 dataset.

For training this model, we used the following hyperparameters: a training set size of 40,000 examples, 30 epochs, a minibatch size of 256 examples, dropout probability of 0.3, and a learning rate of $1e^{-3}$.

2.3.2 DenseNet

The second model that we used was DenseNet169, representing a strong classifier. DenseNet is composed of multiple dense blocks, where each block feeds into all subsequent layers instead of just the next layer. This alleviates the vanishing gradient problem, since the gradients are able to propagate directly from the last layer of the model to the earlier layers without passing through other layers.

For our project, we utilized an entirely pretrained model that was trained on the CIFAR-10 dataset. Thus, all we had to do was evaluate this out of the box model.

Layers	Output Size	DenseNet 169
Convolution	112×112	7×7 conv, stride 2
Pooling	56×56	3×3 max pool, stride 2
Dense Block (1)	56×56	$\begin{bmatrix}1\times 1\text{ conv}\\3\times 3\text{ conv}\end{bmatrix}\times 6$
Transition Layer (1)	56×56	1×1 conv
	28×28	2×2 average pool, stride 2
Dense Block (2)	28×28	$\begin{bmatrix}1\times 1\text{ conv}\\3\times 3\text{ conv}\end{bmatrix}\times 12$
Transition Layer (2)	28×28	1×1 conv
	14×14	2×2 average pool, stride 2
Dense Block (3)	14×14	$\begin{bmatrix}1\times 1\text{ conv}\\3\times 3\text{ conv}\end{bmatrix}\times 32$
Transition Layer (3)	14×14	1×1 conv
	7×7	2×2 average pool, stride 2
Dense Block (4)	7×7	$\begin{bmatrix}1\times 1\text{ conv}\\3\times 3\text{ conv}\end{bmatrix}\times 32$
Classification Layer	1×1	7×7 global average pool
	1000	1000D fully-connected, softmax

Figure 2: The DenseNet169 architecture, sourced from [6].

2.3.3 ResNet50

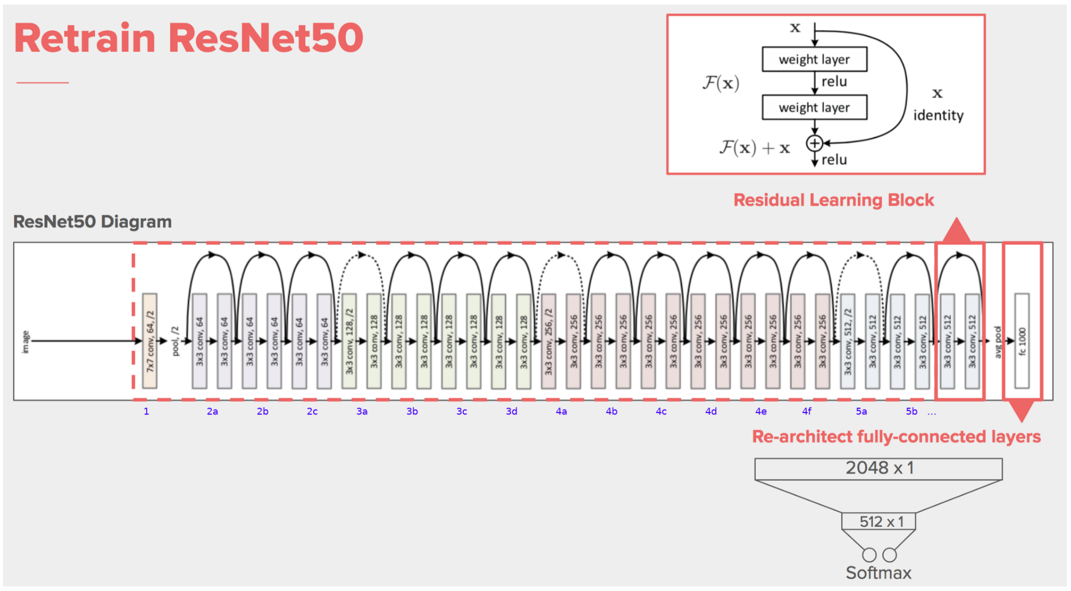

Figure 3: The ResNet50 architecture, sourced from [7].

We also attempted to fine tune a pretrained ResNet50 for our task. The ResNet architecture is a convolutional network, but it also learns residuals, which are layers that are added back to the input with simple matrix addition. Since the information from the input image is able to propagate forward through the network without being lost, the problem of the vanishing gradient is alleviated.

We attempted to perform transfer learning on a ResNet50 model that was pretrained for the ImageNet task. We tried to do this by freezing the gradients of all of the convolutional layers, and replacing the final fully connected layer with several fully connected layers that condensed to output 10 classes instead of 100.

Unfortunately, we were not able to get the ResNet50 model to converge on the CIFAR10 dataset. We tried to guarantee convergence by only training with an small portion of our dataset (1250 images). This normally would result in extreme overfitting on the training data and the training set accuracy converging to 100%, but this did not happen, which indicates that there is a problem in our training approach. Since the accuracy and loss did improve, this means that the model was able to learn, but did so extremely slowly. With more time, we would like to attempt different training approaches to try to get this ResNet model to more accurately classify the CIFAR-10 images. For instance, we would try doing learning rate scheduling which could let us train quickly for the first few epochs and then tune our weights carefully for the final few accuracy percentage points.

For training this model, we used a smaller training set of 1250 images. We trained for 100 epochs with a minibatch size of 256 and a learning rate of $1e^{-3}$.

2.4 Adversarial Examples Generation

$$C = \frac{1}{2}\|y_{goal} - \hat{y}(\vec{x})\|_2^2 + \lambda\|\vec{x} - x_{target}\|_2^2$$

We used the above cost function [5] of adversarial examples generation as inspiration, and created our own cost function:

$$C(\hat{y}, y) = -\sum_i y_i \log(\hat{y_i}) + \lambda_1 \sum_j^M |W_j| + \lambda_2 \sum_j^M W_j^2$$

Here, we included both L2 and L1 regularization to make both the number of pixels we are changing and the amount we are changing each pixel by variables to be learned.

This loss function includes three components:

Firstly, we flip the sign on the calculated cross entropy loss. Effectively, this makes it so that performing gradient descent will now try to minimize the negative of the loss, which is equivalent to maximizing the positive loss. As a result, a resulting network will learn to misclassify all of its inputs.

Secondly, we include L1 regularization, which shrinks smaller (and thus less important) weights to zero. We hope that this results in images with only select pixels changed, and the rest unchanged.

Finally, we also have L2 regularization, which shrinks weights that are larger in magnitude. Our aim for including this term was to minimize the visual impact of changing the input images. The intent was that a human looking at the images wouldn't really be able to notice that the image had changed from the original, since all of the pixel changes are small in magnitude.

Regularizing the weights of the image generation model while trying to maximize misclassification rates is a difficult task, where a careful balance must be found between all three variables. Thus, in our experimental section we include an experiment where we vary the L1 and L2 lambdas while recording the accuracy, in a hope to identify a good balance.

For training a network to generate these adversarial images, we took inspiration from the ResNet architecture. Our network consists of 2 linear layers that have the same input and output size. For each layer, we use a tangent hyperbolic activation function, since we want our perturbations to be both positive and negative, and centered around 0. Once we pass an input image through the layers, we add it back to the original image to create a slightly perturbed image. We then pass the output image to a different classification model.

For performing gradient ascent on our dataset, we used half of the training set, 20,000 images. We trained for 15 epochs with a batch size of 128 images and a learning rate of $1e^{-5}$. The lambda parameters for L1 and L2 regularization respectively were $1e^{-2}$ and $1e^{-3}$.

3 Experiment and Results

The dataset was already pre-split into training and testing. We split the training set of the dataset into training and validation with a 4:1 ratio, and kept testing set as is. After training the network on the training and validation set, we calculated the total loss and accuracy on the testing set. Here are the test loss and test accuracy of the neural networks we built after training:

	Test loss	Test accuracy
ConvNet	.668	77.47%
DenseNet	0.244	94.00%
ResNet	4.913	27.44%

Resnet, in particular, did not have a great accuracy even after extensive hyperparameter tuning, but we decided to include it nevertheless for an extra datapoint.

For the adversarial examples generation, our goal is to decrease the accuracy of the three trained networks. We tuned L1 and L2 with the following result:

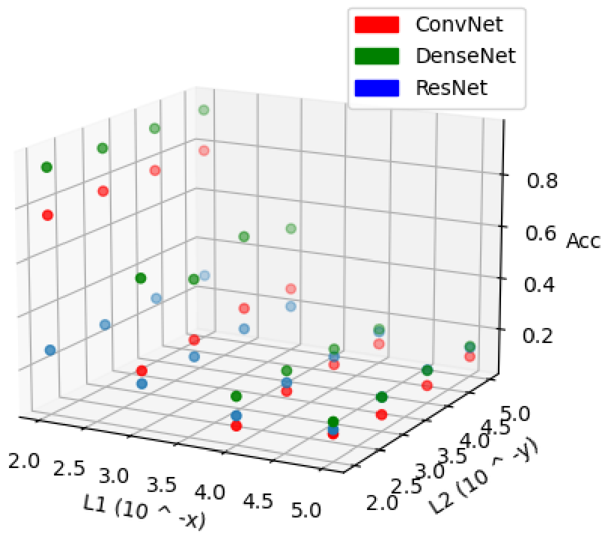

Figure 4: 3D scatterplot of tuning L1 and L2

It's interesting to see how L2 regularization barely had an effect, while L1 had fairly significant effect on decreasing the accuracy, indicating that the number of pixels being modified can easily sway our trained models, and the amount of change per pixel matters relatively less.

In particular, the following is the confusion matrix for ConvNet:

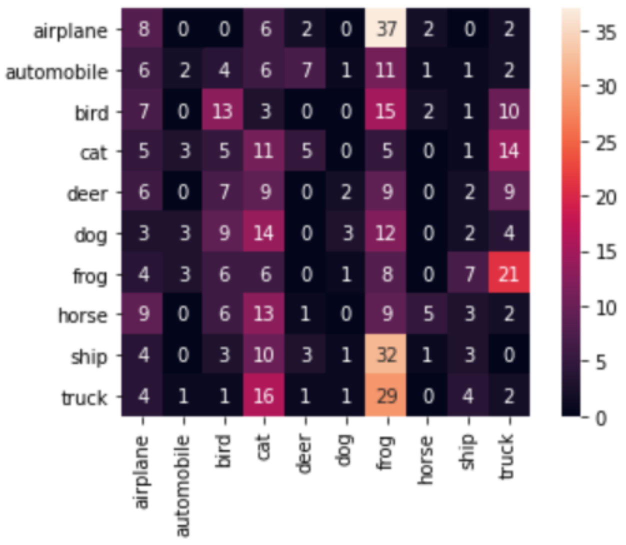

Figure 5: Confusion matrix for ConvNet

According to the matrix, our modifications to the images made it much more likely for this model to classify it as a frog.

4 Conclusions

The difference in the three neural networks suggests it is easier to come up with adversarial examples to fool a relatively simpler network than a deeper/more complex network.

We initially thought that we would spend the majority of time understanding the concepts and the approaches to tackle the generation of adversarial examples. However, it turns out that the approaches are pretty straight forward, but implementing them in PyTorch took a lot longer than expected.

5 Appendix

To run our code, a user can either run the entire notebook at once, or run certain sections of the code. We have designed the notebook so that each section can be run independently of the other sections, with the exception of the Adversarial Image Generation section and the results section. For these sections, you can alternatively upload the trained model states and use those.

Our code can be found at the following link:
`https://colab.research.google.com/drive/1BVxW-FYvEfQ8E71MEotuPQNgAlRfA7wQ?usp=sharing`

In addition to this, a zip containing trained models has been submitted to Canvas. To run certain sections of the code, for instance, the DenseNet section, you will need to upload the corresponding model to Google Colab and move it into the proper folder. Alternatively, if you want to run only the results sections, then you can upload all of the models and selectively run only those cells.

References

[1] `https://towardsdatascience.com/cifar-10-image-classification-in-tensorflow-5b501f7dc77c`

[2] Learning Multiple Layers of Features from Tiny Images, Alex Krizhevsky, 2009.

[3] Culliford, E., &amp; Dave, P. (2020, October 14). YouTube bans coronavirus vaccine misinformation. Retrieved December 09, 2020, from`https://www.reuters.com/article/health-coronavirus-youtube-int/youtube-bans-coronavirus-vaccine-misinformation-idUSKBN26Z21R`

[4] Leskin, P. (2018, December 08). Tumblr is banning all NSFW content - and people are worrying it's the beginning of the end for the Verizon-owned website. Retrieved December 09, 2020, from `https://www.businessinsider.com/tumblr-bans-nfsw-content-and-users-say-the-platform-will-suffer-2018-12`

[5] `https://medium.com/@ml.at.berkeley/tricking-neural-networks-create-your-own-adversarial-examples-a61eb7620fd8`

[6] `https://github.com/huyvnphan/PyTorch_CIFAR10`

[7] Pretrained implementation of the paper, Deep Residual Learning for Image Recognition by Kaiming He, Xiangyu Zhang, Shaoqing Ren, Jian Sun. From `https://PyTorch.org/docs/stable/torchvision/models.html`