Deep Learning Algorithms and the Threat of Adversarial Examples

Connor Claypool

30 April 2019

Contents

Abstract 1
1 Introduction 1
1.1 Background 1
1.2 Aim 3
2 Procedure 4
2.1 Environment Setup 4
2.2 Preparing the Dataset 6
2.3 Training a Classification Model 9
2.4 Generating Untargeted White-Box Adversarial Examples 10
2.5 Generating Targeted White-Box Adversarial Examples 13
2.6 Assessing the Transferability of the Generated Adversarial Examples 16
2.7 Generating More Transferable Adversarial Examples 19
2.8 Countermeasures: Training on Adversarial Examples 22
3 Results 26
4 Discussion 27
4.1 General Discussion 27
4.2 Conclusions 28
4.3 Future Work 29
References 29
Appendices 31
Appendix A: Output Generated During Training of Deep Learning Algorithms 31

Abstract

Deep learning algorithms represent the cutting edge of machine learning technology, enabling advanced applications such as facial recognition, machine translation and intelligent agents which exceed human performance even in complex tasks such as the game of Go. However, they are known to be vulnerable to adversarial examples, input data which has been deliberately perturbed to elicit an erroneous output such as a false classification. The aim of this project was to demonstrate the creation and use of these by training a deep learning-based image classification model and generating adversarial examples to fool this model. Additional aims were to assess and improve upon the transferability of these examples and to re-train the model on a combination of clean and adversarial data to create a model more robust to adversarial examples. Adversarial examples generated both with and without specific target labels were successful in fooling the model but were extremely limited in their inter-model transferability. Using a combination of models in the creation of adversarial examples led to samples which successfully fooled each model in the combination used, but which had very limited transferability beyond this. Finally, retraining the model on data which included adversarial examples resulted in a model robust only to adversarial examples which were generated using the same specific model as those used in training. These results confirmed that creating effective adversarial examples is indeed possible, but that methods other than those used in this project are needed to create transferable adversarial examples and robust defence mechanisms.

1 Introduction

1.1 Background

Deep learning refers to a popular and incredibly powerful class of machine learning algorithms with a wide range of modern applications (LeCun, Bengio and Hinton, 2015). Facebook's face recognition software (Taigman *et al*, 2014; Munson, 2015), Google Translate (Le and Schuster, 2016) and Google's Go-playing software AlphaGo Zero (Silver *et al.*, 2017) all make use of deep learning technology to closely approach or even exceed human performance in their respective domains. In the field of computer security, possible uses of deep learning include the detection of fraudulent user interactions (Esman, 2017), malware (Raff *et al.*, 2017) and even network intrusions (Yin *et al.*, 2017). While the first example of deep learning appeared in 1965, it has taken many years of experimental development, coupled with a huge increase in the availability of computing power and training data, to bring deep learning to its current viable state (Dettmers, 2015a).

In general, deep learning algorithms are based on deep neural networks, software systems very loosely analogous to the biological brain. Deep neural networks are composed of units commonly known as artificial neurons, though it should be noted that these bear only a passing resemblance to their biological counterparts. These units are essentially mathematical functions with one or more inputs and one or more parameters. In the simplest and most

common case, the inputs are each multiplied by a corresponding weight parameter and then summed together with a bias parameter. The resulting number is then passed through some non-linear function known as an activation function, so called because the output of an artificial neuron is known as an activation. More complex units do exist, however, such as long short-term memory units (LSTMs) (Dettmers, 2015b).

In deep neural networks, units are combined in multiple layers, with the outputs of one layer becoming the inputs of the next. Layers may be fully connected, with every output of one layer being passed to every unit of the next, or may be structured in a more complicated fashion. Additionally, deep neural networks often incorporate other features such as pooling, where a layer's activations are reduced by grouping them and taking, for example, the maximum activation of each group. Each layer represents the input data with an increasing level of abstraction, with the input layer consisting of the raw data and the output layer aiming to distil one or more high-level characteristics of the input data (Dettmers, 2015b). For example, the inputs could be the numerical pixel data of an image, and the outputs could be probabilities that different categories of object are present in the image. The power of deep neural networks is based in the fact that their structure allows the approximation of any continuous mathematical function. However, this relies on an effective set of parameters being discovered (Nielsen, 2018).

The training of deep neural networks, by which the parameters are set, makes use of the techniques of backpropagation and gradient descent. Input data is passed through the network, and the outputs are compared against the desired outputs, which allows an error or loss quantity to be calculated using a function known as a loss function. Backpropagation is the method by which the gradients of the loss with respect to each parameter, i.e. how the loss would change depending on the value each parameter, are calculated. This allows the weights to be updated using gradient descent, where each parameter is adjusted in the direction which would cause the loss to decrease. This is done by changing each parameter by its gradient multiplied by the negative of a quantity called the learning rate. Performing these steps for every item in the set of training data, usually in batches where multiple sets of outputs and their losses are computed simultaneously, is known as an epoch. Training usually involves many epochs, with performance being evaluated throughout using a validation set consisting of data which the network has not been trained on (Dettmers, 2015a).

Despite their immense power and versatility, it has been discovered that deep neural networks are vulnerable to adversarial examples. Adversarial examples are data inputs which have been deliberately perturbed in such a way that they maintain their original meaning in the eyes of a human, but fool the network into providing an erroneous output, such as a highly confident false classification (Szegedy *et al.*, 2013). These may be used in white-box attacks, in which the parameters of the model are known, or black-box attacks, in which they are not known and examples are created with the aim of fooling a variety of different models (Kurakin *et al.*, 2018). Additionally, physical adversarial objects can be created to fool deep learning-based computer

vision systems whose input is a real-time camera feed (Athalye *et al.*, 2017). Many implementations of adversarial examples focus on image classification, as the pixel data of images is continuous and can be perturbed evenly in imperceptibly small amounts. This is more challenging with discrete data such as text where all changes are noticeable. Nevertheless, the generation of adversarial examples has been successfully demonstrated in areas including the sentiment analysis of text (Alzantot *et al.*, 2018) and even malware detection (Kolosnjaji *et al.*, 2018).

One simple and highly efficient method of generating adversarial images designed to fool image classification models is known as the fast gradient sign method. Essentially, this method involves calculating the gradients of numerical pixel data in much the same way as the gradients of the weights are calculated during training. Each numeric element is then altered very slightly in the direction which would cause the loss to change as desired, with each small change contributing to a large effect (Goodfellow, Shlens and Szegedy, 2014). This may be used to increase the loss in relation to the true label of the image, with the aim of causing a general false classification, or to decrease the loss in relation to a false label, with the aim of causing a specific false classification (Kurakin, A., Goodfellow, I. and Bengio, S. 2016a). Additionally, the perturbations may be applied in a single step, or in a series of iterations. The latter of these is highly effective even for specifically targeted adversarial examples, but comes at the cost of inter-model transferability due to the possibility of the iterative process resulting in too close a fit to one specific model (Kurakin, A., Goodfellow, I. and Bengio, S. 2016b).

The possibility of adversarial examples has a variety of implications for computer security. Adversarial images or text on websites could evade automated content moderation, while physical adversarial objects could be deployed to impede the operation of future self-driving cars (Eykholt *et al.*, 2017). Facebook's face recognition software, which is designed to alert users when their photos are used fraudulently by another account (Munson, 2015), could also be at risk of circumvention. Most obviously, adversarial perturbations applied to malware could greatly decrease the effectiveness of next-generation anti-virus software. However, there could be some positive applications, such as to defend against automated captcha solving. Moreover, there are some possible countermeasures against adversarial examples, including de-noising input data, training networks to detect and reject adversarial inputs, and training networks on adversarial examples alongside clean data (Kurakin *et al.*, 2018).

1.2 Aim

The aim of this project is to demonstrate the creation of adversarial examples in the area of image classification using the fast gradient sign method, and then to implement some form of countermeasures against these. The specific objectives are as follows:

- To train a deep learning algorithm to solve an image classification problem

- To use the fast gradient sign method to generate untargeted white-box adversarial examples which are perturbed as imperceptibly as possible, and to evaluate their effectiveness
- To use the fast gradient sign method to generate targeted white-box adversarial examples, whose purpose is to elicit a specific false classification, again perturbing these as imperceptibly as possible, and to evaluate their effectiveness
- To assess the inter-model transferability of the generated adversarial images, and then to improve upon this level of transferability by using an ensemble (or combination) of models to generate examples, and to evaluate the effectiveness of this technique
- To implement a defense mechanism against adversarial examples by training a new model on adversarial as well as clean images, and to evaluate the effectiveness of this defense method

2 Procedure

2.1 Environment Setup

For this project, Google Colaboratory (Google LLC, 2019) was selected as a convenient development environment suitable for smaller-scale deep learning research. Google Colaboratory is a cloud-based platform designed for data science which allows running Python code interactively with the option of GPU acceleration. In Google Colaboratory notebooks, code is organised into individually-run cells, below which any output they generate is displayed. This allows for an incremental development cycle in which changes do not necessitate all code being re-run. Code cells may be executed by clicking the 'Play' icon on their left-hand edge or using the keyboard shortcut 'Ctrl+Enter'. New code cells may be created by clicking the button below the menu bar labelled 'CODE'. Alternatively, to run a code cell and move to the next cell, creating a new one if none exists, the keyboard shortcut 'Shift+Enter' may be used. It should be noted that each block of code given in this procedure was executed in its own cell. Figure 1 shows an example Google Colaboratory notebook.

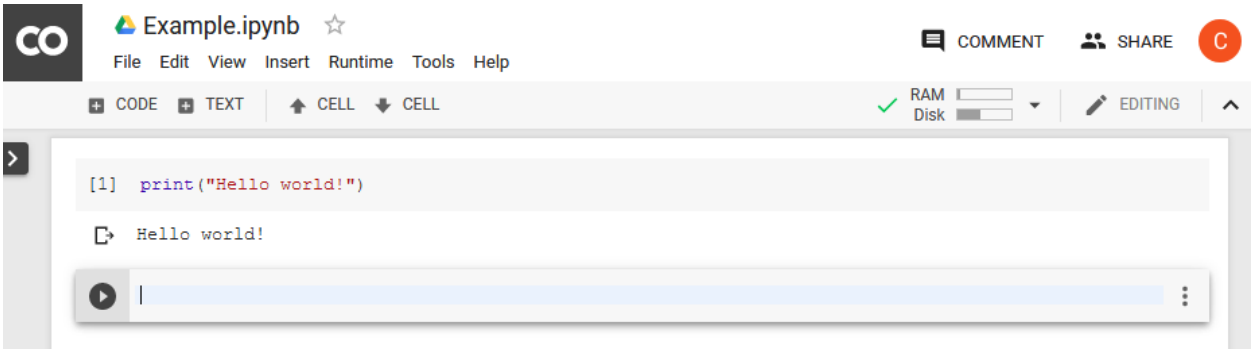

Figure 1: A Google Colaboratory notebook

Additionally, the fastai library was chosen to provide deep learning functionality. fastai is a high-level deep learning library for Python built on top of the lower-level PyTorch library, and was chosen because it allows excellent results to be achieved in very few lines of code.

Moreover, since it is built on top of PyTorch, this lower-level library may be used when more control is required.

The first step in setting up the environment was to create a Python 3 notebook in Google Colaboratory by navigating to https://colab.research.google.com, signing in with a Google account and selecting 'New Python 3 notebook' from the 'File' menu. Additionally, since GPU acceleration is a necessity for efficiently training deep learning algorithms, this was enabled by selecting 'Change runtime type' from the 'Runtime' menu and setting the 'Hardware accelerator' option to 'GPU'.

Next, the following commands were executed to install the fastai library and the official Kaggle API. The Kaggle API provides a command line tool which allows downloading datasets from Kaggle, an online platform which hosts data science competitions. This was required as the dataset used in this project was sourced from Kaggle.

```
! curl -s https://course.fast.ai/setup/colab | bash
! pip install kaggle
```

The '!' character signifies to Google Colaboratory that the command following it should be run using the bash shell instead of the Python interpreter.

Some additional setup was required to allow the Kaggle API to authenticate with Kaggle's servers. This required a Kaggle account which was created at https://www.kaggle.com. Next, an API token was created by navigating to https://www.kaggle.com/<username>/account (substituting in the relevant username) and clicking 'Create API Token'. The contents of the 'kaggle.json' file which was downloaded as a result were substituted into the following code to save the relevant Kaggle credentials to the Google Colaboratory notebook environment.

```
! mkdir ~/.kaggle/
! echo '<contents_of_kaggle.json>' > ~/.kaggle/kaggle.json
! chmod 600 ~/.kaggle/kaggle.json
```

It is important to note that a notebook containing valid Kaggle credentials should not be shared.

Finally, the following code was run to import all the libraries required for this project.

```
from fastai.vision import *
from fastai.metrics import error_rate

import torch.nn as nn
import torch.nn.functional as F
import torchvision.transforms.functional as TF

from PIL import Image
from tqdm import tqdm_notebook as tqdm
```

2.2 Preparing the Dataset

The next step was to prepare the image classification dataset which would be used throughout this project. The chosen dataset was the Plant Seedlings Dataset (Giselsson *et al*, 2017), which consists of images of 12 different species of plant seedlings. The plant species in this dataset include both weeds and crops, making the problem of classifying them analogous to that of discriminating between benign and malicious content in practical computer security-related scenarios. This dataset was selected because, while the problem of accurately classifying multiple similar categories is far from trivial, it functioned as a sufficiently simple example for demonstrating the process of training a deep learning algorithm and generating adversarial examples. The Plant Seedlings Dataset is available on Kaggle, which is where it was obtained for this project.

Firstly, the following code was run to define the path where the dataset files would be stored.

```
dataset_path = Config.data_path()/'seedlings'
dataset_path.mkdir(parents=True, exist_ok=True)
```

Before the dataset could be downloaded, the rules of the Kaggle competition from which the data was sourced had to be accepted. This was done by navigating to https://www.kaggle.com/c/plant-seedlings-classification/rules, signing in with the relevant Kaggle account, and confirming acceptance of the rules under the 'Rules' tab. Once this had been done, the following commands were run to download and extract the dataset to the location defined above. The dataset as extracted was organised in folders by category.

```
! kaggle competitions download -c plant-seedlings-classification \\
  -f train.zip -p {dataset_path}
! unzip -q -n {dataset_path}/train.zip -d {dataset_path}
```

To define the location of the training set, on which the deep learning algorithm(s) would be trained, and to display the categories present in this training set, the following code was run.

```
train_path = dataset_path/'train'
categories = sorted([f.name for f in train_path.ls()]); categories
```

The output of this code is shown below.

```
['Black-grass',
 'Charlock',
 'Cleavers',
 'Common Chickweed',
 'Common wheat',
 'Fat Hen',
 'Loose Silky-bent',
 'Maize',
 'Scentless Mayweed',
 'Shepherds Purse',
 'Small-flowered Cranesbill',
 'Sugar beet']
```

Next, a function was defined to display the number of items of each category in a particular path.

```
def show_num_samples(path):
  for category in categories:
    print('{0:<25} {1:>15}'.format(category,
      str(len((path/category).ls())) + " samples"))
```

This was then used to investigate how many samples existed for each category in the current training set, which at this point consisted of the entire downloaded dataset, by running the following line of code.

```
show_num_samples(train_path)
```

The output of this command is listed below, and shows that there existed approximately 200-600 samples for each category.

```
Black-grass                   263 samples
Charlock                      390 samples
Cleavers                      287 samples
Common Chickweed              611 samples
Common wheat                  221 samples
Fat Hen                       475 samples
Loose Silky-bent              654 samples
Maize                         221 samples
Scentless Mayweed             516 samples
Shepherds Purse               231 samples
Small-flowered Cranesbill     496 samples
Sugar beet                    385 samples
```

As mentioned above, at this point only a training set existed, with no validation set on which the performance of the trained algorithm(s) could be evaluated. The following code was used to define the location of the validation set to be generated and then to create this folder as well as sub-folders for each category.

```
valid_path = dataset_path/'valid'
valid_path.mkdir(exist_ok=True)
for category in categories:
  (valid_path/category).mkdir(exist_ok=True)
```

To create a validation set, the following code was used to randomly decide whether to move each sample from the training set to the validation set. There was a 30% chance each item would be moved, meaning approximately 30% of the training set was moved to the validation set. The random seed was set beforehand to ensure the splitting of the dataset was reproducible.

```
random.seed(42)
for category in categories:
  for file in (train_path/category).ls():
    if random.randint(1, 10) >= 8:
      shutil.move(file, valid_path/category/file.name)
```

The following command was then used to determine how many samples existed in the newly created validation set.

```
show_num_samples(valid_path)
```

The output of this command is shown below.

```
Black-grass                  73 samples
Charlock                    111 samples
Cleavers                     96 samples
Common Chickweed            187 samples
Common wheat                 74 samples
Fat Hen                     156 samples
Loose Silky-bent            197 samples
Maize                        75 samples
Scentless Mayweed           137 samples
Shepherds Purse              61 samples
Small-flowered Cranesbill   146 samples
Sugar beet                  126 samples
```

Next, the following command was used to check how many samples remained in the training set.

```
show_num_samples(train_path)
```

The output of this command is shown below.

```
Black-grass                 190 samples
Charlock                    279 samples
Cleavers                    191 samples
Common Chickweed            424 samples
Common wheat                147 samples
Fat Hen                     319 samples
Loose Silky-bent            457 samples
Maize                       146 samples
Scentless Mayweed           379 samples
Shepherds Purse             170 samples
Small-flowered Cranesbill   350 samples
Sugar beet                  259 samples
```

Once the dataset had been split, the following code was used to load the training and validation sets in a format suitable for use by the fastai library, specifying a batch size of 48.

```
clean_data = ImageDataBunch.from_folder(dataset_path, train='train',
    valid='valid', ds_tfms=get_transforms(), size=224, bs=48)
clean_data.normalize(([0.5, 0.5, 0.5], [0.5, 0.5, 0.5]));
```

Finally, the following command was used to visualize the loaded dataset by displaying the first few samples it contains.

```
clean_data.show_batch(rows=3, figsize=(7,6))
```

The output of this command is shown in Figure 2.

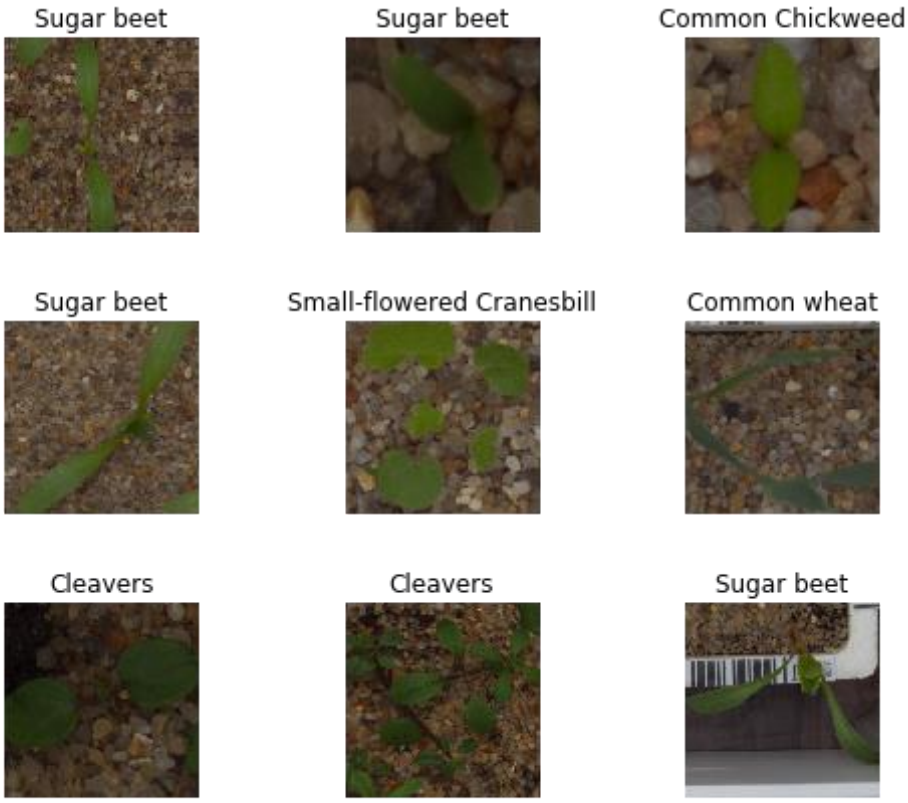

Figure 2: Visualizing the dataset

2.3 Training a Classification Model

After the data was loaded, the next task was to train a deep learning model to classify the different categories of image present in the dataset. The model used was a ResNet-34-based model with pretrained parameters. The ResNet architecture is a powerful deep neural network architecture used for image recognition tasks, and the '34' in the name refers to the number of layers it contains (He *et al.*, 2016). The pretrained parameters result from training a ResNet-34-based image classification model on the ImageNet dataset, a database of millions of images of everyday objects (ImageNet, 2019). The fact that the model had been pretrained on an image classification dataset meant that it would be quicker and easier to train the model for the specific task this project entails.

Firstly, the following command was used to create a ResNet-34-based model with pretrained parameters.

```
classifier_resnet34 = cnn_learner(clean_data, models.resnet34,
                                  metrics=error_rate)
```

Next, the following code was used to train this model for 16 epochs. The output of this command is shown in Appendix A (Figure 6).

```
classifier_resnet34.fit_one_cycle(16)
```

After it had been trained, the following command was run to save the model's parameters to a file called 'resnet34'.

```
classifier_resnet34.save('resnet34')
```

To easily check the accuracy of a model based on its confusion matrix (a matrix where one axis refers to true labels, the other refers to predicted labels, and the element indicates how many times the corresponding prediction occurred) the following function was defined.

```
def accuracy_from_confusion_matrix(confusion_matrix):
  return confusion_matrix.diagonal().sum() / confusion_matrix.sum()
```

To evaluate the accuracy of the trained model, this function was used in the following code.

```
interp = ClassificationInterpretation.from_learner(classifier_resnet34)
accuracy_from_confusion_matrix(interp.confusion_matrix())
```

The results of this command are shown below.

```
0.9305072967338429
```

This shows that the accuracy of the trained model was approximately 93.1%.

2.4 Generating Untargeted White-Box Adversarial Examples

After the model had been trained, the next step was to generate a set of untargeted white-box adversarial examples. The following function was defined to apply an adversarial perturbation to an image using the fast gradient sign method based on a given model. No targeting of the adversarial examples was done at this stage, with the aim being simply to achieve any false classification.

```
def fast_gradient_sign(img, true_label, perturbation_size, model):
  img.requires_grad = True
  out = model(img)
  loss = F.cross_entropy(out, true_label)
  loss.backward()
  gradient_sign = img.grad.sign()
  img_adv = img + perturbation_size * gradient_sign
  return img_adv
```

This function was used in the following code to create a copy of the validation set where each image has been perturbed according to the fast gradient sign method. The size of the perturbation of each numeric element was specified as 0.001. These were white-box adversarial examples as the adversarial perturbations applied were based on the exact target model.

```
adv_untargeted_valid_path = dataset_path/'valid_adv_untargeted'
adv_untargeted_valid_path.mkdir(exist_ok=True)
for i, category in enumerate(categories):
  (adv_untargeted_valid_path/category).mkdir(exist_ok=True)
  for file in tqdm((valid_path/category).ls(), desc=category):
    img = Image.open(file).convert('RGB')
    img_as_tensor = TF.normalize(TF.to_tensor(TF.resize(img, 224)),
                                 (0.5, 0.5, 0.5),
                                 (0.5, 0.5, 0.5))
    img_adv_as_tensor = fast_gradient_sign(img_as_tensor.unsqueeze(0).cuda(),
                                           torch.tensor([i]).cuda(),
                                           0.001,
                                           classifier_resnet34.model)
    img_adv = TF.to_pil_image(
                  img_adv_as_tensor.squeeze().detach().cpu() * 0.5 + 0.5)
    img_adv.save(adv_untargeted_valid_path/category/file.name)
```

Once generated, this dataset of adversarial examples was loaded using the following code.

```
data_adv_untargeted = ImageDataBunch.from_folder(
                          dataset_path,
                          train='train',
                          valid = adv_untargeted_valid_path.name,
                          ds_tfms=get_transforms(),
                          size=224,
                          bs=48)
data_adv_untargeted.normalize(([0.5, 0.5, 0.5], [0.5, 0.5, 0.5]));
```

To ensure they were not noticeably different from the original samples, the following code was run to visualize the generated adversarial examples.

```
data_adv_untargeted.show_batch(rows=3,
                               figsize=(7,6),
                               ds_type=DatasetType.Valid)
```

The output of this command is shown in Figure 3.

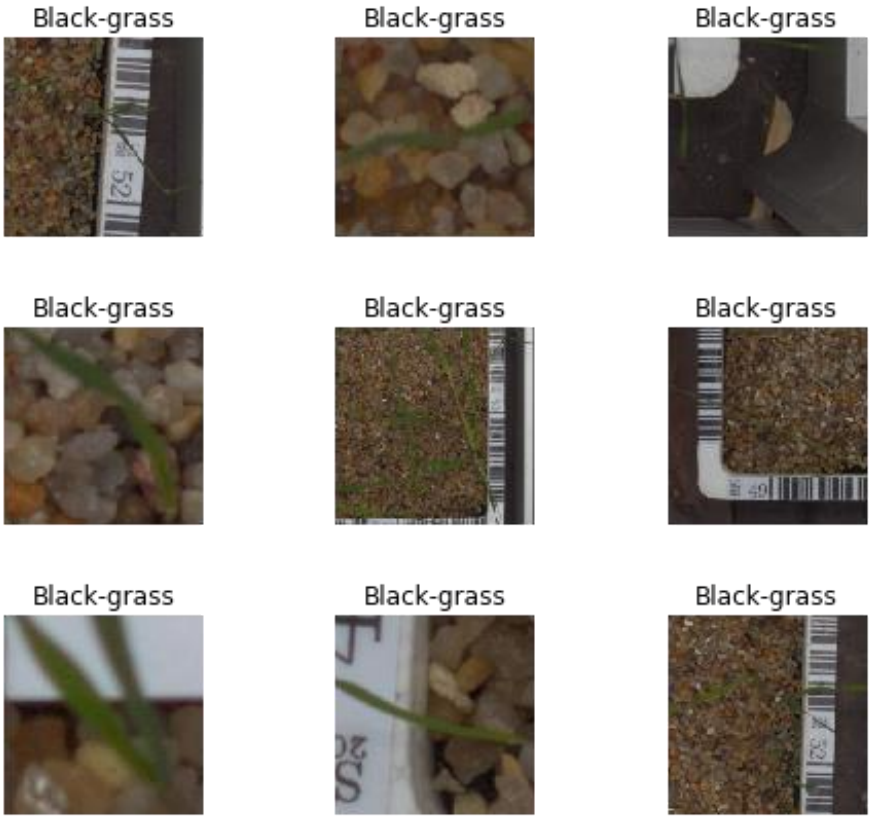

Figure 3: Untargeted white-box adversarial examples

Next, their effectiveness was evaluated. The following code was run to create a copy of the trained model which used the adversarial validation set.

```
classifier_resnet34_adv_untargeted = cnn_learner(data_adv_untargeted,
                                                 models.resnet34,
                                                 metrics=error_rate,
                                                 pretrained=False)
classifier_resnet34_adv_untargeted.load('resnet34');
```

Finally, the following code was run to test the accuracy of this model on the adversarial validation set.

```
interp = ClassificationInterpretation.from_learner(
             classifier_resnet34_adv_untargeted)
accuracy_from_confusion_matrix(interp.confusion_matrix())
```

The output of this code is shown below.

```
0.011118832522585128
```

This reveals that the trained model correctly classified these adversarial examples around 1.1% of the time.

2.5 Generating Targeted White-Box Adversarial Examples

The next task was to create a set of adversarial examples with specific intended misclassifications. To achieve this, the following function was defined to apply a targeted perturbation to an input image using the fast gradient sign method, again based on a given model.

```
def fast_gradient_sign_targeted(img, target_label, perturbation_size, model):
  img.requires_grad = True
  out = model(img)
  loss = F.cross_entropy(out, target_label)
  loss.backward()
  gradient_sign = img.grad.sign()
  img_adv = img - perturbation_size * gradient_sign
  return img_adv
```

For a more powerful effect, the following function was defined to iteratively apply a series of targeted adversarial perturbations.

```
def fast_gradient_sign_targeted_it(img, target_label, perturbation_size,
                                   iterations, model):
  for _ in range(iterations):
    img_adv = fast_gradient_sign_targeted(img, target_label,
                                          perturbation_size, model)
    img = img_adv.detach()
  return img_adv
```

Next, the following code was run to randomly choose a target category for each true category. The indices of the generated list referred to the true category while the elements referred to the target category.

```
random.seed(12)
target_categories = []
for i in range(len(categories)):
  other_categories = list(range(len(categories))
  other_categories.remove(i)
  target_categories.append(random.choice(other_categories))
```

After target classes had been chosen, the following code was run to create an adversarial validation set with the iterative fast gradient sign method using these target categories and the parameters of the trained model.

```
adv_targeted_valid_path = dataset_path/'valid_adv_targeted'
adv_targeted_valid_path.mkdir(exist_ok=True)
for i, category in enumerate(categories):
  (adv_targeted_valid_path/category).mkdir(exist_ok=True)
  print(category + ' -> ' + categories[target_categories[i]])
  for file in tqdm((valid_path/category).ls()):
    img = Image.open(file).convert('RGB')
    img_as_tensor = TF.normalize(TF.to_tensor(TF.resize(img, 224)),
                                 (0.5, 0.5, 0.5),
                                 (0.5, 0.5, 0.5))
    img_adv_as_tensor = fast_gradient_sign_targeted_it(
                            img_as_tensor.unsqueeze(0).cuda(),
                            torch.tensor([target_categories[i]]).cuda(),
                            0.0002,
                            40,
                            classifier_resnet34.model)
    img_adv = TF.to_pil_image(
        img_adv_as_tensor.squeeze().detach().cpu() * 0.5 + 0.5)
    img_adv.save(adv_targeted_valid_path/category/file.name)
```

Once it had been generated, the following code was run to load the targeted adversarial validation set.

```
data_adv_targeted = ImageDataBunch.from_folder(
                                    dataset_path,
                                    train='train',
                                    valid = adv_targeted_valid_path.name,
                                    ds_tfms=get_transforms(),
                                    size=224,
                                    bs=48)
data_adv_targeted.normalize(([0.5, 0.5, 0.5], [0.5, 0.5, 0.5]));
```

As with the untargeted adversarial examples, the following code was run to visualize the targeted adversarial validation set to ensure it was not noticeably perturbed.

```
data_adv_targeted.show_batch(rows=3,
                             figsize=(7,6),
                             ds_type=DatasetType.Valid)
```

The output of this code is shown in Figure 4.

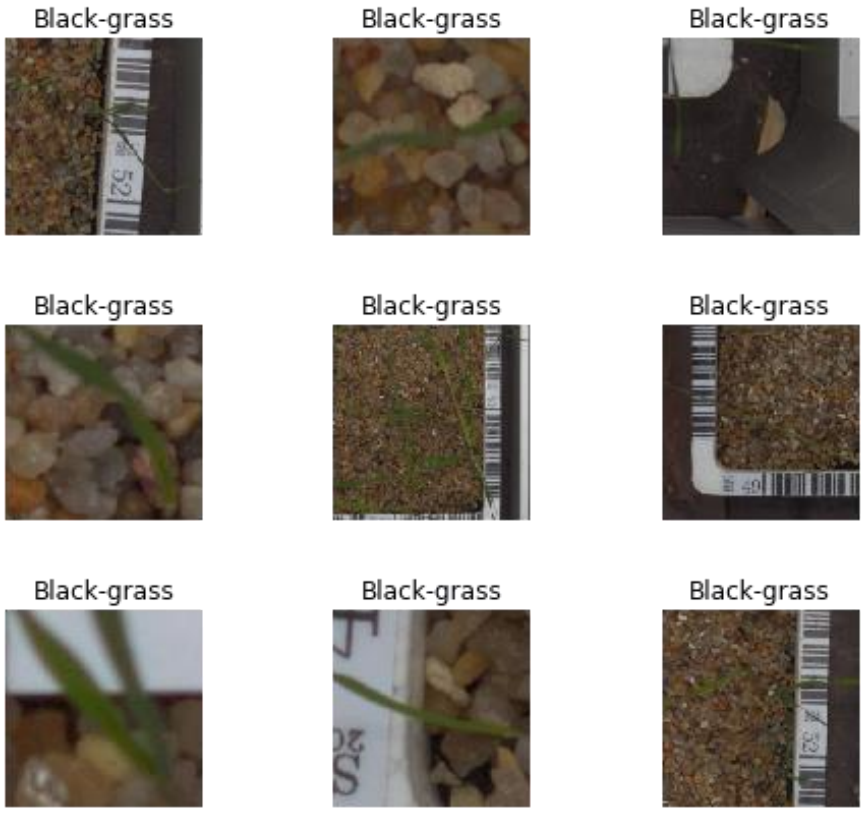

Figure 4: Targeted white-box adversarial examples

To allow testing these samples' effectiveness, a copy of the original model which used the targeted adversarial validation set was created using the following code.

```
classifier_resnet34_adv_targeted = cnn_learner(data_adv_targeted,
                                               models.resnet34,
                                               metrics=error_rate,
                                               pretrained=False)
classifier_resnet34_adv_targeted.load('resnet34');
```

The following code was then run to assess the accuracy of this model.

```
interp = ClassificationInterpretation.from_learner(
             classifier_resnet34_adv_targeted)
accuracy_from_confusion_matrix(interp.confusion_matrix())
```

The output of this code is shown below.

```
0.004169562195969423
```

This output reveals that the model classified the targeted adversarial examples correctly around 0.4% of the time. However, to truly assess their effectiveness, the accuracy in terms of the adversarial target labels had to be determined. The following function was defined to calculate this measure using a confusion matrix and the list of target labels.

```
def targeted_accuracy(confusion_matrix):
  hits = 0
  for i in range(12):
    hits += confusion_matrix[i,target_categories[i]]
  return hits / confusion_matrix.sum()
```

This function was then run to calculate the effectiveness of the targeted adversarial examples.

```
targeted_accuracy(interp.confusion_matrix())
```

The output of this command is shown below, revealing that the model classified the targeted adversarial examples according to their target label approximately 99.4% of the time.

```
0.9944405837387075
```

2.6 Assessing the Transferability of the Generated Adversarial Examples

The adversarial examples generated thus far were white-box examples generated using the known parameters of the target model. To test their transferability, and, by extension, their suitability for black-box attacks, these examples were tested on models of different architectures.

The first step was to create a model based on the VGG16 architecture (Simonyan and Zisserman, 2014) and pretrained on ImageNet using the following code.

```
classifier_vgg16 = cnn_learner(clean_data,
                               models.vgg16_bn,
                               metrics=error_rate)
```

Next, the following code was used to train this model on the Plant Seedlings Dataset for 8 epochs. The output of this code is shown in Appendix A (Figure 7).

```
classifier_vgg16.fit_one_cycle(8)
```

The parameters of this model were then saved using the following command.

```
classifier_vgg16.save('vgg16')
```

Once trained, the following code was run to determine the accuracy of the VGG16 model on the clean dataset.

```
interp = ClassificationInterpretation.from_learner(classifier_vgg16)
accuracy_from_confusion_matrix(interp.confusion_matrix())
```

The output produced by this code is shown below.

```
0.9186935371785963
```

This reveals that the accuracy of the VGG16-based model on the standard validation set was approximately 91.9%.

Next, copies of the VGG16 model which used the untargeted and targeted adversarial validation sets were created using the following code.

```
classifier_vgg16_adv_untargeted = cnn_learner(data_adv_untargeted,
                                              models.vgg16_bn,
                                              metrics=error_rate,
                                              pretrained=False)
classifier_vgg16_adv_untargeted.load('vgg16');

classifier_vgg16_adv_targeted = cnn_learner(data_adv_targeted,
                                            models.vgg16_bn,
                                            metrics=error_rate,
                                            pretrained=False)
classifier_vgg16_adv_targeted.load('vgg16');
```

The VGG16 model was then assessed on the untargeted adversarial validation set using the following code.

```
interp = ClassificationInterpretation.from_learner(
             classifier_vgg16_adv_untargeted)
accuracy_from_confusion_matrix(interp.confusion_matrix())
```

The output of this code is shown below, revealing that this model correctly classified the untargeted adversarial examples approximately 82.1% of the time.

```
0.8207088255733148
```

This same step was then performed for the targeted adversarial validation set using the following code.

```
interp = ClassificationInterpretation.from_learner(
             classifier_vgg16_adv_targeted)
accuracy_from_confusion_matrix(interp.confusion_matrix())
```

The output of this code is shown below and reveals that the model correctly classified the targeted adversarial examples approximately 87.4% of the time.

```
0.8735232800555942
```

Additionally, the following code was run to calculate how often the VGG16-based model classified the targeted adversarial samples according to their target label.

```
targeted_accuracy(interp)
```

The output of this command, listed below, shows that the model misclassified the targeted examples as desired approximately 1.9% of the time.

```
0.019457956914523976
```

As well as a VGG16-based model, a new model using the DenseNet-121 architecture (Huang *et al*, 2017) and pretrained on ImageNet was also created, using the following code.

```
classifier_densenet121 = cnn_learner(clean_data,
                                     models.densenet121,
                                     metrics=error_rate)
```

This model was then trained for 8 epochs using the following command. The output of this command is shown in Appendix A (Figure 8).

```
classifier_densenet121.fit_one_cycle(8)
```

Next, the parameters of the DenseNet-121 model were saved using the following code.

```
classifier_densenet121.save('dn121');
```

After the model was trained, the following code was run to determine its accuracy on the clean validation set.

```
interp = ClassificationInterpretation.from_learner(classifier_densenet121)
accuracy_from_confusion_matrix(interp.confusion_matrix())
```

The output of this code is listed below and shows that the validation-set accuracy of this model on the clean data was around 94.4%.

```
0.9444058373870744
```

As with the VGG16-based model, copies of the DenseNet-121 model which used the untargeted and targeted adversarial validation sets were then created using the following code.

```
classifier_densenet121_adv_untargeted = cnn_learner(data_adv_untargeted,
                                                    models.densenet121,
                                                    metrics=error_rate,
                                                    pretrained=False)
classifier_densenet121_adv_untargeted.load('dn121');

classifier_densenet121_adv_targeted = cnn_learner(data_adv_targeted,
                                                  models.densenet121,
                                                  metrics=error_rate,
                                                  pretrained=False)
classifier_densenet121_adv_targeted.load('dn121');
```

Next, the following code was used to evaluate the DenseNet-121-based model's accuracy on the untargeted adversarial validation set.

```
interp = ClassificationInterpretation.from_learner(
             classifier_densenet121_adv_untargeted)
accuracy_from_confusion_matrix(interp.confusion_matrix())
```

The output of this code is shown below, and reveals that this model classified the untargeted adversarial examples correctly approximately 84.6% of the time.

```
0.8457261987491314
```

The DenseNet-121-based model's accuracy on the targeted adversarial validation set was then tested using the following code.

```
interp = ClassificationInterpretation.from_learner(
             classifier_densenet121_adv_targeted)
accuracy_from_confusion_matrix(interp.confusion_matrix())
```

The output of this code is listed below, and shows that this model's accuracy on the targeted adversarial data was approximately 91.9%.

```
0.9193884642112579
```

Finally, the following command was used to calculate how often the targeted samples were classified as desired.

```
targeted_accuracy(interp)
```

The output of this command is shown below.

```
0.011118832522585128
```

This reveals that the DenseNet-121 model classified the targeted adversarial examples according to their target class approximately 1.1% of the time.

2.7 Generating More Transferable Adversarial Examples

With the aim of creating a more transferable set of adversarial examples, an ensemble of multiple models was created and used to generate a new adversarial validation set using the fast gradient sign method. Essentially, an ensemble model passes input data through multiple models and averages their outputs. The idea behind this method was that an adversarial example which fooled an ensemble of several models would likely fool at least most of the individual models the ensemble was composed of. To create an ensemble of the three models trained so far, the ResNet-34-based model, the VGG16-based model and the DenseNet-121-based model, the following code was run.

```
class EnsembleNet(nn.Module):

  def __init__(self):
    super().__init__()
    self.resnet34 = classifier_resnet34.model
    self.vgg16 = classifier_vgg16.model
    self.densenet121 = classifier_densenet121.model

  def forward(self, input):
    out_resnet34 = self.resnet34(input)
    out_vgg16 = self.vgg16(input)
    out_densenet121 = self.densenet121(input)
    return (out_resnet34 + out_vgg16 + out_densenet121) / 3

ensemble_model = EnsembleNet()
```

Next, this ensemble model was used in generating a new untargeted adversarial validation set, using the following code.

```
adv_untargeted_e_valid_path = dataset_path/'valid_adv_untargeted_e'
adv_untargeted_e_valid_path.mkdir(exist_ok=True)
for i, category in enumerate(categories):
  (adv_untargeted_e_valid_path/category).mkdir(exist_ok=True)
  for file in tqdm((valid_path/category).ls(), desc=category):
    img = Image.open(file).convert('RGB')
    img_as_tensor = TF.normalize(TF.to_tensor(TF.resize(img, 224)),
                                 (0.5, 0.5, 0.5),
                                 (0.5, 0.5, 0.5))
    img_adv_as_tensor = fast_gradient_sign(img_as_tensor.unsqueeze(0).cuda(),
                                           torch.tensor([i]).cuda(),
                                           0.0005,
                                           ensemble_model)
    img_adv = TF.to_pil_image(
        img_adv_as_tensor.squeeze().detach().cpu() * 0.5 + 0.5)
    img_adv.save(adv_untargeted_e_valid_path/category/file.name)
```

The following code was then used to load this new dataset.

```
data_adv_untargeted_e = ImageDataBunch.from_folder(
                            dataset_path,
                            train='train',
                            valid=adv_untargeted_e_valid_path.name,
                            ds_tfms=get_transforms(),
                            size=224,
                            bs=48)
data_adv_untargeted_e.normalize(([0.5, 0.5, 0.5], [0.5, 0.5, 0.5]));
```

Next, the following code was run to create copies of the Resnet34-based model, the VGG16-based model and the Densenet121-based model, each using the newly created adversarial validation set.

```
classifier_resnet34_adv_untargeted_e = cnn_learner(data_adv_untargeted_e,
                                                   models.resnet34,
                                                   metrics=error_rate,
                                                   pretrained=False)
classifier_resnet34_adv_untargeted_e.load('resnet34');

classifier_vgg16_adv_untargeted_e = cnn_learner(data_adv_untargeted_e,
                                                models.vgg16_bn,
                                                metrics=error_rate,
                                                pretrained=False)
classifier_vgg16_adv_untargeted_e.load('vgg16');

classifier_densenet121_adv_untargeted_e = cnn_learner(data_adv_untargeted_e,
                                                      models.densenet121,
                                                      metrics=error_rate,
                                                      pretrained=False)
classifier_densenet121_adv_untargeted_e.load('dn121');
```

To evaluate the classification accuracy of the ResNet-34 model on the ensemble-based untargeted adversarial validation set, the following code was run.

```
interp = ClassificationInterpretation.from_learner(
             classifier_resnet34_adv_untargeted_e)
accuracy_from_confusion_matrix(interp.confusion_matrix())
```

The output of this code is listed below, and shows that the ResNet-34-based model correctly classified the ensemble-based untargeted adversarial samples approximately 6.9% of the time.

```
0.06949270326615706
```

Next, the same step was performed for the VGG16-based model using the following code.

```
interp = ClassificationInterpretation.from_learner(
             classifier_vgg16_adv_untargeted_e)
accuracy_from_confusion_matrix(interp.confusion_matrix())
```

The output of this code is shown below, revealing that the VGG16-based model correctly classified the ensemble-based untargeted adversarial samples 0% of the time.

```
0.0
```

Again, the same step was performed for the DenseNet121-based model using the following code.

```
interp = ClassificationInterpretation.from_learner(
             classifier_densenet121_adv_untargeted_e)
accuracy_from_confusion_matrix(interp.confusion_matrix())
```

The output of this code is listed below, and reveals that the DenseNet121-based model correctly classified the ensemble-based untargeted adversarial samples approximately 5.5% of the time.

```
0.05489923558026407
```

Additionally, a new model was also created, one using the ResNet-50 architecture, using the following code. This would be used to test the effectiveness of the ensemble-based adversarial examples on models not included in the ensemble.

```
classifier_resnet50 = cnn_learner(clean_data,
                                  models.resnet50,
                                  metrics=error_rate)
```

The following code was then run to train this model on the clean dataset for 6 epochs. The output of this code is shown in Appendix A (Figure 9).

```
classifier_resnet50.fit_one_cycle(6)
```

The parameters of this model were then saved with the following command.

```
classifier_resnet50.save('resnet50')
```

Two copies of this model were then created, one using the ResNet-34-based untargeted adversarial validation set and one using the ensemble-based untargeted adversarial validation set. This was achieved with the following code.

```
classifier_resnet50_adv_untargeted = cnn_learner(data_adv_untargeted,
                                                 models.resnet50,
                                                 metrics=error_rate,
                                                 pretrained=False)
classifier_resnet50_adv_untargeted.load('resnet50');

classifier_resnet50_adv_untargeted_e = cnn_learner(data_adv_untargeted_e,
                                                   models.resnet50,
                                                   metrics=error_rate,
                                                   pretrained=False)
classifier_resnet50_adv_untargeted_e.load('resnet50');
```

The ResNet-50-based model's accuracy on the ResNet-34-based untargeted adversarial validation set was then evaluated using the following code.

```
interp = ClassificationInterpretation.from_learner(
             classifier_resnet50_adv_untargeted)
accuracy_from_confusion_matrix(interp.confusion_matrix())
```

The output of this code is shown below, and reveals that the ResNet50-based model correctly classified the ResNet-34-based untargeted adversarial samples approximately 78.5% of the time.

```
0.7852675469075747
```

Finally, the accuracy of the ResNet-50-based model on the ensemble-based untargeted adversarial validation set was calculated using the following code.

```
interp = ClassificationInterpretation.from_learner(
             classifier_resnet50_adv_untargeted_e)
accuracy_from_confusion_matrix(interp.confusion_matrix())
```

The output of this code is listed below, and shows that the ResNet-50-based model correctly classified the ensemble-based untargeted adversarial samples approximately 73.2% of the time.

```
0.7324530924252953
```

2.8 Countermeasures: Training on Adversarial Examples

In an attempt to create a model more resilient to adversarial examples, a new ResNet-34-based model was created and trained on a training set which included both clean and adversarial images, the idea being to explicitly train the model to correctly classify adversarial examples. Firstly, the following code was run to create a new training set which included two copies of each image, one clean and one with untargeted adversarial perturbations applied.

```
adv_untargeted_train_path = dataset_path/'train_adv_untargeted'
adv_untargeted_train_path.mkdir(exist_ok=True)
for i, category in enumerate(categories):
  (adv_untargeted_train_path/category).mkdir(exist_ok=True)
  for file in tqdm((train_path/category).ls(), desc=category):
    shutil.copyfile(file, adv_untargeted_train_path/category/file.name)
    img = Image.open(file).convert('RGB')
    img_as_tensor = TF.normalize(TF.to_tensor(TF.resize(img, 224)),
                                 (0.5, 0.5, 0.5),
                                 (0.5, 0.5, 0.5))
    img_adv_as_tensor = fast_gradient_sign(img_as_tensor.unsqueeze(0).cuda(),
                                           torch.tensor([i]).cuda(),
                                           0.0005,
                                           classifier_resnet34.model)
    img_adv = TF.to_pil_image(
          img_adv_as_tensor.squeeze().detach().cpu() * 0.5 + 0.5)
    img_adv.save(adv_untargeted_train_path/category/('adv'+file.name))
```

This data was then loaded with the following code. The dataset created using this code used the mixed training set and the ResNet-34-based untargeted adversarial validation set.

```
data_train_adv_untargeted = ImageDataBunch.from_folder(
                                dataset_path,
                                train='train_adv_untargeted',
                                valid='valid_adv_untargeted',
                                ds_tfms=get_transforms(),
                                size=224,
                                bs=48)
data_train_adv_untargeted.normalize(([0.5, 0.5, 0.5], [0.5, 0.5, 0.5]));
```

Next, a ResNet-34 model which used this dataset was created with the following command.

```
classifier_resnet34_train_adv_untargeted = cnn_learner(
                                               data_train_adv_untargeted,
                                               models.resnet34,
                                               metrics=error_rate)
```

This model was then trained for 8 epochs with the following code. The output of this command is shown in Appendix A (Figure 10).

```
classifier_resnet34_train_adv_untargeted.fit_one_cycle(8)
```

To further improve the accuracy of this model, the following code was run to unfreeze the pretrained parameters and test a variety of learning rates and their effect on training.

```
classifier_resnet34_train_adv_untargeted.unfreeze()
classifier_resnet34_train_adv_untargeted.lr_find()
```

The results of the learning rate test were then plotted using the following command.

```
classifier_resnet34_train_adv_untargeted.recorder.plot()
```

The plot produced by this command is shown in Figure 5.

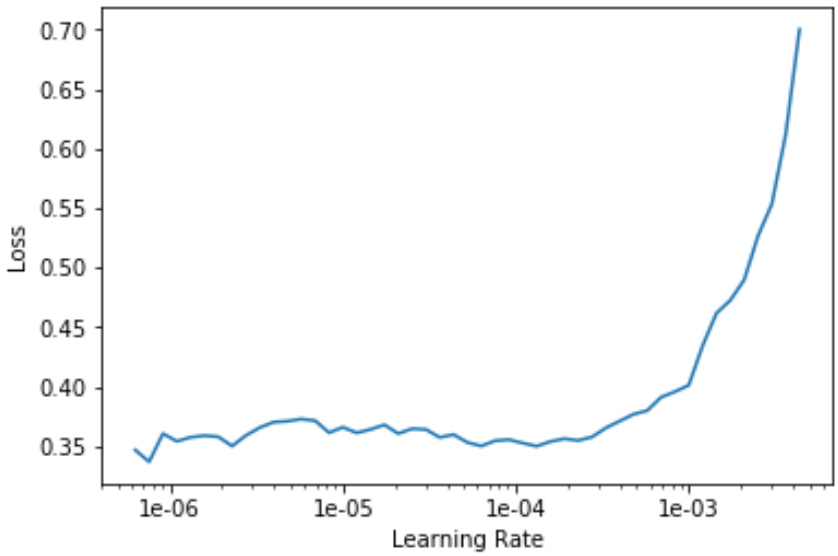

Figure 5: Plot of loss against learning rate

Learning rate ranges in which the loss is decreasing likely represented the best choices. Based on this plot, the model was then trained for a further 60 epochs with a learning rate between 0.000003 and 0.00005 depending on the layer using the following commands. Training was done in three stages due to a runtime error encountered in Google Colaboratory when training for 60 epochs in a single command. The output of these commands is shown in Appendix A (Figures 11 and 12).

```
classifier_resnet34_train_adv_untargeted.fit_one_cycle(20, slice(3e-6, 5e-5))
classifier_resnet34_train_adv_untargeted.fit_one_cycle(20, slice(3e-6, 5e-5))
classifier_resnet34_train_adv_untargeted.fit_one_cycle(20, slice(3e-6, 5e-5))
```

After the model had been trained, its parameters were saved using the following command.

```
classifier_resnet34_train_adv_untargeted.save('r34-adv')
```

To determine the effectiveness of the original adversarial validation set on this model, which has been trained using mixed data, the following code was run.

```
interp = ClassificationInterpretation.from_learner(
      classifier_resnet34_train_adv_untargeted)
accuracy_from_confusion_matrix(interp.confusion_matrix())
```

The output of this code is shown below, and reveals that the model trained on mixed data correctly classified samples in the adversarial validation set approximately 86.4% of the time.

```
0.8637943015983321
```

Additionally, the mixed training set was loaded with the standard validation set using the following code.

```
data_train_adv_untargeted_s = ImageDataBunch.from_folder(
                                    dataset_path,
                                    train='train_adv_untargeted',
                                    valid='valid',
                                    ds_tfms=get_transforms(),
                                    size=224,
                                    bs=48)
data_train_adv_untargeted_s.normalize(([0.5, 0.5, 0.5], [0.5, 0.5, 0.5]));
```

Next, a copy of the model was created which used the standard instead of the adversarial validation set was created using the following code.

```
classifier_resnet34_train_adv_untargeted_s = cnn_learner(
                                                data_train_adv_untargeted_s,
                                                models.resnet34,
                                                metrics=error_rate)
classifier_resnet34_train_adv_untargeted_s.load('r34-adv');
```

The accuracy of this model was then calculated with the following code.

```
interp = ClassificationInterpretation.from_learner(
    classifier_resnet34_train_adv_untargeted_s)
accuracy_from_confusion_matrix(interp.confusion_matrix())
```

The output of this code, listed below, reveals that the model trained on mixed data classified clean samples correctly approximately 96.7% of the time.

```
0.9666435024322446
```

Next, a new untargeted adversarial validation set, which used the model trained on mixed samples in its fast gradient sign calculations, was created using the following code.

```
adv_untargeted_t_valid_path = dataset_path/'valid_adv_untargeted_t'
adv_untargeted_t_valid_path.mkdir(exist_ok=True)
for i, category in enumerate(categories):
  (adv_untargeted_t_valid_path/category).mkdir(exist_ok=True)
  for file in tqdm((valid_path/category).ls(), desc=category):
    img = Image.open(file).convert('RGB')
    img_as_tensor = TF.normalize(TF.to_tensor(TF.resize(img, 224)),
                                 (0.5, 0.5, 0.5),
                                 (0.5, 0.5, 0.5))
    img_adv_as_tensor = fast_gradient_sign(
        img_as_tensor.unsqueeze(0).cuda(),
        torch.tensor([i]).cuda(),
        0.0005,
        classifier_resnet34_train_adv_untargeted.model)
    img_adv = TF.to_pil_image(
      img_adv_as_tensor.squeeze().detach().cpu() * 0.5 + 0.5)
    img_adv.save(adv_untargeted_t_valid_path/category/file.name)
```

This dataset was then loaded with the following code.

```
data_train_adv_untargeted_t = ImageDataBunch.from_folder(
                                    dataset_path,
                                    train='train',
                                    valid='valid_adv_untargeted_t',
                                    ds_tfms=get_transforms(),
                                    size=224,
                                    bs=48)
data_train_adv_untargeted_t.normalize(([0.5, 0.5, 0.5], [0.5, 0.5, 0.5]));
```

Next, a copy of the model which had been trained on mixed data but which used the second untargeted adversarial validation set, the one which used the model trained on mixed data to calculate the adversarial perturbations, was created using the following code.

```
classifier_resnet34_train_adv_untargeted_t = cnn_learner(
    data_train_adv_untargeted_t,
    models.resnet34,
    metrics=error_rate)
classifier_resnet34_train_adv_untargeted_t.load('r34-adv');
```

Finally, the accuracy of this model was evaluated using the following code.

```
interp = ClassificationInterpretation.from_learner(
             classifier_resnet34_train_adv_untargeted_t)
accuracy_from_confusion_matrix(interp.confusion_matrix())
```

The output of this code is shown below, revealing that the model trained on mixed data correctly classified adversarial examples which were generated using this same model approximately 20.2% of the time.

```
0.20222376650451704
```

3 Results

The results generated by the above procedure are summarized in the following tables. Table 1 gives each model trained a unique identifier and lists its architecture, the training set it used and the number of epochs it was trained for.

Table 1: Models trained

Model No.	Model Architecture	Training Set	Epochs
0	ResNet-34	Clean	16
1	VGG16	Clean	8
2	DenseNet-121	Clean	8
3	Ensemble of models 0, 1 and 2	N/A	N/A
4	ResNet-50	Clean	6
5	ResNet-34	Clean + adversarial, untargeted, generated using model 0	68

Table 2 lists the accuracies of each model as tested on a specific validation set. Targeted accuracy refers to the accuracy in terms of the adversarial target labels instead of the true labels.

Table 2: Model accuracies

Model No.	Validation Set	Accuracy	Targeted Accuracy
0	Clean	93.1%	N/A
0	Adversarial, untargeted, generated using model 0	1.1%	N/A
0	Adversarial, targeted, generated using model 0	0.4%	99.4%
1	Clean	91.9%	N/A
1	Adversarial, untargeted, generated using model 0	82.1%	N/A
1	Adversarial, targeted, generated using model 0	87.4%	1.9%
2	Clean	94.4%	N/A
2	Adversarial, untargeted, generated using model 0	84.6%	N/A
2	Adversarial, targeted, generated using model 0	91.9%	1.1%
0	Adversarial, untargeted, generated using model 3	6.9%	N/A
1	Adversarial, untargeted, generated using model 3	0%	N/A
2	Adversarial, untargeted, generated using model 3	5.5%	N/A
4	Adversarial, untargeted, generated using model 0	78.5%	N/A
4	Adversarial, untargeted, generated using model 3	73.2%	N/A
5	Adversarial, untargeted, generated using model 0	86.4%	N/A
5	Clean	96.7%	N/A
5	Adversarial, untargeted, generated using model 5	20.2%	N/A

4 Discussion

4.1 General Discussion

The first objective of this project was to train a deep learning algorithm to solve an image classification problem. Using a ResNet-34-based model pre-trained on ImageNet and further trained on the Plant Seedlings Dataset for 16 epochs, an accuracy of 93.1% was achieved on this dataset. While this result was probably not ground-breaking, it was far from poor. For comparison, since the Plant Seedlings Dataset has 12 categories, the accuracy of a random guess would be approximately 8.3%.

The next objective was to generate a set of untargeted white-box adversarial examples which were not noticeably visually changed, but which succeeded in fooling the trained classification model. Figure 3 shows a sample of the generated untargeted adversarial validation set, and these did not appear noticeably changed. Additionally, the trained model accurately classified them correctly only 1.1% of the time, as opposed to 93.1% of the time for clean samples. While there is room for improvement in this result, it still represents a clear success.

The third objective involved generating a set of adversarial examples with specific target labels while also maintaining their visual integrity. Figure 4 reveals that this adversarial dataset also appears normal, while the trained model classified these samples correctly only 0.4% of the time and according to their target labels 99.4% of the time, a high level of success. These samples were generated using an iterative version of the fast gradient sign method, which was likely a reason for their higher effectiveness than the untargeted adversarial examples.

The fourth objective was to test and improve the transferability of the adversarial examples. The VGG16 and DenseNet-121 models correctly classified the untargeted examples which were generated using the ResNet-34 model 82.1% and 84.6% of the time, respectively, as opposed to 91.9% and 94.4% of the time, respectively, for clean data. Additionally, the VGG16 model correctly classified the targeted adversarial examples generated using the ResNet-34 model 87.4% of the time, and according to their target labels 1.9% of the time, while the DenseNet-121 model classified these correctly 91.9% of the time and according to their target labels 1.1% of the time. This showed that the adversarial examples generated using the fast gradient sign method and the ResNet-34 model's parameters had very limited transferability at best, especially for targeted examples. This is similar to what was found by Liu *et al* (2016).

As for the generation of more transferable examples, the ResNet-34 model, the VGG16 model and the DenseNet-121 model correctly classified the untargeted adversarial examples generated using an ensemble model of all three of these 6.9%, 0% and 5.5% of the time, respectively. This showed that adversarial examples generated with the fast gradient sign method using an ensemble of models can achieve a reasonable although possibly varying level of effectiveness on each model in the ensemble, at least in this situation involving an ensemble of three models. While not a perfect result, this represents a great improvement in applicability over examples generated using a single model. However, these ensemble-based examples were classified correctly 73.2% of the time by a ResNet50-based model not included in the ensemble, compared to this model's 78.5% accuracy on the original examples generated using the ResNet-34 model. This indicates that examples generated using this ensemble method are much less effective on models not included in the ensemble, although possibly still more so than examples generated using a single model. Again, this is fairly similar to what Liu *et al* (2016) observed.

The final objective was to create a model more resilient to adversarial examples by training on these as well as clean data. The model trained on clean and adversarial images which were generated based on a model of the same architecture but trained on only clean data correctly classified the original adversarial examples 86.4% of the time and clean examples 96.7% of the time. This indicates that training on adversarial examples is indeed effective while also possibly benefiting the classification of regular samples. However, this model only correctly classified adversarial examples generated using this new model 20.2% of the time. This suggests that to be truly effective, training on adversarial examples would have to be a continuous process, with new examples being generated using the latest model and these being used to retrain the model. That is, it appears models trained on adversarial examples are primarily robust only to the exact variety of adversarial examples the model was trained on. This is similar to what was found by Kurakin, Goodfellow and Bengio (2016b) and Tramèr *et al* (2017).

4.2 Conclusions

Based on this project, it was concluded that both targeted and untargeted adversarial examples can be highly effective in fooling deep learning algorithms, but that those generated using the fast gradient sign method are primarily effective for white-box attacks. Examples generated using this method seem to be limited in terms of inter-model transferability, even when ensembles of models are used to generate them. Finally, training on adversarial examples was found to have the potential to greatly increase resilience to those of generated using the same method and model as those trained, and seemed to benefit accuracy in general, but the re-training process would have to be continuous in order to be continuously effective.

4.3 Future Work

There a few primary ways the work in this project could be extended given more time or resources. Firstly, the models trained in this project were trained for a limited number of epochs due to time constraints and thus may not have reached their full potential in terms of accuracy. Training the models more rigorously may lead to slightly different results.

Additionally, methods of generating adversarial examples other than the fast gradient sign method were not investigated in this project. These have the potential to address the fast gradient sign method's clear limitations in terms of transferability, as demonstrated by Liu *et al* (2016).

Finally, this project only explored adversarial examples in the area of image classification, and future work could extend this to other areas in which adversarial examples have been demonstrated, such as text classification (Alzantot *et al.*, 2018) or binary file classification (Kolosnjaji *et al.*, 2018).

References

Alzantot, M., Sharma, Y., Elgohary, A., Ho, B.J., Srivastava, M. and Chang, K.W. (2018) 'Generating natural language adversarial examples', *arXiv preprint*, arXiv:1804.07998.

Athalye, A., Engstrom, L., Ilyas, A. and Kwok, K. (2017) 'Synthesizing robust adversarial examples', *arXiv preprint*, arXiv:1707.07397.

Dettmers, T. (2015a) *Deep Learning in a Nutshell: History and Training*. Available at: https://devblogs.nvidia.com/deep-learning-nutshell-history-training/ (Accessed: 8 April 2019).

Dettmers, T. (2015b) *Deep Learning in a Nutshell: Core Concepts*. Available at: https://devblogs.nvidia.com/deep-learning-nutshell-core-concepts/ (Accessed: 8 April 2019).

Esman, G. (2017) *Splunk and Tensorflow for Security: Catching the Fraudster with Behavior Biometrics*. Available at: https://www.splunk.com/blog/2017/04/18/deep-learning-with-splunk-and-tensorflow-for-security-catching-the-fraudster-in-neural-networks-with-behavioral-biometrics.html (Accessed: 8 April 2019).

Eykholt, K., Evtimov, I., Fernandes, E., Li, B., Rahmati, A., Xiao, C., Prakash, A., Kohno, T. and Song, D. (2017) 'Robust physical-world attacks on deep learning models', *arXiv preprint*, arXiv:1707.08945.

Giselsson, T.M., Jørgensen, R.N., Jensen, P.K., Dyrmann, M. and Midtiby, H.S. (2017) 'A public image database for benchmark of plant seedling classification algorithms', *arXiv preprint*, arXiv:1711.05458.

Goodfellow, I.J., Shlens, J. and Szegedy, C. (2014) 'Explaining and harnessing adversarial examples', *arXiv preprint*, arXiv:1412.6572.

Google LLC (2019) *Google Colaboratory* [Computer program, Online]. Available at https://colab.research.google.com (Accessed: 22 April 2019).

He, K., Zhang, X., Ren, S. and Sun, J. (2016) 'Deep residual learning for image recognition', *Proceedings of the IEEE conference on computer vision and pattern recognition*, pp. 770-778.

Huang, G., Liu, Z., Van Der Maaten, L. and Weinberger, K.Q. (2017) 'Densely connected convolutional networks', *Proceedings of the IEEE conference on computer vision and pattern recognition,* pp. 4700-4708.

ImageNet (2019) *ImageNet.* Available at: http://www.image-net.org/about-overview (Accessed: 12 April 2019).

Kolosnjaji, B., Demontis, A., Biggio, B., Maiorca, D., Giacinto, G., Eckert, C. and Roli, F. (2018) 'Adversarial malware binaries: Evading deep learning for malware detection in executables', *2018 26th European Signal Processing Conference (EUSIPCO)*, pp. 533-537.

Kurakin, A., Goodfellow, I. and Bengio, S. (2016a) 'Adversarial examples in the physical world', *arXiv preprint*, arXiv:1607.02533.

Kurakin, A., Goodfellow, I. and Bengio, S. (2016b) 'Adversarial machine learning at scale', *arXiv preprint*, arXiv:1611.01236.

Kurakin, A., Goodfellow, I., Bengio, S., Dong, Y., Liao, F., Liang, M., Pang, T., Zhu, J., Hu, X., Xie, C. and Wang, J. (2018) 'Adversarial attacks and defences competition', *The NIPS'17 Competition: Building Intelligent Systems*, pp. 195-231.

Le, Q. and Schuster, M. (2016) *A Neural Network for Machine Translation, at Production Scale*. Available at: https://ai.googleblog.com/2016/09/a-neural-network-for-machine.html (Accessed: 8 April 2019).

LeCun, Y., Bengio, Y. and Hinton, G. (2015) 'Deep learning', *Nature*, 521(7553), pp. 436-444.

Liu, Y., Chen, X., Liu, C. and Song, D. (2016) 'Delving into transferable adversarial examples and black-box attacks', *arXiv preprint*, arXiv:1611.02770.

Munson, L. (2015) *Facebook's DeepFace facial recognition technology has human-like accuracy*. Available at: https://nakedsecurity.sophos.com/2015/02/06/facebooks-deepface-facial-recognition-technology-has-human-like-accuracy/ (Accessed: 8 Apr. 2019).

Nielsen, M. (2018) *A visual proof that neural nets can compute any function*. Available at: http://neuralnetworksanddeeplearning.com/chap4.html (Accessed: 8 April 2019).

Raff, E., Barker, J., Sylvester, J., Brandon, R., Catanzaro, B. and Nicholas, C. (2017) 'Malware Detection by Eating a Whole EXE', *arXiv preprint*, arXiv:1710.09435.

Silver, D., Schrittwieser, J., Simonyan, K., Antonoglou, I., Huang, A., Guez, A., Hubert, T., Baker, L., Lai, M., Bolton, A., Chen, Y., Lillicrap, T., Hui, F., Sifre, L., van den Driessche, G., Graepel, T. and Hassabis, D. (2017) 'Mastering the game of Go without human knowledge', *Nature*, 550(7676), pp. 354-359.

Simonyan, K. and Zisserman, A. (2014) 'Very deep convolutional networks for large-scale image recognition', *arXiv preprint*, arXiv:1409.1556.

Szegedy, C., Zaremba, W., Sutskever, I., Bruna, J., Erhan, D., Goodfellow, I. and Fergus, R. (2013) 'Intriguing properties of neural networks', *arXiv preprint*, arXiv:1312.6199.

Taigman, Y., Yang, M., Ranzato, M. and Wolf, L. (2014) 'DeepFace: Closing the Gap to Human-Level Performance in Face Verification', *Conference on Computer Vision and Pattern Recognition (CVPR)*.

Available at: https://research.fb.com/publications/deepface-closing-the-gap-to-human-level-performance-in-face-verification/ (Accessed: 8 Apr. 2019).

Tramèr, F., Kurakin, A., Papernot, N., Goodfellow, I., Boneh, D. and McDaniel, P. (2017) 'Ensemble adversarial training: Attacks and defenses', *arXiv preprint*, arXiv:1705.07204.

Yin, C., Zhu, Y., Fei, J. and He, X. (2017) 'A Deep Learning Approach for Intrusion Detection Using Recurrent Neural Networks', *IEEE Access*, 5, pp. 21954-21961.

Appendices

Appendix A: Output Generated During Training of Deep Learning Algorithms

Figure 6: Output generated by training a ResNet-34 model on the Plant Seedlings Dataset for 16 epochs

epoch	train_loss	valid_loss	error_rate	time
0	2.412446	1.588891	0.490618	01:40
1	1.631501	0.887924	0.289090	01:37
2	1.068575	0.612517	0.208478	01:36
3	0.765255	0.487102	0.172342	01:36
4	0.565815	0.362957	0.125087	01:36
5	0.454169	0.319607	0.096595	01:36
6	0.395818	0.299391	0.101459	01:35
7	0.344555	0.287225	0.100069	01:36
8	0.321700	0.267520	0.092425	01:35
9	0.274559	0.244122	0.090341	01:35
10	0.230281	0.223646	0.071577	01:36
11	0.214568	0.214752	0.075052	01:36
12	0.208934	0.213815	0.073662	01:36
13	0.201646	0.206389	0.070188	01:36
14	0.189676	0.206193	0.068798	01:38
15	0.182272	0.206594	0.069493	01:37

Figure 7: Output generated by training a VGG16 model on the Plant Seedlings Dataset for 8 epochs

epoch	train_loss	valid_loss	error_rate	time
0	2.191624	1.285735	0.403753	01:54
1	1.303915	0.608995	0.200139	01:54
2	0.825484	0.420203	0.144545	01:53
3	0.560778	0.323622	0.109798	01:52
4	0.438575	0.284652	0.093815	01:52
5	0.371405	0.248670	0.084086	01:53
6	0.319173	0.241808	0.079917	01:53
7	0.280977	0.238900	0.081306	01:53

Figure 8: Output generated by training a DenseNet-121 model on the Plant Seedlings Dataset for 8 epochs

epoch	train_loss	valid_loss	error_rate	time
0	1.805645	0.939989	0.275191	01:47
1	0.966908	0.396469	0.141765	01:46
2	0.565523	0.257678	0.084086	01:48
3	0.382333	0.202248	0.076442	01:46
4	0.283218	0.177989	0.069493	01:48
5	0.213332	0.159022	0.057679	01:47
6	0.178006	0.147447	0.051425	01:48
7	0.144156	0.147266	0.055594	01:48

Figure 9: Output generated by training a ResNet-50 model on the Plant Seedlings Dataset for 6 epochs

epoch	train_loss	valid_loss	error_rate	time
0	1.518615	0.696576	0.232801	01:49
1	0.828608	0.353467	0.127172	01:48
2	0.522105	0.297074	0.106324	01:47
3	0.362662	0.243041	0.082001	01:47
4	0.277024	0.207511	0.071577	01:47
5	0.226484	0.196880	0.061848	01:47

Figure 10: Output generated by training a ResNet-34 model on both clean and adversarial versions of the Plant Seedlings Dataset for 8 epochs

epoch	train_loss	valid_loss	error_rate	time
0	1.870776	2.317155	0.782488	01:59
1	1.118410	2.348382	0.755386	01:59
2	0.791641	2.104018	0.678944	01:59
3	0.619517	1.969791	0.628909	01:59
4	0.505185	1.819433	0.581654	01:59
5	0.431195	1.703422	0.560111	01:59
6	0.360497	1.653044	0.553857	01:59
7	0.329481	1.645283	0.537179	01:57

Figure 11: Output generated by training a ResNet-34 model on both clean and adversarial versions of the Plant Seedlings Dataset for a further 40 epochs

epoch	train_loss	valid_loss	error_rate	time
0	0.322823	1.632367	0.528839	02:04
1	0.311734	1.576342	0.519805	02:02
2	0.309219	1.482525	0.503127	02:03
3	0.279063	1.339798	0.456567	02:04
4	0.228404	1.224715	0.413482	02:04
5	0.198590	1.105657	0.370396	02:04
6	0.160481	1.001995	0.342599	02:03
7	0.135286	0.938145	0.314107	02:03
8	0.106285	0.851058	0.285615	02:05
9	0.090223	0.820406	0.266852	02:03
10	0.077459	0.808099	0.256428	02:04
11	0.074354	0.764978	0.242530	02:04
12	0.055181	0.720763	0.234190	02:03
13	0.050687	0.708214	0.216817	02:04
14	0.041663	0.728768	0.224461	02:04
15	0.044202	0.748255	0.231411	02:03
16	0.037407	0.731320	0.230021	02:04
17	0.042055	0.755304	0.230716	02:04
18	0.037911	0.722031	0.225156	02:04
19	0.037038	0.724001	0.231411	02:03

epoch	train_loss	valid_loss	error_rate	time
0	0.039838	0.674920	0.216817	02:04
1	0.036653	0.700747	0.220292	02:03
2	0.028350	0.702757	0.220987	02:05
3	0.031981	0.713795	0.222377	02:06
4	0.031615	0.661515	0.213343	02:05
5	0.028658	0.703750	0.204309	02:04
6	0.026745	0.618314	0.187630	02:04
7	0.025445	0.605817	0.177901	02:05
8	0.017973	0.641139	0.184156	02:05
9	0.014517	0.578458	0.164698	02:05
10	0.016939	0.594032	0.168867	02:05
11	0.013050	0.561784	0.158443	02:05
12	0.009844	0.583164	0.164698	02:05
13	0.014308	0.575101	0.164698	02:05
14	0.011383	0.586625	0.169562	02:04
15	0.009958	0.578049	0.168172	02:05
16	0.009132	0.559247	0.164698	02:03
17	0.007730	0.542578	0.159833	02:05
18	0.008937	0.551549	0.159138	02:05
19	0.007835	0.563993	0.157748	02:03

Figure 12: Output generated by training a ResNet-34 model on both clean and adversarial versions of the Plant Seedlings Dataset for a further 20 epochs

epoch	train_loss	valid_loss	error_rate	time
0	0.008748	0.561670	0.161918	02:02
1	0.006592	0.551359	0.154969	02:02
2	0.009509	0.566927	0.158443	02:04
3	0.007766	0.560591	0.159138	02:04
4	0.010239	0.557230	0.147325	02:04
5	0.011380	0.553411	0.154969	02:04
6	0.010583	0.547203	0.154969	02:03
7	0.006563	0.565402	0.152884	02:02
8	0.008961	0.553419	0.149409	02:02
9	0.006559	0.531775	0.140375	02:03
10	0.006407	0.590931	0.153579	02:04
11	0.006148	0.542343	0.145935	02:03
12	0.005732	0.532248	0.143155	02:03
13	0.004300	0.526312	0.138985	02:03
14	0.002767	0.506562	0.136901	02:04
15	0.004413	0.490943	0.134121	02:03
16	0.004689	0.517485	0.137596	02:03
17	0.003177	0.525563	0.141765	02:02
18	0.002461	0.523140	0.140375	02:02
19	0.002961	0.517210	0.136206	02:02