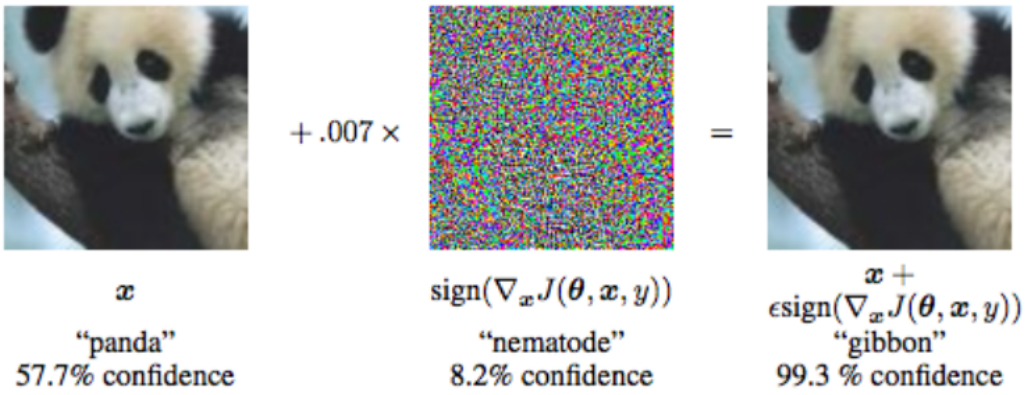
# Creating Adversarial Examples for Neural Networks with JAX

In this tutorial, we will see how to create Adversarial Examples that fool neural networks using JAX.

Agasti Kishor Dukare Follow
Apr 1 · 5 min read

Photo by Debbie Molle on Unsplash

Firstly, let's see some definitions. What are the Adversarial Examples? Simply put, Adversarial Examples are inputs to a neural network that are optimized to fool the algorithm i.e. result in misclassification of the target variable. We can misclassify the target variable by adding 'proper' noise to it. The concept has been demonstrated in the following image.




From Explaining and Harnessing Adversarial Examples by Goodfellow et al.

Today, the focus of this tutorial is to demonstrate how you can create the Adversarial Examples. We are going to generate using the ***fast gradient sign method.***

In this method, if x is an input image, we modify x into

$$x' = x + \epsilon \cdot sign(\bigtriangledown_x J(\theta, x, y))$$

where the adversarial input is obtained by taking the ***sign*** of the gradient of cross-entropy loss w.r.t input image ***x*** and adding it to the original image. ε is the hyperparameter here.

For this tutorial, we are going to use the popular MNIST dataset. If you don't know what MNIST dataset is, I suggest going to the following link.

**MNIST database**

The MNIST database (Modified National Institute of Standards and Technology database) is a large database of...

en.wikipedia.org

To train our model and generate Adversarial Examples, we are going to use the JAX module. JAX is Automatic Differentiation (AD) toolbox which comes handy when it comes to training massive datasets such as MNIST. Someone has aptly described JAX as Numpy on steroids! As this is not 'Introduction to JAX' tutorial I won't be diving deeper into it.

You can refer to the following documentation to know further about JAX.

**google/JAX**

Announcement: JAX has dropped Python 2 support, and requires Python 3.6 or newer. See docs/CHANGELOG.rst. JAX is...

github.com

For beginners, I suggest referring the following tutorial on JAX and its usage.

**You don't know JAX**

You don't know JAX This brief tutorial covers the basics of JAX. JAX is a Python library which augments numpy and...

colinraffel.com

Now let's dive into coding. The code that I'm providing has been built upon the following GitHub repository. I have made the necessary changes and have added some new functions to make it suitable for the application at hand. You can visit the following link for reference.

**tensorflow/cleverhans**

An adversarial example library for constructing attacks, building defenses, and benchmarking both ...

github.com

Firstly, we will import all the important libraries.

```
import array
import gzip
import itertools
import numpy
import numpy.random as npr
import os
import struct
import time
from os import path
import urllib.request
import jax.numpy as np
from jax.api import jit, grad
from jax.config import config
from jax.scipy.special import logsumexp
from jax import random
import matplotlib.pyplot as plt
```

imports.py hosted with ♡ by GitHub view raw

Next, we will download and load MNIST data.

```
_DATA = "/tmp/"

def _download(url, filename):
  """Download a url to a file in the JAX data temp directory."""
  if not path.exists(_DATA):
    os.makedirs(_DATA)
  out_file = path.join(_DATA, filename)
  if not path.isfile(out_file):
    urllib.request.urlretrieve(url, out_file)
    print("downloaded {} to {}".format(url, _DATA))

```

```
def _partial_flatten(x):
  """Flatten all but the first dimension of an ndarray."""
  return numpy.reshape(x, (x.shape[0], -1))


def _one_hot(x, k, dtype=numpy.float32):
  """Create a one-hot encoding of x of size k."""
  return numpy.array(x[:, None] == numpy.arange(k), dtype)


def mnist_raw():
  """Download and parse the raw MNIST dataset."""
  # CVDF mirror of http://yann.lecun.com/exdb/mnist/
  base_url = "https://storage.googleapis.com/cvdf-datasets/mnist/"

  def parse_labels(filename):
    with gzip.open(filename, "rb") as fh:
      _ = struct.unpack(">II", fh.read(8))
      return numpy.array(array.array("B", fh.read()), dtype=numpy.uint8)

  def parse_images(filename):
    with gzip.open(filename, "rb") as fh:
      _, num_data, rows, cols = struct.unpack(">IIII", fh.read(16))
      return numpy.array(array.array("B", fh.read()),
                         dtype=numpy.uint8).reshape(num_data, rows, cols)

  for filename in ["train-images-idx3-ubyte.gz", "train-labels-idx1-ubyte.gz",
                   "t10k-images-idx3-ubyte.gz", "t10k-labels-idx1-ubyte.gz"]:
    _download(base_url + filename, filename)

  train_images = parse_images(path.join(_DATA, "train-images-idx3-ubyte.gz"))
  train_labels = parse_labels(path.join(_DATA, "train-labels-idx1-ubyte.gz"))
  test_images = parse_images(path.join(_DATA, "t10k-images-idx3-ubyte.gz"))
  test_labels = parse_labels(path.join(_DATA, "t10k-labels-idx1-ubyte.gz"))

  return train_images, train_labels, test_images, test_labels


def mnist(create_outliers=False):
  """Download, parse and process MNIST data to unit scale and one-hot labels."""
  train_images, train_labels, test_images, test_labels = mnist_raw()

  train_images = _partial_flatten(train_images) / numpy.float32(255.)
  test_images = _partial_flatten(test_images) / numpy.float32(255.)
```

```
  train_labels = _one_hot(train_labels, 10)
  test_labels = _one_hot(test_labels, 10)

  if create_outliers:
    mum_outliers = 30000
    perm = numpy.random.RandomState(0).permutation(mum_outliers)
    train_images[:mum_outliers] = train_images[:mum_outliers][perm]

  return train_images, train_labels, test_images, test_labels

def shape_as_image(images, labels, dummy_dim=False):
  target_shape = (-1, 1, 28, 28, 1) if dummy_dim else (-1, 28, 28, 1)
  return np.reshape(images, target_shape), labels

train_images, train_labels, test_images, test_labels = mnist(create_outliers=False)
num_train = train_images.shape[0]
```

neural network by iterating over all of its layers, taking the activations of the input/previous layer and applying the tanh activation.

Remember that for the output we use, $\mathbf{z=w \cdot x+b}$

```
def predict(params, inputs):
  activations = inputs
  for w, b in params[:-1]:
    outputs = np.dot(activations, w) + b
    activations = np.tanh(outputs)

  final_w, final_b = params[-1]
  logits = np.dot(activations, final_w) + final_b
  return logits - logsumexp(logits, axis=1, keepdims=True)
```

pred.py hosted with ♡ by GitHub view raw

In this tutorial, we are going to use a cross-entropy loss. The following function will return us the loss of our model.

```
# loss function for calculating predictions and accuracy before pertubation
def loss(params, batch, test=0):
  inputs, targets = batch
  logits = predict(params, inputs)
  preds  = stax.logsoftmax(logits)
```

```
    if(test==1):
      print('Prediction Vector before softmax')
      print(logits)
      print("_____________________________________________________________________________")
      print('Prediction Vector after softmax')
      print(preds)
      print("_____________________________________________________________________________")
    return -(1/(preds.shape[0]))*np.sum(targets*preds)

# loss function for calculating gradients of loss w.r.t. input image
def lo(batch,params):
    inputs, targets = batch
    logits = predict(params, inputs)
    preds  = stax.logsoftmax(logits)
    return -(1/(preds.shape[0]))*np.sum(targets*preds)
```

The following cell defines the accuracy of our model and how to initialize its parameters.

```
def accuracy(params, batch):
  inputs, targets = batch
  target_class = np.argmax(targets, axis=1)
  predicted_class = np.argmax(predict(params, inputs), axis=1)
  return np.mean(predicted_class == target_class), target_class, predicted_class
```

accuracy.py hosted with ♡ by GitHub view raw

Now we have to generate batches of our training data. For this purpose, we will create a Python generator for our dataset. It outputs one batch of n training examples at a time.

```
batch_size = 128
num_complete_batches, leftover = divmod(num_train, batch_size)
num_batches = num_complete_batches + bool(leftover)

def data_stream():
  rng = npr.RandomState(0)
  while True:
    perm = rng.permutation(num_train)
    for i in range(num_batches):
      batch_idx = perm[i * batch_size:(i + 1) * batch_size]
      yield train_images[batch_idx], train_labels[batch_idx]
```

```
batches = data_stream()
```

generator.py hosted with ♡ by GitHub view raw

Next, our job is to create a fully-connected neural network architecture using ***'stax'***. Stax is a neural net specification library. Here we are detailing the specifications for the layers within our Convolution Neural Network.

```
init_random_params, predict = stax.serial(
    stax.Conv(64, (7,7), padding='SAME'),
    stax.Relu,
    stax.Conv(32, (4, 4), padding='SAME'),
    stax.Relu,
    stax.MaxPool((3, 3)),
    stax.Flatten,
    stax.Dense(128),
    stax.Relu,
    stax.Dense(10),
)
```

stax.py hosted with ♡ by GitHub view raw

Now we have to define the mini-batch SGD optimizer. The optimizer gives us 3 things.

1] a method opt_init that takes in a set of initial parameter values returned by init_fun and returns the initial optimizer state opt_state,

2]a method opt_update which takes in gradients and parameters and updates the optimizer states by applying one step of optimization, and

3] a method get_params that take in an optimizer state and return current parameter values.

```
learning_rate = 0.14
opt_init, opt_update, get_params = optimizers.sgd(learning_rate)

@jit
def update(_, i, opt_state, batch):
  params = get_params(opt_state)
  return opt_update(i, grad(loss)(params, batch), opt_state)
```

update.py hosted with ♡ by GitHub view raw

Next, we will train our model on the training examples. At the end of the training, we will obtain the 'params' which we are going to use to calculate the gradient of our loss function w.r.t. the test image.

```
num_epochs = 1
key = random.PRNGKey(123)
_, init_params = init_random_params(key, (-1, 28, 28, 1))
opt_state = opt_init(init_params)
itercount = itertools.count()
for _ in range(num_batches):
  opt_state= update(key, next(itercount), opt_state, shape_as_image(*next(batches)))
params = get_params(opt_state)
```

train.py hosted with ♡ by GitHub view raw

Finally, we define the function which will return us the gradient of the loss function w.r.t the test input. Also, this function will calculate test loss as well as predict the class of our target variable.

```
# This function calculates, loss, predictions and gradients
def covnet(t,params):
  test_acc,target_class, predicted_class = accuracy(params, shape_as_image(test_images, test_labe
  test_loss = loss(params, shape_as_image(test_images, test_labels),test=t)
  grads = grad(lo)(shape_as_image(test_images, test_labels),params)
  if(t==1):
      print('Test set loss, accuracy (%): ({:.2f}, {:.2f})'.format(test_loss, 100 * test_acc))
      print('predicted_class,target_class', predicted_class,target_class)
  return grads, test_acc
```

gradient.py hosted with ♡ by GitHub view raw

Now it's time to test our model.

Firstly let's take one test input. Here we are choosing an image which belongs to the class '7'.

Let's visualize the original image.

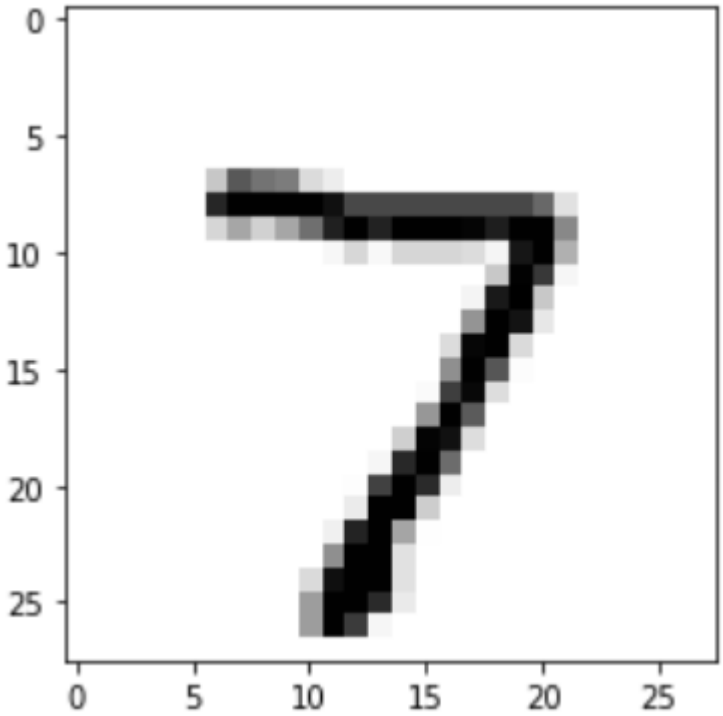
```
def display(image):
  img = image[0].reshape((28,28))
```

```
    plt.imshow(img, cmap="Greys")
    plt.show()
    return

display(a)
```

display.py hosted with ♡ by GitHub view raw

The code above gives us the following output.




Let's see if our trained model predicts the accurate class for this image.

```
# load desired image and its label in test set
def load_img(image,img_label):
  img = np.array(image)
  img = img.reshape(1,784)
  label = np.array(img_label)
  label = label.reshape(1,10)
  return img, label

img, label = load_img(test_images[0],test_labels[0])
test_images = img
test_labels = label

#Predictions Before Pertubation
```

```
14   grads,acc = covnet(1,params)
```

example.py hosted with ♡ by GitHub view raw

After running the code above, we get the following output.

```
Prediction Vector before softmax
[[ -1.3639672  -1.3191454   5.12589     3.7558646  -7.102557   -0.8500258
  -10.143162   14.223044    0.6110929   3.5588162]]
________________________________________________________________________________
Prediction Vector after softmax
[[-1.5587177e+01 -1.5542356e+01 -9.0973206e+00 -1.0467346e+01
  -2.1325768e+01 -1.5073236e+01 -2.4366373e+01 -1.6556800e-04
  -1.3612118e+01 -1.0664394e+01]]
________________________________________________________________________________
Test set loss, accuracy (%): (0.00, 100.00)
predicted_class,target_class [7] [7]
================================================================================
```

We see that our model has correctly predicted the class of our input image.

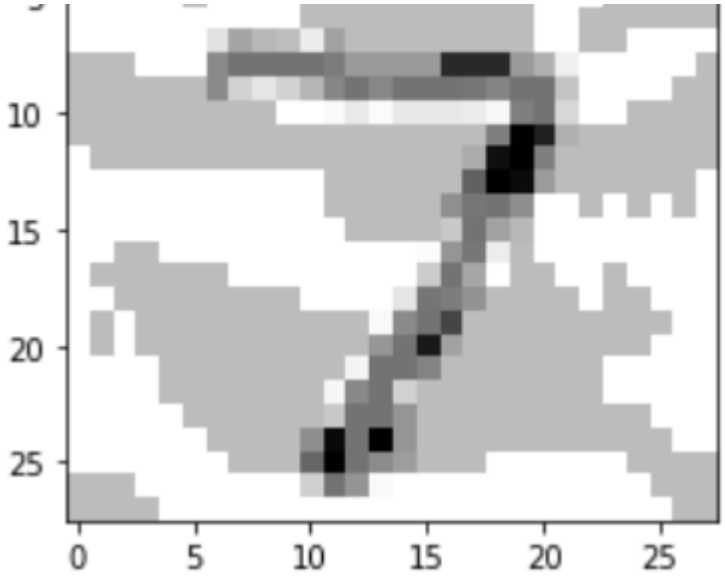
Now let's jump to the fun stuff. Let's perturb the same image with ***fast gradient sign method.***

For this purpose, we have defined the function. Here we are using 0.3 as the value of hyper-parameter epsilon.

```
1  # This function calculates perturbted image
2  def perturb(grad,img,ep):
3    grads = grad[0]
4    a = numpy.reshape(grads, (1, 784))
5    s = np.sign(a)
6    perturbed_image = img + np.dot(ep,s)
7    return perturbed_image
8  per_img = perturb(grads,img,0.3)
9  display(per_img)
```

perturb.py hosted with ♡ by GitHub view raw

After running code above we get the following output.




We see that the perturbed image has a lot of noise. This noise amount can be controlled by the value of hyperparameter epsilon.

Finally, let's see if the noise has any effect on the model classification, i.e. if our model misclassifies the perturbed image.

```
# Load perturbted image and print predictions after pertubation
test_images = per_img
test_labels = label
grads,acc = covnet(1,params)
```

verify.py hosted with ♡ by GitHub view raw

The code above gives us the following results.

```
Prediction Vector before softmax
[[ -3.073985   -3.1178942   3.7730384   6.364301   -2.2132506   2.5553052
  -10.346308   -0.6956062   4.248633    2.6351058]]
_____________________________________________________________________________
Prediction Vector after softmax
[[ -9.655704    -9.699613    -2.808681    -0.21741812  -8.79497
   -4.026414   -16.928028    -7.2773256   -2.3330865   -3.9466136 ]]
_____________________________________________________________________________
Test set loss, accuracy (%): (7.28, 0.00)
predicted_class,target_class [3] [7]
=============================================================================
```

Voila! our model has misclassified the input image to class **'3'**.

Thus we saw how to generate Adversarial Examples. You can repeat the same procedure for the entire test set.

If you are curious about JAX, I suggest visiting the following page and see what more you can do with JAX.

**Getting started with JAX (MLPs, CNNs & RNNs)**

JAX, Jax, JaX. Twitter seems to know nothing else nowadays (next to COVID-19). If you are like me and want to know what...

roberttlange.github.io

The image sources are:

https://www.researchgate.net/figure/An-illustration-of-machine-learning-adversarial-examples-Studies-have-shown-that-by_fig1_325370539

. . .

If you like this tutorial, please give it a clap! Also, any suggestions/corrections are welcome.

Happy Coding!!

Deep Learning Machine Learning Artificial Intelligence Data Science Python