Vulnerability of generative adversarial networks to adversarial images

Cognitive computational neuroscience - Final assignment

Stijn Voss, s4150511

January 2017

Abstract

In this assignment I will compare the vulnerability to adversarial examples of discriminators trained by generative adversarial networks to the vulnerability of supervised trained models. I find some interesting evidence that suggests that indeed generative trained models might be less vulnerable for adversarial examples.

1 Introduction

The rise of deep learning techniques has lead to enormousness improvements in the machine learning field. However there still seems to be a lot of differences between the way our brain learns and the way these algorithms learn. A deep neural network is capable of learning low level features and hierarchical patterns in the data by giving it a lot of labeled examples(Yosinski, Clune, Nguyen, Fuchs, & Lipson, 2015). Our brain however is not provided with so many labeled examples: it seems to figure out the patterns in our world from the data itself.

1.1 Generative adversarial networks

Recently a lot of techniques have been proposed to learn data distributions in an (semi-) or un-supervised manner. Most notably Variational autoencoders(Kingma & Welling, 2013) and Generative adversarial nets(GAN)(I. Goodfellow et al., 2014). A GAN consists of a Generator net, that produces a sample from the data distribution given a random noise input and a Discriminator that tries to distinguish samples from the dataset and from the generator. Both networks are then trained while competing against each other. GAN's especially perform well on image data and have been applied to different image datasets(Radford, Metz, & Chintala, 2015; Salimans et al., 2016).

The intuition is that the weights learned by the generator are related to the patterns for a traditional classifier network. Where traditional networks work by going from low-level features to high-level features, the generator would go the from high level to low level. Sadly the generators weights of a GAN cannot be converted so easily to be used in a classifier network. Although some solutions have been proposed(Donahue, Krähenbühl, & Darrell, 2016).

Other efforts to minimize the need for labeled data make use of the discriminator. Instead of letting the discriminator choose between real or fake, these semi-supervised methods (Odena, 2016) let the discriminator choose between one of the original classification labels or fake. Since the generator will try to generate similar images the discriminator can be trained with way less labeled data.

1.2 Adversarial images

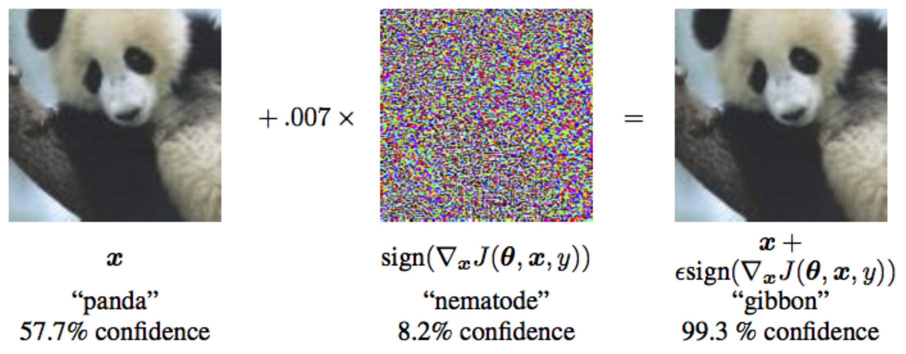
Szegedy et al. (2013) showed that deep convolutional networks have some very interesting properties. Networks can be tricked in wrongly classifying a synthetically modified image with quit some confidence. For humans such an image looks a lot like the original image, however the network classifies the image as something entirely different.

The problem of adversarial images is present in all types of machine learning algorithms and especially when these models tend to behave linearly, networks that use the sigmoid activation function are less vulnerable (I. J. Goodfellow, Shlens, & Szegedy, 2014) for example. Furthermore those examples are universal across different models and not model specific. The same adversarial example is likely to fail in another network as long as it trained with the same or a subset of the same dataset.

Based on it's theoretical findings I. J. Goodfellow et al. (2014) also proposed a simple method to generate adversarial images. Given an input $\mathbf{x}$ with corresponding target variable $\mathbf{y}$, a model θ and loss function $J(\theta, x, y)$ we can calculate the gradient for the loss with respect to the input $\Delta_x J(\theta, x, y)$. In normal gradient descent we would make sure to minimize the loss, but here we will enlarge the loss on purpose.

$$x_{adv} = x + \epsilon sign(J(\theta, x, y))$$

Where x_{adv} will be our adversarial example. The loss gives us a sense of where we have to go to trick the network. Please note that we take the sign() of the gradient. We can interpret the result as the pixels we have to change in which direction.



$+ .007 \times$
$=$
$\boldsymbol{x}$
"panda"
57.7% confidence
$\text{sign}(\nabla_{\boldsymbol{x}} J(\boldsymbol{\theta}, \boldsymbol{x}, y))$
"nematode"
8.2% confidence
$\boldsymbol{x} + \epsilon\text{sign}(\nabla_{\boldsymbol{x}} J(\boldsymbol{\theta}, \boldsymbol{x}, y))$
"gibbon"
99.3 % confidence

Figure 1: Example of adversarial example produced by method of 1.2 using the ImageNet network (I. J. Goodfellow et al., 2014)

1.3 Research

Some defense systems against adversarial neural networks actually use a generative approach. For example (Fei Xia, 2016) show that one can actually train a network to generate adversarial examples. The generator of a normal GAN might actually be tempted to try to create adversarial images it self and try to trick the discriminator. In my research I want to compare the vulnerability of supervised trained networks to adversarial images with that of a GAN trained discriminator.

To do so I will train both types of networks on the MNIST dataset. This is a simple dataset of 28x28 black and white images representing handwritten digits in the range from 0-9. The training set contains 60.000 samples where the test set contains 10.000 samples. I will both train a discriminator using the GAN approach and a normal supervised network. Next I will create adversarial examples using the supervised learned model and compare the performance of the two networks on these examples.

2 Experiments

2.1 Generating the adversarial images

I started by generating some adversarial images, for this purpose I used the default MNIST implementation from the chainer tutorial[1]. I adapted the network a bit such that it would correspond to the GAN discriminator. This is a simple fully connected neural network consisting of 3 hidden layers with 1000, 500 and 250 hidden units respectively. The networks has 756(28*28) units as input, the output layer consists of 10 units.

After I trained the network I could use the learned parameters to generate the images. For

[1]https://github.com/pfnet/chainer/blob/master/examples/mnist/train_mnist.py

this I was inspired by the work of Robert Lacok [2]. Then for every image in the test set we:

- Forward it trough the network giving me the original probabilities and prediction
- Based on the corresponding target class calculate the gradient on the input
- For $\epsilon \in \{.1, .25, 1.0\}$ calculate the new image using the method described in 1.2. Each of these images are passed forward trough the network, giving me a loss, predicted label and confidence levels for each image and each ϵ

Note that we make sure that all adversarial images stay within the 0 and 1 range by setting all values lower then 0 to 0 and all values higher then 1 to 1. This way all images remain valid input. In figure 2 and 3 we show some adversarial examples. Please note that since MNIST data is binary and relatively low dimensional the changes noise added will be higher required to create adversarial examples than for example is the case for imagenet examples.

In general it seems that by humans images with $\epsilon \in \{0.1, 0.25\}$ are being observed as the same images as the original but with a bit of random noise, but it can be enough to trick the neural network. However $\epsilon = 1.0$ are completely unreadable. We summarize results over all images in table 1. For each value of ϵ we show the accuracy(the fraction of the images that where classified correctly with the original image label) and the average confidence(the average max. probability).

type	**accuracy**	**avg. confidence**
$\epsilon = 0.00$	0.981	0.994
$\epsilon = 0.10$	0.537	0.956
$\epsilon = 0.25$	0.306	0.965
$\epsilon = 1.00$	0.142	0.988

Table 1: Results for different values of ϵ for a supervised trained neural network

We see that even with quit a small $\epsilon = 0.1$ we can trick the network in miss classifying an additional 40% of the images with quit a high confidence. For $\epsilon = 0.25$ it's even worse.

[2]https://github.com/robertlacok/mnist-adversarial-examples

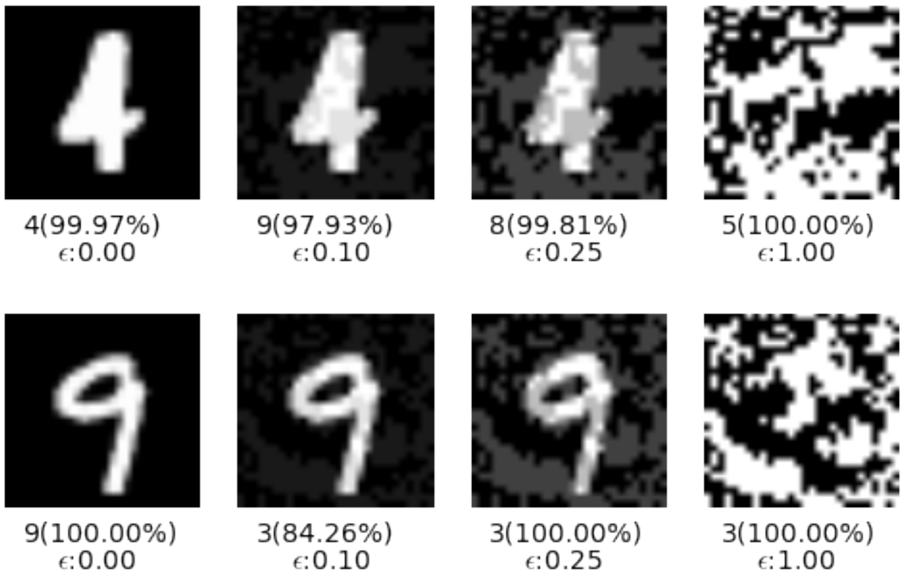


Figure 2: Adversarial example images that where classified incorrectly starting from $\epsilon \geq 0.1$. Picked randomly from all available examples.We show the confidence in brackets, note that this number is rounded to two decimals.

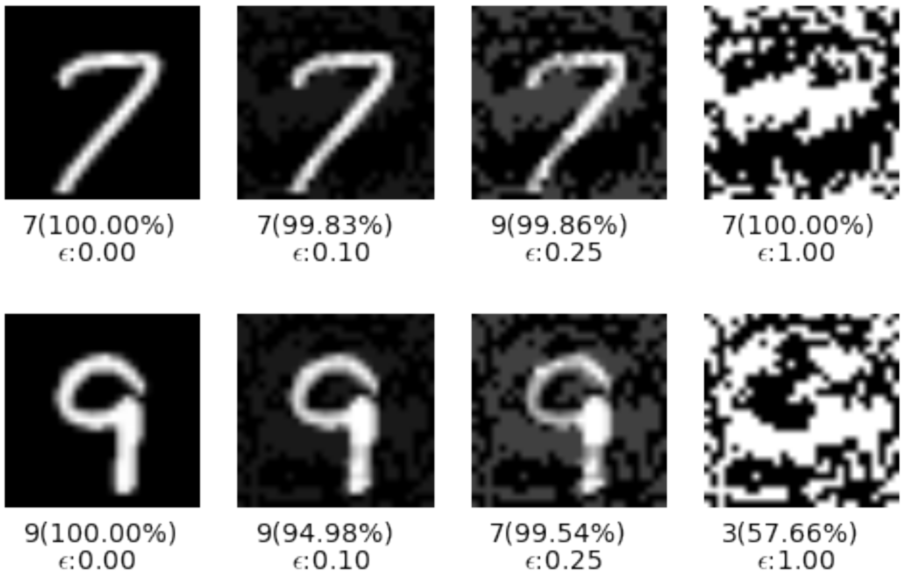


Figure 3: Images that where classified incorrectly starting from $\epsilon \geq 0.25$ Picked randomly from all available examples.We show the confidence in brackets, note that this number is rounded to two decimals.

2.2 Descriminator of the gan

We will now look into how our GAN trained discriminator performs. Sadly we cannot create adversarial images from the discriminator itself, for some reason the discriminator usually predicts with a very high confidence which results in a loss of 0.0 and no gradient. Luckily adversarial images tend to be universal for different models, so we will re-use the examples found in 2.1 and see how our generator performs on these examples.

First we have to learn our discriminator using the GAN method. To do this we use the chainer implementation[3] of the work of Salimans et al. (2016). Please note that one crucial difference between this network and the network learned in 2.1 gets floats between 0-1 as input where as the discriminator is trained 0-255 integer input.

Next we forward all the images found in 2.1 trough our discriminator while calculating predicted labels and confidence. Since the GAN training is very much focused on the generator the discriminator performs a bit poorly on the normal images. Therefore we also stopped training after 250 epoch instead of the original 1000, in this case the discriminator performs a bit better. The results are summarized in table 2 and 3.

type	accuracy	avg. confidence
$\epsilon = 0.00$	0.618	1.000
$\epsilon = 0.10$	0.554	1.000
$\epsilon = 0.25$	0.446	1.000
$\epsilon = 1.00$	0.154	1.000

Table 2: Results for different values of ϵ for a gan trained discriminator model with 1000 training epochs

type	accuracy	avg. confidence
$\epsilon = 0.00$	0.768	1.000
$\epsilon = 0.10$	0.698	1.000
$\epsilon = 0.25$	0.525	1.000
$\epsilon = 1.00$	0.115	1.000

Table 3: Results for different values of ϵ for a gan trained discriminator model with 250 training epochs

The model performs way worse than the supervised model in general. It seems that in both cases the accuracy drops less hard when presented with the adversarial examples compared to the supervised example. In the case of the 1000 trained epoch model as shown in table 2 we see that $\epsilon = 0.25$ performs quit a bit better than it's supervise counter part. In the case of 250 epochs as shown in table 3 we see that it performs better for $\epsilon \in \{.1, .25\}$.

3 Discussion and conclusion

Sadly the discriminator learned by our GAN performs worse than our supervised model by a large margin. The code I used is mainly targeted at learning an effective generator, not at building a high quality classifier. This results in the poor performing discriminator which made my results a bit weaker. Due to the time constraints I had no time to look into the complicated GAN code and try to achieve better results.

However as far as I am concerned there is no reason why a network that performs worse on non-adversarial examples would automatically perform better on adversarial examples. So I think my evidence is still pretty strong.

A question that arises in the context of the course is if this also tells use something about how the brain learns, since the brain seems less vulnerable to adversarial examples. Of course it's very hard to give an answer to this question. The learned generator might just be tempted to generate adversarial examples and thus the discriminator learns to deal with them, but we

[3] https://github.com/musyoku/improved-gan

can not say that this means that the brain is also generating images to learn the pattern. But I theorize that the brain is more involved in learning patterns and a model of the world for later predictions than just making classifications: it learns unsupervised. It could be that this difference is also responsible for the difference in vulnerability to adversarial examples.

References

Donahue, J., Krähenbühl, P., & Darrell, T. (2016). Adversarial feature learning. *arXiv preprint arXiv:1605.09782*.

Fei Xia, R. L. (2016, dec). *Adversarial examples generation and defense based on generative adversarial network.* http://cs229.stanford.edu/proj2016/report/LiuXia-AdversarialExamplesGenerationAndDefenseBasedOnGenerativeAdversarialNetwork-report.pdf.

Goodfellow, I., Pouget-Abadie, J., Mirza, M., Xu, B., Warde-Farley, D., Ozair, S., … Bengio, Y. (2014). Generative adversarial nets. In *Advances in neural information processing systems* (pp. 2672–2680).

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

Kingma, D. P., & Welling, M. (2013). Auto-encoding variational bayes. *arXiv preprint arXiv:1312.6114*.

Odena, A. (2016). Semi-supervised learning with generative adversarial networks. *arXiv preprint arXiv:1606.01583*.

Radford, A., Metz, L., & Chintala, S. (2015). Unsupervised representation learning with deep convolutional generative adversarial networks. *arXiv preprint arXiv:1511.06434*.

Salimans, T., Goodfellow, I., Zaremba, W., Cheung, V., Radford, A., & Chen, X. (2016). Improved techniques for training gans. In *Advances in neural information processing systems* (pp. 2226–2234).

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

Yosinski, J., Clune, J., Nguyen, A., Fuchs, T., & Lipson, H. (2015). Understanding neural networks through deep visualization. *arXiv preprint arXiv:1506.06579*.

4 Code

My code consists of three notebooks:

- **Training GAN** which trains the generator and discriminator. Since the model training is quit intensive i did the actual training on a PC with a GPU via a shell script.
- **Training supervised** which trains the supervised network
- **Adversarial images** which based on the trained models above will generate adversarial images and show obtain the results

The notebooks are attached in the same order below. All notebook can also be found in my repository for the CCN course: `https://github.com/svoss/ru-ccn/tree/master/end-assignment/notebooks`

Training GAN

January 10, 2017

1 Training GAN

The openAI foundation proposed an improved GAN and was able to apply it on the MNIST dataset. You can found the paper here: https://arxiv.org/abs/1606.03498. Someone else re-implemented the code in Chainer here: https://github.com/musyoku/improved-gan. However the code is quit hard to understand so i will first try to reproduce their results and understand what they did. The code is divided in general code that is used for all different GAN applications and models specific code. For example code that is used for the MNIST model in particular or generating anime faces.

```
In [1]: # Some dependencies
        import math
        import json
        import numpy as np
        import chainer, os, collections, six, math, random, time, copy,sys
        from chainer import cuda, Variable, optimizers, serializers, function, optimizer, initia
        from chainer.utils import type_check
        from chainer import functions as F
        from chainer import links as L
        # add the imported repository to the path, so we can always just import
        sys.path.append(os.path.join(os.path.split(os.getcwd())[0],'improved-gan'))
```

1.1 Params

They formalize the params of the discrimator, generator and classifier in classes. The formalized classes are then used as input by the general GAN code to fit the different applications.

```
In [2]: # Base class
        # Found in params.py
        class Params():
            def __init__(self, dict=None):
                if dict:
                    self.from_dict(dict)

            def from_dict(self, dict):
                for attr, value in dict.iteritems():
                    if hasattr(self, attr):
```

```
                setattr(self, attr, value)

    def to_dict(self):
        dict = {}
        for attr, value in self.__dict__.iteritems():
            if hasattr(value, "to_dict"):
                dict[attr] = value.to_dict()
            else:
                dict[attr] = value
        return dict

    def dump(self):
        for attr, value in self.__dict__.iteritems():
            print "        {}: {}".format(attr, value)

# General GAN code (found in gan.py) :
# These params can be defined for a Discriminator class
class DiscriminatorParams(Params):
    def __init__(self):
        self.ndim_input = 28 * 28
        self.ndim_output = 10
        self.weight_init_std = 1
        self.weight_initializer = "Normal"  # Normal, GlorotNormal or HeNormal
        self.nonlinearity = "elu"
        self.optimizer = "Adam"
        self.learning_rate = 0.001
        self.momentum = 0.5
        self.gradient_clipping = 10
        self.weight_decay = 0
        self.use_feature_matching = False
        self.use_minibatch_discrimination = False

# These params can be defined for a Generator class
class GeneratorParams(Params):
    def __init__(self):
        self.ndim_input = 10
        self.ndim_output = 28 * 28
        self.distribution_output = "universal"  # universal, sigmoid or tanh
        self.weight_init_std = 1
        self.weight_initializer = "Normal"  # Normal, GlorotNormal or HeNormal
        self.nonlinearity = "relu"
        self.optimizer = "Adam"
        self.learning_rate = 0.001
        self.momentum = 0.5
        self.gradient_clipping = 10
        self.weight_decay = 0

# These parameters can
```

```
class ClassifierParams(Params):
    def __init__(self):
        self.ndim_input = 28 * 28
        self.ndim_output = 10
        self.weight_init_std = 1
        self.weight_initializer = "Normal"  # Normal, GlorotNormal or HeNormal
        self.nonlinearity = "elu"
        self.optimizer = "Adam"
        self.learning_rate = 0.001
        self.momentum = 0.5
        self.gradient_clipping = 10
        self.weight_decay = 0
        self.use_feature_matching = False
        self.use_minibatch_discrimination = False
```

1.2 Sequentials

The sequentials folder implements a lot of general neural network functionality to support the GAN model. For example a deconvolutional layer and weight normalization(https://arxiv.org/abs/1602.07868). I will not discuss all the code in detail since it's quit

One important class is the Sequential class, which implements a sequence of neural network layer. It is loaded into a chain before optimizing.

```
In [3]: import sequential
```

1.3 General GAN

The code below shows the general code that implements a GAN given the params defined above and a model for the discriminator and generator.

```
In [4]: class Sequential(sequential.Sequential):
            """
            Sequential formalizes a sequence of neural network layers
            """
            def __call__(self, x, test=False):
                activations = []
                for i, link in enumerate(self.links):
                    if isinstance(link, sequential.functions.dropout):
                        x = link(x, train=not test)
                    elif isinstance(link, chainer.links.BatchNormalization):
                        x = link(x, test=test)
                    else:
                        x = link(x)
                        if isinstance(link, sequential.functions.ActivationFunction):
                            activations.append(x)
                return x, activations

        # Following two help saving objects
```

```
class Object(object):
    pass


def to_object(dict):
    obj = Object()
    for key, value in dict.iteritems():
        setattr(obj, key, value)
    return obj

class GAN():
    def __init__(self, params_discriminator, params_generator):
        """
        As an input a GAN gets two arguments: a dictionary for the discriminator and a d
        Both have two items with the key config and model.
        The config key contains a param object implementing one of the param classes abo
        The model key contains a neural network, converted to a dictioniary via the Sequ

        """
        self.params_discriminator = copy.deepcopy(params_discriminator)
        self.config_discriminator = to_object(params_discriminator["config"])

        self.params_generator = copy.deepcopy(params_generator)
        self.config_generator = to_object(params_generator["config"])

        self.build_discriminator()
        self.build_generator()
        self._gpu = False

    def build_discriminator(self):
        # discriminator model is extracted and loaded into a chain next we can build a o
        self.discriminator = sequential.chain.Chain()
        self.discriminator.add_sequence(sequential.from_dict(self.params_discriminator["
        config = self.config_discriminator
        self.discriminator.setup_optimizers(config.optimizer, config.learning_rate, conf

    def build_generator(self):
        #generator model is extracted and loaded into a chain next we can build a optimi
        self.generator = sequential.chain.Chain()
        self.generator.add_sequence(sequential.from_dict(self.params_generator["model"])
        config = self.config_discriminator
        self.generator.setup_optimizers(config.optimizer, config.learning_rate, config.m

    def update_learning_rate(self, lr):
        #Change learning rate of both discriminator and generator seperately
        self.discriminator.update_learning_rate(lr)
        self.generator.update_learning_rate(lr)
```

```
def to_gpu(self):
    #Make sure both networks are trained on GPU
    self.discriminator.to_gpu()
    self.generator.to_gpu()
    self._gpu = True

@property
def gpu_enabled(self):
    # If gpu is set to true and cuda is available
    if cuda.available is False:
        return False
    return self._gpu

@property
def xp(self):
    # Get's cupy if gpu is enabled otherwise numpy
    if self.gpu_enabled:
        return cuda.cupy
    return np

def to_variable(self, x):
    # Helper function converts variable deals with gpu
    if isinstance(x, Variable) == False:
        x = Variable(x)
        if self.gpu_enabled:
            x.to_gpu()
    return x

def to_numpy(self, x):
    # helper functions converts to numpy deals with gpu
    if isinstance(x, Variable) == True:
        x = x.data
    if isinstance(x, cuda.ndarray) == True:
        x = cuda.to_cpu(x)
    return x

def get_batchsize(self, x):
    # Gets batch size
    return x.shape[0]

def zero_grads(self):
    # Reset all grads
    self.optimizer_discriminator.zero_grads()
    self.optimizer_generative_model.zero_grads()

def sample_z(self, batchsize=1):
    """ Generates a random z sample from an uniform distribution
    the gerenator will generate an image based on that input will use a complete bat
```

```
        """

        config = self.config_generator
        ndim_z = config.ndim_input
        # uniform
        z_batch = np.random.uniform(-1, 1, (batchsize, ndim_z)).astype(np.float32)
        # gaussian
        # z_batch = np.random.normal(0, 1, (batchsize, ndim_z)).astype(np.float32)
        return z_batch

    def generate_x(self, batchsize=1, test=False, as_numpy=False):
        """
        This function lets the generator generate a variable
        It combines the input from sample_z with generate_x_from_z to generator z
        """
        return self.generate_x_from_z(self.sample_z(batchsize), test=test, as_n

    def generate_x_from_z(self, z_batch, test=False, as_numpy=False):
        """
        This functions generates x a sample by the generator given a random input z.
        it will automatically get the batch size from z_batch
        """
        z_batch = self.to_variable(z_batch)
        x_batch, _ = self.generator(z_batch, test=test, return_activations=True)
        if as_numpy:
            return self.to_numpy(x_batch)
        return x_batch

    def discriminate(self, x_batch, test=False, apply_softmax=True):
        """
        Given an example produced by the generator (likely by using generate_x), will ca
        the discriminator and return an probability to see if this is a fake our another
        """
        x_batch = self.to_variable(x_batch)
        prob, activations = self.discriminator(x_batch, test=test, return_activations=Tr
        if apply_softmax:
            prob = F.softmax(prob)
        return prob, activations

    def backprop_discriminator(self, loss):
        # Backpropagate and learn through discriminator
        self.discriminator.backprop(loss)

    def backprop_generator(self, loss):
        # Backpropagate and learn generator
        self.generator.backprop(loss)

    def compute_kld(self, p, q):
        # Helper function that calculates
        return F.reshape(F.sum(p * (F.log(p + 1e-16) - F.log(q + 1e-16)), axis=1), (-1,

    def get_unit_vector(self, v):
        v /= (np.sqrt(np.sum(v ** 2, axis=1)).reshape((-1, 1)) + 1e-16)
        return v

    def compute_lds(self, x, xi=10, eps=1, Ip=1):
        x = self.to_variable(x)
        y1, _ = self.discriminate(x, apply_softmax=True)
        y1.unchain_backward()
        d = self.to_variable(self.get_unit_vector(np.random.normal(size=x.shape).astype(

        for i in xrange(Ip):
            y2, _ = self.discriminate(x + xi * d, apply_softmax=True)
            kld = F.sum(self.compute_kld(y1, y2))
            kld.backward()
            d = self.to_variable(self.get_unit_vector(self.to_numpy(d.grad)))

        y2, _ = self.discriminate(x + eps * d, apply_softmax=True)
        return -self.compute_kld(y1, y2)

    def load(self, dir=None):
        if dir is None:
            raise Exception()
        self.generator.load(dir + "/generator.hdf5")
        self.discriminator.load(dir + "/discriminator.hdf5")

    def save(self, dir=None):
        if dir is None:
            raise Exception()
        try:
            os.mkdir(dir)
        except:
            pass
        self.generator.save(dir + "/generator.hdf5")
        self.discriminator.save(dir + "/discriminator.hdf5")
```

1.4 MNIST

Next we will look how they use these general classes to implement MNIST training ### Args The provided code assumes that the terminal is used and uses args.py to parse the arguments. Since i work in a notebook i solve it by manually crafting an args object

```
In [6]: sys.path.append(os.path.join(os.path.split(os.getcwd())[0],'improved-gan/train_mnist'))
        args = Object()
        args.model_dir = 'mnist'
        args.gpu_device = 0
        args.seed = None
        args.plot_dir = 'mnist-plot'
        args.num_labeled = 100
```

1.4.1 Model

This defines the generator and descriminator file. It saves the params and model to json and afterwards loads from this again. This makes it easy to change the model to your needs.

The discriminator gets a 28x28 image input flattened to a 756x1 vector and outputs one vector of 10x1: a value for each of the classes. It consists of three fully connected hidden layers 756x1000x500x250x10.

The generator gets a 50x1 vector latent dimensional input(random variable) and produces a 756 output(the 28x28 image). It also has two hidden layers: 50x500x500x756

```
In [7]: from sequential import Sequential
        from sequential.layers import Linear, BatchNormalization, MinibatchDiscrimination
        from sequential.functions import Activation, dropout, gaussian_noise, softmax

        # load params.json
        try:
            os.mkdir(args.model_dir)
        except:
            pass

        # data
        image_width = 28
        image_height = image_width
        ndim_latent_code = 50 # 50 latent dimensional input

        # specify discriminator
        discriminator_sequence_filename = args.model_dir + "/discriminator.json"

        if os.path.isfile(discriminator_sequence_filename):
            print "loading", discriminator_sequence_filename
            with open(discriminator_sequence_filename, "r") as f:
                try:
                    params = json.load(f)
                except Exception as e:
                    raise Exception("could not load {}".format(discriminator_sequence_filename))
        else:
            config = ClassifierParams()
            config.ndim_input = image_width * image_height
            config.ndim_output = 10
            config.weight_init_std = 1
            config.weight_initializer = "GlorotNormal"
            config.use_weightnorm = False
            config.nonlinearity = "softplus"
            config.optimizer = "Adam"
```

```
    config.learning_rate = 0.001
    config.momentum = 0.5
    config.gradient_clipping = 10
    config.weight_decay = 0
    config.use_feature_matching = True
    config.use_minibatch_discrimination = False

    discriminator = Sequential(weight_initializer=config.weight_initializer, weight_init
    discriminator.add(gaussian_noise(std=0.3))
    discriminator.add(Linear(config.ndim_input, 1000, use_weightnorm=config.use_weightno
    discriminator.add(gaussian_noise(std=0.5))
    discriminator.add(Activation(config.nonlinearity))
    # discriminator.add(BatchNormalization(1000))
    discriminator.add(Linear(None, 500, use_weightnorm=config.use_weightnorm))
    discriminator.add(gaussian_noise(std=0.5))
    discriminator.add(Activation(config.nonlinearity))
    # discriminator.add(BatchNormalization(500))
    discriminator.add(Linear(None, 250, use_weightnorm=config.use_weightnorm))
    discriminator.add(gaussian_noise(std=0.5))
    discriminator.add(Activation(config.nonlinearity))
    # discriminator.add(BatchNormalization(250))
    if config.use_minibatch_discrimination:
        discriminator.add(MinibatchDiscrimination(None, num_kernels=50, ndim_kernel=5))
    discriminator.add(Linear(None, config.ndim_output, use_weightnorm=config.use_weightn
    # no need to add softmax() here
    discriminator.build()

    params = {
        "config": config.to_dict(),
        "model": discriminator.to_dict(),
    }

    with open(discriminator_sequence_filename, "w") as f:
        json.dump(params, f, indent=4, sort_keys=True, separators=(',', ': '))

discriminator_params = params

# specify generator
generator_sequence_filename = args.model_dir + "/generator.json"

if os.path.isfile(generator_sequence_filename):
    print "loading", generator_sequence_filename
    with open(generator_sequence_filename, "r") as f:
        try:
            params = json.load(f)
        except:
            raise Exception("could not load {}".format(generator_sequence_filename))
else:
```

```
            config = GeneratorParams()
            config.ndim_input = ndim_latent_code
            config.ndim_output = image_width * image_height
            config.distribution_output = "tanh"
            config.use_weightnorm = False
            config.weight_init_std = 1
            config.weight_initializer = "GlorotNormal"
            config.nonlinearity = "relu"
            config.optimizer = "Adam"
            config.learning_rate = 0.001
            config.momentum = 0.5
            config.gradient_clipping = 10
            config.weight_decay = 0

            # generator
            generator = Sequential(weight_initializer=config.weight_initializer, weight_init_std
            generator.add(Linear(config.ndim_input, 500, use_weightnorm=config.use_weightnorm))
            generator.add(BatchNormalization(500))
            generator.add(Activation(config.nonlinearity))
            generator.add(Linear(None, 500, use_weightnorm=config.use_weightnorm))
            generator.add(BatchNormalization(500))
            generator.add(Activation(config.nonlinearity))
            generator.add(Linear(None, config.ndim_output, use_weightnorm=config.use_weightnorm)
            if config.distribution_output == "sigmoid":
                generator.add(Activation("sigmoid"))
            if config.distribution_output == "tanh":
                generator.add(Activation("tanh"))
            generator.build()

            params = {
                "config": config.to_dict(),
                "model": generator.to_dict(),
            }

            with open(generator_sequence_filename, "w") as f:
                json.dump(params, f, indent=4, sort_keys=True, separators=(',', ': '))

        generator_params = params

        gan = GAN(discriminator_params, generator_params)
        gan.load(args.model_dir)

        if args.gpu_device != -1:
            cuda.get_device(args.gpu_device).use()
            gan.to_gpu()
```

```
loading mnist/discriminator.json
loading mnist/generator.json
loading mnist/generator.hdf5 ...
loading mnist/discriminator.hdf5 ...
```

1.5 Dataset

Creates a semi-supervised dataset

```
In [8]: import mnist_tools
        def load_train_images():
            return mnist_tools.load_train_images()


        def load_test_images():
            return mnist_tools.load_test_images()

        def binarize_data(x):
            threshold = np.random.uniform(size=x.shape)
            return np.where(threshold < x, 1.0, 0.0).astype(np.float32)


        def create_semisupervised(images, labels, num_validation_data=10000, num_labeled_data=10
                                  seed=0):
            if len(images) < num_validation_data + num_labeled_data:
                raise Exception("len(images) < num_validation_data + num_labeled_data")
            training_labeled_x = []
            training_unlabeled_x = []
            validation_x = []
            validation_labels = []
            training_labels = []
            indices_for_label = {}
            num_data_per_label = int(num_labeled_data / num_types_of_label)
            num_unlabeled_data = len(images) - num_validation_data - num_labeled_data

            np.random.seed(seed)
            indices = np.arange(len(images))
            np.random.shuffle(indices)

            def check(index):
                label = labels[index]
                if label not in indices_for_label:
                    indices_for_label[label] = []
                    return True
                if len(indices_for_label[label]) < num_data_per_label:
                    for i in indices_for_label[label]:
                        if i == index:
                            return False
                    return True
                return False

            for n in xrange(len(images)):
                index = indices[n]
                if check(index):
                    indices_for_label[labels[index]].append(index)
                    training_labeled_x.append(images[index])
                    training_labels.append(labels[index])
                else:
                    if len(training_unlabeled_x) < num_unlabeled_data:
                        training_unlabeled_x.append(images[index])
                    else:
                        validation_x.append(images[index])
                        validation_labels.append(labels[index])

            # reset seed
            np.random.seed()

            return training_labeled_x, training_labels, training_unlabeled_x, validation_x, vali


        def sample_labeled_data(images, labels, batchsize, ndim_x, ndim_y, binarize=True):
            image_batch = np.zeros((batchsize, ndim_x), dtype=np.float32)
            label_onehot_batch = np.zeros((batchsize, ndim_y), dtype=np.float32)
            label_id_batch = np.zeros((batchsize,), dtype=np.int32)
            indices = np.random.choice(np.arange(len(images), dtype=np.int32), size=batchsize, r
            for j in range(batchsize):
                data_index = indices[j]
                img = images[data_index].astype(np.float32) / 255.0
                image_batch[j] = img.reshape((ndim_x,))
                label_onehot_batch[j, labels[data_index]] = 1
                label_id_batch[j] = labels[data_index]
            if binarize:
                image_batch = binarize_data(image_batch)
            # [0, 1] -> [-1, 1]
            image_batch = image_batch * 2.0 - 1.0
            return image_batch, label_onehot_batch, label_id_batch


        def sample_unlabeled_data(images, batchsize, ndim_x, binarize=True):
            image_batch = np.zeros((batchsize, ndim_x), dtype=np.float32)
            indices = np.random.choice(np.arange(len(images), dtype=np.int32), size=batchsize, r
            for j in range(batchsize):
                data_index = indices[j]
                img = images[data_index].astype(np.float32) / 255.0
                image_batch[j] = img.reshape((ndim_x,))
            if binarize:
                image_batch = binarize_data(image_batch)
                # [0, 1] -> [-1, 1]
                image_batch = image_batch * 2.0 - 1.0
                return image_batch
```

1.5.1 training

```
In [9]: import sys, os
        import numpy as np
        import visualizer
        from progress import Progress
        import pandas as pd
        sys.path.append(os.path.split(os.getcwd())[0])


        def plot(filename="gen"):
            try:
                os.mkdir(args.plot_dir)
            except:
                pass

            x_fake = gan.generate_x(100, test=True, as_numpy=True)
            x_fake = (x_fake + 1.0) / 2.0
            visualizer.tile_binary_images(x_fake.reshape((-1, 28, 28)), dir=args.plot_dir, filen

In [10]: def get_learning_rate_for_epoch(epoch):
             if epoch < 10:
                 return 0.001
             if epoch < 50:
                 return 0.0003
             return 0.0001


         def main():
             # load MNIST images
             images, labels = load_train_images()

             # config
             discriminator_config = gan.config_discriminator
             generator_config = gan.config_generator

             # settings
             # _l -> labeled
             # _u -> unlabeled
             # _g -> generated
             max_epoch = 1000
             num_trains_per_epoch = 500
             plot_interval = 5
             batchsize_l = 100
```

```
batchsize_u = 100
batchsize_g = batchsize_u

# seed
np.random.seed(args.seed)
if args.gpu_device != -1:
    cuda.cupy.random.seed(args.seed)

# save validation accuracy per epoch
csv_results = []

# create semi-supervised split
num_validation_data = 10000
num_labeled_data = args.num_labeled
if batchsize_l > num_labeled_data:
    batchsize_l = num_labeled_data

training_images_l, training_labels_l, training_images_u, validation_images, validat
    images, labels, num_validation_data, num_labeled_data, discriminator_config.ndi
print training_labels_l

# training
progress = Progress()
for epoch in xrange(1, max_epoch):
    progress.start_epoch(epoch, max_epoch)
    sum_loss_supervised = 0
    sum_loss_unsupervised = 0
    sum_loss_adversarial = 0
    sum_dx_labeled = 0
    sum_dx_unlabeled = 0
    sum_dx_generated = 0

    gan.update_learning_rate(get_learning_rate_for_epoch(epoch))

    for t in xrange(num_trains_per_epoch):
        # sample from data distribution
        images_l, label_onehot_l, label_ids_l = sample_labeled_data(training_images
                                                                     batchsi
                                                                     discrim
                                                                     discrim
                                                                     binariz
        images_u = sample_unlabeled_data(training_images_u, batchsize_u, discrimina
                                                  binarize=False)
        images_g = gan.generate_x(batchsize_g)
        images_g.unchain_backward()

        # supervised loss
        py_x_l, activations_l = gan.discriminate(images_l, apply_softmax=False)
```

```
loss_supervised = F.softmax_cross_entropy(py_x_l, gan.to_variable(label_ids

log_zx_l = F.logsumexp(py_x_l, axis=1)
log_dx_l = log_zx_l - F.softplus(log_zx_l)
dx_l = F.sum(F.exp(log_dx_l)) / batchsize_l

# unsupervised loss
# D(x) = Z(x) / {Z(x) + 1}, where Z(x) = \sum_{k=1}^K exp(l_k(x))
# softplus(x) := log(1 + exp(x))
# logD(x) = logZ(x) - log(Z(x) + 1)
#                    = logZ(x) - log(exp(log(Z(x))) + 1)
#                    = logZ(x) - softplus(logZ(x))
# 1 - D(x) = 1 / {Z(x) + 1}
# log{1 - D(x)} = log1 - log(Z(x) + 1)
#                                    = -log(exp(log(Z(x))) + 1)
#                                    = -softplus(logZ(x))
py_x_u, _ = gan.discriminate(images_u, apply_softmax=False)
log_zx_u = F.logsumexp(py_x_u, axis=1)
log_dx_u = log_zx_u - F.softplus(log_zx_u)
dx_u = F.sum(F.exp(log_dx_u)) / batchsize_u
loss_unsupervised = -F.sum(log_dx_u) / batchsize_u  # minimize negative log
py_x_g, _ = gan.discriminate(images_g, apply_softmax=False)
log_zx_g = F.logsumexp(py_x_g, axis=1)
loss_unsupervised += F.sum(F.softplus(log_zx_g)) / batchsize_u  # minimize 

# update discriminator
gan.backprop_discriminator(loss_supervised + loss_unsupervised)

# adversarial loss
images_g = gan.generate_x(batchsize_g)
py_x_g, activations_g = gan.discriminate(images_g, apply_softmax=False)
log_zx_g = F.logsumexp(py_x_g, axis=1)
log_dx_g = log_zx_g - F.softplus(log_zx_g)
dx_g = F.sum(F.exp(log_dx_g)) / batchsize_g
loss_adversarial = -F.sum(log_dx_g) / batchsize_u  # minimize negative logD

# feature matching
if discriminator_config.use_feature_matching:
    features_true = activations_l[-1]
    features_true.unchain_backward()
    if batchsize_l != batchsize_g:
        images_g = gan.generate_x(batchsize_l)
        _, activations_g = gan.discriminate(images_g, apply_softmax=False)
    features_fake = activations_g[-1]
    loss_adversarial += F.mean_squared_error(features_true, features_fake)

# update generator
gan.backprop_generator(loss_adversarial)
```

```
            sum_loss_supervised += float(loss_supervised.data)
            sum_loss_unsupervised += float(loss_unsupervised.data)
            sum_loss_adversarial += float(loss_adversarial.data)
            sum_dx_labeled += float(dx_l.data)
            sum_dx_unlabeled += float(dx_u.data)
            sum_dx_generated += float(dx_g.data)
            if t % 10 == 0:
                progress.show(t, num_trains_per_epoch, {})

        gan.save(args.model_dir)

        # validation
        images_l, _, label_ids_l = sample_labeled_data(validation_images, validation_la
                                                           num_validation_data, dis
                                                           discriminator_config.ndi
        images_l_segments = np.split(images_l, num_validation_data // 500)
        label_ids_l_segments = np.split(label_ids_l, num_validation_data // 500)
        sum_accuracy = 0
        for images_l, label_ids_l in zip(images_l_segments, label_ids_l_segments):
            y_distribution, _ = gan.discriminate(images_l, apply_softmax=True, test=Tru
            accuracy = F.accuracy(y_distribution, gan.to_variable(label_ids_l))
            sum_accuracy += float(accuracy.data)
        validation_accuracy = sum_accuracy / len(images_l_segments)

        progress.show(num_trains_per_epoch, num_trains_per_epoch, {
            "loss_l": sum_loss_supervised / num_trains_per_epoch,
            "loss_u": sum_loss_unsupervised / num_trains_per_epoch,
            "loss_g": sum_loss_adversarial / num_trains_per_epoch,
            "dx_l": sum_dx_labeled / num_trains_per_epoch,
            "dx_u": sum_dx_unlabeled / num_trains_per_epoch,
            "dx_g": sum_dx_generated / num_trains_per_epoch,
            "accuracy": validation_accuracy,
        })

        # write accuracy to csv
        csv_results.append([epoch, validation_accuracy, progress.get_total_time()])
        data = pd.DataFrame(csv_results)
        data.columns = ["epoch", "accuracy", "min"]
        data.to_csv("{}/result.csv".format(args.model_dir))

        if epoch % plot_interval == 0 or epoch == 1:
            plot(filename="epoch_{}_time_{}min".format(epoch, progress.get_total_time()
```

```
In [ ]: main()
```

Training supervised

January 10, 2017

1 Training supervised

In this notebook I will train a simple supervised MNIST model, so that we have comparison for our model trained with the GAN. The model is based on https://github.com/pfnet/chainer/blob/master/examples/mnist/train_mnist.py i only did some small adaptions to make it more similar to the GAN example.

```
In [2]: from __future__ import print_function
        import argparse
        import sys,os
        import numpy as np
        import chainer
        import chainer.functions as F
        import chainer.links as L
        from chainer import training
        from chainer.training import extensions
```

This the network used. I added one hidden layer and set the number of hidden units to the number used by the GAN. Furthermore I changed the relu activation function to the elu function as this is also the one used by the GAN.

```
In [3]: # Network definition
        class MLP(chainer.Chain):

            def __init__(self, n_out):
                super(MLP, self).__init__(
                    # the size of the inputs to each layer will be inferred
                    l1=L.Linear(None, 1000),  # n_in -> n_units
                    l2=L.Linear(None, 500),  # n_units -> n_units
                    l3=L.Linear(None, 250),  # n_units -> n_units
                    l4=L.Linear(None, n_out),  # n_units -> n_out
                )

            def __call__(self, x):
                h1 = F.elu(self.l1(x))
                h2 = F.elu(self.l2(h1))
                h3 = F.elu(self.l3(h2))
                return self.l4(h3)
```

We can largely re-use the batch script from the tutorial. I overwrite arguments variable such that argparse works correctly. I also make sure the model it self is serialized, not only the complete trainer. This makes life a little easier once we want to load the model in order to create adversarial images.

```
In [4]: GPU = -1
        sys.argv=['main','-b','128','-g',str(GPU)]

In [5]: def main():
            parser = argparse.ArgumentParser(description='Chainer example: MNIST')
            parser.add_argument('--batchsize', '-b', type=int, default=100,
                                help='Number of images in each mini-batch')
            parser.add_argument('--epoch', '-e', type=int, default=20,
                                help='Number of sweeps over the dataset to train')
            parser.add_argument('--gpu', '-g', type=int, default=-1,
                                help='GPU ID (negative value indicates CPU)')
            parser.add_argument('--out', '-o', default='result',
                                help='Directory to output the result')
            parser.add_argument('--resume', '-r', default='',
                                help='Resume the training from snapshot')
            args = parser.parse_args()

            print('GPU: {}'.format(args.gpu))
            print('# Minibatch-size: {}'.format(args.batchsize))
            print('# epoch: {}'.format(args.epoch))
            print('')

            # Set up a neural network to train
            # Classifier reports softmax cross entropy loss and accuracy at every
            # iteration, which will be used by the PrintReport extension below.
            model = L.Classifier(MLP( 10))
            if args.gpu >= 0:
                chainer.cuda.get_device(args.gpu).use()  # Make a specified GPU current
                model.to_gpu()  # Copy the model to the GPU

            # Setup an optimizer
            optimizer = chainer.optimizers.Adam()
            optimizer.setup(model)

            # Load the MNIST dataset
            train, test = chainer.datasets.get_mnist()

            train_iter = chainer.iterators.SerialIterator(train, args.batchsize)
            test_iter = chainer.iterators.SerialIterator(test, args.batchsize,
                                                         repeat=False, shuffle=False)

            # Set up a trainer
            updater = training.StandardUpdater(train_iter, optimizer, device=args.gpu)
            trainer = training.Trainer(updater, (args.epoch, 'epoch'), out=args.out)

            # Evaluate the model with the test dataset for each epoch
            trainer.extend(extensions.Evaluator(test_iter, model, device=args.gpu))

            # Dump a computational graph from 'loss' variable at the first iteration
            # The "main" refers to the target link of the "main" optimizer.
            trainer.extend(extensions.dump_graph('main/loss'))

            # Take a snapshot at each epoch
            trainer.extend(extensions.snapshot(), trigger=(args.epoch, 'epoch'))

            # Write a log of evaluation statistics for each epoch
            trainer.extend(extensions.LogReport())

            # Print selected entries of the log to stdout
            # Here "main" refers to the target link of the "main" optimizer again, and
            # "validation" refers to the default name of the Evaluator extension.
            # Entries other than 'epoch' are reported by the Classifier link, called by
            # either the updater or the evaluator.
            trainer.extend(extensions.PrintReport(
                ['epoch', 'main/loss', 'validation/main/loss',
                 'main/accuracy', 'validation/main/accuracy', 'elapsed_time']))

            # Print a progress bar to stdout
            trainer.extend(extensions.ProgressBar())

            if args.resume:
                # Resume from a snapshot
                chainer.serializers.load_npz(args.resume, trainer)

            # Run the training
            trainer.run()

            #export model so we can load it in our model without having to load the full trainer
            chainer.serializers.save_npz(os.path.join(args.out,'model.npz'),model)
```

```
In [6]: main()
```

```
GPU: -1
# Minibatch-size: 128
# epoch: 20

epoch       main/loss   validation/main/loss  main/accuracy  validation/main/accuracy  elapsed_t
     total [...]  1.07%
this epoch [##########...] 21.33%
       100 iter, 0 epoch / 20 epochs
       inf iters/sec. Estimated time to finish: 0:00:00.
     total [#...]  2.13%
this epoch [#####################...] 42.67%
       200 iter, 0 epoch / 20 epochs
    26.321 iters/sec. Estimated time to finish: 0:05:48.577524.
     total [#...]  3.20%
this epoch [################################...] 64.00%
       300 iter, 0 epoch / 20 epochs
     26.95 iters/sec. Estimated time to finish: 0:05:36.735091.
     total [##...]  4.27%
this epoch [##########################################...] 85.33%
       400 iter, 0 epoch / 20 epochs
    26.835 iters/sec. Estimated time to finish: 0:05:34.450026.
1           0.244283    0.134864              0.92579        0.958267                          18.3368
     total [##...]  5.33%
this epoch [###...]  6.67%
       500 iter, 1 epoch / 20 epochs
    25.476 iters/sec. Estimated time to finish: 0:05:48.361790.
     total [###...]  6.40%
this epoch [##############...] 28.00%
       600 iter, 1 epoch / 20 epochs
     25.78 iters/sec. Estimated time to finish: 0:05:40.385547.
     total [###...]  7.47%
this epoch [########################...] 49.33%
       700 iter, 1 epoch / 20 epochs
    25.814 iters/sec. Estimated time to finish: 0:05:36.057397.
     total [####...]  8.53%
this epoch [###################################...] 70.67%
       800 iter, 1 epoch / 20 epochs
    25.895 iters/sec. Estimated time to finish: 0:05:31.146776.
     total [####...]  9.60%
this epoch [##############################################...] 92.00%
       900 iter, 1 epoch / 20 epochs
    26.061 iters/sec. Estimated time to finish: 0:05:25.194457.
2           0.117284    0.0983768             0.963553       0.96964                           36.7813
     total [#####...] 10.67%
this epoch [######...] 13.33%
      1000 iter, 2 epoch / 20 epochs
    25.541 iters/sec. Estimated time to finish: 0:05:27.907109.
     total [#####...] 11.73%
this epoch [#################...] 34.67%
      1100 iter, 2 epoch / 20 epochs
    25.658 iters/sec. Estimated time to finish: 0:05:22.507832.
     total [######...] 12.80%
this epoch [############################...] 56.00%
      1200 iter, 2 epoch / 20 epochs
    25.742 iters/sec. Estimated time to finish: 0:05:17.569171.
     total [######...] 13.87%
this epoch [######################################...] 77.33%
```

```
      1300 iter, 2 epoch / 20 epochs
    25.793 iters/sec. Estimated time to finish: 0:05:13.073503.
     total [#######...] 14.93%
this epoch [#################################################.] 98.67%
      1400 iter, 2 epoch / 20 epochs
    25.879 iters/sec. Estimated time to finish: 0:05:08.166380.
3           0.0812357   0.109512              0.974447       0.964498                         55.0971
     total [########...] 16.00%
this epoch [##########...] 20.00%
      1500 iter, 3 epoch / 20 epochs
    25.536 iters/sec. Estimated time to finish: 0:05:08.385663.
     total [########...] 17.07%
this epoch [####################...] 41.33%
      1600 iter, 3 epoch / 20 epochs
    25.636 iters/sec. Estimated time to finish: 0:05:03.289932.
     total [#########...] 18.13%
this epoch [###############################...] 62.67%
      1700 iter, 3 epoch / 20 epochs
     25.71 iters/sec. Estimated time to finish: 0:04:58.525193.
     total [#########...] 19.20%
this epoch [##########################################...] 84.00%
      1800 iter, 3 epoch / 20 epochs
    25.774 iters/sec. Estimated time to finish: 0:04:53.900001.
4           0.0667263   0.0792326             0.978983       0.97587                          73.3563
     total [##########...] 20.27%
this epoch [##...]  5.33%
      1900 iter, 4 epoch / 20 epochs
    25.518 iters/sec. Estimated time to finish: 0:04:52.931637.
     total [##########...] 21.33%
this epoch [#############...] 26.67%
      2000 iter, 4 epoch / 20 epochs
    25.507 iters/sec. Estimated time to finish: 0:04:49.134526.
     total [###########...] 22.40%
this epoch [########################...] 48.00%
      2100 iter, 4 epoch / 20 epochs
    25.437 iters/sec. Estimated time to finish: 0:04:46.001569.
     total [###########...] 23.47%
this epoch [##################################...] 69.33%
      2200 iter, 4 epoch / 20 epochs
    25.457 iters/sec. Estimated time to finish: 0:04:41.844448.
     total [############...] 24.53%
this epoch [#############################################...] 90.67%
      2300 iter, 4 epoch / 20 epochs
    25.479 iters/sec. Estimated time to finish: 0:04:37.677648.
5           0.055902    0.0999595             0.98176        0.971321                         92.7986
     total [############...] 25.60%
this epoch [######...] 12.00%
      2400 iter, 5 epoch / 20 epochs
    25.216 iters/sec. Estimated time to finish: 0:04:36.612318.
     total [#############...] 26.67%
this epoch [################...] 33.33%
      2500 iter, 5 epoch / 20 epochs
    25.262 iters/sec. Estimated time to finish: 0:04:32.151449.
     total [#############...] 27.73%
this epoch [###########################...] 54.67%
      2600 iter, 5 epoch / 20 epochs
    25.363 iters/sec. Estimated time to finish: 0:04:27.121076.
     total [##############...] 28.80%
this epoch [######################################...] 76.00%
      2700 iter, 5 epoch / 20 epochs
    25.458 iters/sec. Estimated time to finish: 0:04:22.194955.
     total [##############...] 29.87%
this epoch [################################################..] 97.33%
      2800 iter, 5 epoch / 20 epochs
    25.555 iters/sec. Estimated time to finish: 0:04:17.285417.
6           0.0457681   0.09037               0.985008       0.975178                  110.656
     total [###############...] 30.93%
this epoch [#########...] 18.67%
      2900 iter, 6 epoch / 20 epochs
    25.445 iters/sec. Estimated time to finish: 0:04:14.471332.
     total [################...] 32.00%
this epoch [####################...] 40.00%
      3000 iter, 6 epoch / 20 epochs
    25.524 iters/sec. Estimated time to finish: 0:04:09.761926.
     total [################...] 33.07%
this epoch [##############################...] 61.33%
      3100 iter, 6 epoch / 20 epochs
    25.603 iters/sec. Estimated time to finish: 0:04:05.091238.
     total [#################...] 34.13%
this epoch [#########################################...] 82.67%
      3200 iter, 6 epoch / 20 epochs
    25.667 iters/sec. Estimated time to finish: 0:04:00.580255.
7           0.0434809   0.0974111             0.986357       0.975574                  128.255
     total [#################...] 35.20%
this epoch [##...]  4.00%
      3300 iter, 7 epoch / 20 epochs
    25.582 iters/sec. Estimated time to finish: 0:03:57.469965.
     total [##################...] 36.27%
this epoch [############...] 25.33%
      3400 iter, 7 epoch / 20 epochs
    25.639 iters/sec. Estimated time to finish: 0:03:53.040354.
     total [##################...] 37.33%
this epoch [#######################...] 46.67%
      3500 iter, 7 epoch / 20 epochs
    25.705 iters/sec. Estimated time to finish: 0:03:48.553748.
     total [###################...] 38.40%
```

```
this epoch [##################################...] 68.00%
      3600 iter, 7 epoch / 20 epochs
    25.775 iters/sec. Estimated time to finish: 0:03:44.050599.
     total [###################...] 39.47%
this epoch [############################################...] 89.33%
      3700 iter, 7 epoch / 20 epochs
    25.841 iters/sec. Estimated time to finish: 0:03:39.612698.
8           0.0420835   0.0999922             0.986195       0.974684                          145.627
     total [####################...] 40.53%
this epoch [#####...] 10.67%
      3800 iter, 8 epoch / 20 epochs
    25.754 iters/sec. Estimated time to finish: 0:03:36.469159.
     total [####################...] 41.60%
this epoch [################...] 32.00%
      3900 iter, 8 epoch / 20 epochs
    25.782 iters/sec. Estimated time to finish: 0:03:32.356101.
     total [#####################...] 42.67%
this epoch [##########################...] 53.33%
      4000 iter, 8 epoch / 20 epochs
    25.813 iters/sec. Estimated time to finish: 0:03:28.229367.
     total [#####################...] 43.73%
this epoch [#####################################...] 74.67%
      4100 iter, 8 epoch / 20 epochs
    25.867 iters/sec. Estimated time to finish: 0:03:23.931351.
     total [######################...] 44.80%
this epoch [################################################..] 96.00%
      4200 iter, 8 epoch / 20 epochs
    25.905 iters/sec. Estimated time to finish: 0:03:19.769634.
9           0.0331907   0.0893161             0.989872       0.98032                           163.583
     total [######################...] 45.87%
this epoch [########...] 17.33%
      4300 iter, 9 epoch / 20 epochs
    25.803 iters/sec. Estimated time to finish: 0:03:16.684602.
     total [#######################...] 46.93%
this epoch [###################...] 38.67%
      4400 iter, 9 epoch / 20 epochs
    25.791 iters/sec. Estimated time to finish: 0:03:12.897263.
     total [########################...] 48.00%
this epoch [##############################...] 60.00%
      4500 iter, 9 epoch / 20 epochs
    25.784 iters/sec. Estimated time to finish: 0:03:09.067370.
     total [########################...] 49.07%
this epoch [########################################...] 81.33%
      4600 iter, 9 epoch / 20 epochs
     25.84 iters/sec. Estimated time to finish: 0:03:04.788844.
10          0.0262785   0.0993367             0.991405       0.97676                           181.978
     total [#########################...] 50.13%
this epoch [#...]  2.67%
```

```
      4700 iter, 10 epoch / 20 epochs
    25.758 iters/sec. Estimated time to finish: 0:03:01.499393.
     total [#########################...] 51.20%
this epoch [############...] 24.00%
      4800 iter, 10 epoch / 20 epochs
     25.72 iters/sec. Estimated time to finish: 0:02:57.880020.
     total [##########################...] 52.27%
this epoch [######################...] 45.33%
      4900 iter, 10 epoch / 20 epochs
    25.749 iters/sec. Estimated time to finish: 0:02:53.792352.
     total [##########################...] 53.33%
this epoch [#################################...] 66.67%
      5000 iter, 10 epoch / 20 epochs
    25.751 iters/sec. Estimated time to finish: 0:02:49.894712.
     total [###########################...] 54.40%
this epoch [############################################...] 88.00%
      5100 iter, 10 epoch / 20 epochs
    25.769 iters/sec. Estimated time to finish: 0:02:45.897259.
11          0.0353472   0.0900451             0.989556       0.979134                  200.662
     total [###########################...] 55.47%
this epoch [####...]  9.33%
      5200 iter, 11 epoch / 20 epochs
    25.683 iters/sec. Estimated time to finish: 0:02:42.556507.
     total [############################...] 56.53%
this epoch [###############...] 30.67%
      5300 iter, 11 epoch / 20 epochs
    25.683 iters/sec. Estimated time to finish: 0:02:38.662478.
     total [############################...] 57.60%
this epoch [##########################...] 52.00%
      5400 iter, 11 epoch / 20 epochs
    25.674 iters/sec. Estimated time to finish: 0:02:34.826821.
     total [#############################...] 58.67%
this epoch [####################################...] 73.33%
      5500 iter, 11 epoch / 20 epochs
    25.693 iters/sec. Estimated time to finish: 0:02:30.819362.
     total [#############################...] 59.73%
this epoch [###############################################...] 94.67%
      5600 iter, 11 epoch / 20 epochs
    25.719 iters/sec. Estimated time to finish: 0:02:26.778166.
12          0.0298706   0.0779276             0.990685       0.980419                  219.344
     total [##############################...] 60.80%
this epoch [########...] 16.00%
      5700 iter, 12 epoch / 20 epochs
    25.643 iters/sec. Estimated time to finish: 0:02:23.314846.
     total [##############################...] 61.87%
this epoch [##################...] 37.33%
      5800 iter, 12 epoch / 20 epochs
    25.613 iters/sec. Estimated time to finish: 0:02:19.575762.
```

```
     total [###############################...] 62.93%
this epoch [#############################...] 58.67%
      5900 iter, 12 epoch / 20 epochs
    25.604 iters/sec. Estimated time to finish: 0:02:15.718700.
     total [################################...] 64.00%
this epoch [########################################...] 80.00%
      6000 iter, 12 epoch / 20 epochs
    25.614 iters/sec. Estimated time to finish: 0:02:11.761904.
13          0.020571    0.0966734             0.99322        0.979727                  238.539
     total [################################...] 65.07%
this epoch [...]  1.33%
      6100 iter, 13 epoch / 20 epochs
    25.538 iters/sec. Estimated time to finish: 0:02:08.238947.
     total [#################################...] 66.13%
this epoch [###########...] 22.67%
      6200 iter, 13 epoch / 20 epochs
    25.531 iters/sec. Estimated time to finish: 0:02:04.358427.
     total [#################################...] 67.20%
this epoch [######################...] 44.00%
      6300 iter, 13 epoch / 20 epochs
    25.532 iters/sec. Estimated time to finish: 0:02:00.435402.
     total [##################################...] 68.27%
this epoch [################################...] 65.33%
      6400 iter, 13 epoch / 20 epochs
    25.531 iters/sec. Estimated time to finish: 0:01:56.522944.
     total [##################################...] 69.33%
this epoch [###########################################...] 86.67%
      6500 iter, 13 epoch / 20 epochs
     25.53 iters/sec. Estimated time to finish: 0:01:52.612888.
14          0.0238655   0.10531               0.992837       0.979727                  257.815
     total [###################################...] 70.40%
this epoch [####...]  8.00%
      6600 iter, 14 epoch / 20 epochs
    25.443 iters/sec. Estimated time to finish: 0:01:49.067661.
     total [###################################...] 71.47%
this epoch [##############...] 29.33%
      6700 iter, 14 epoch / 20 epochs
    25.424 iters/sec. Estimated time to finish: 0:01:45.215001.
     total [####################################...] 72.53%
this epoch [#########################...] 50.67%
      6800 iter, 14 epoch / 20 epochs
    25.386 iters/sec. Estimated time to finish: 0:01:41.435426.
     total [####################################...] 73.60%
this epoch [####################################...] 72.00%
      6900 iter, 14 epoch / 20 epochs
    25.387 iters/sec. Estimated time to finish: 0:01:37.491685.
     total [#####################################...] 74.67%
this epoch [##############################################...] 93.33%
```

```
      7000 iter, 14 epoch / 20 epochs
    25.379 iters/sec. Estimated time to finish: 0:01:33.582260.
15          0.0242614   0.0889186             0.992637       0.980617                         277.649
     total [#####################################...] 75.73%
this epoch [#######...] 14.67%
      7100 iter, 15 epoch / 20 epochs
    25.307 iters/sec. Estimated time to finish: 0:01:29.894962.
     total [######################################...] 76.80%
this epoch [##################...] 36.00%
      7200 iter, 15 epoch / 20 epochs
    25.286 iters/sec. Estimated time to finish: 0:01:26.016076.
     total [######################################...] 77.87%
this epoch [############################...] 57.33%
      7300 iter, 15 epoch / 20 epochs
    25.279 iters/sec. Estimated time to finish: 0:01:22.084783.
     total [#######################################...] 78.93%
this epoch [#######################################...] 78.67%
      7400 iter, 15 epoch / 20 epochs
    25.267 iters/sec. Estimated time to finish: 0:01:18.165967.
16          0.0224246   0.10066               0.993289       0.980617                         297.804
     total [########################################...] 80.00%
this epoch [...]  0.00%
      7500 iter, 16 epoch / 20 epochs
    25.174 iters/sec. Estimated time to finish: 0:01:14.480345.
     total [########################################...] 81.07%
this epoch [##########...] 21.33%
      7600 iter, 16 epoch / 20 epochs
    25.137 iters/sec. Estimated time to finish: 0:01:10.613834.
     total [#########################################...] 82.13%
this epoch [#####################...] 42.67%
      7700 iter, 16 epoch / 20 epochs
     25.09 iters/sec. Estimated time to finish: 0:01:06.758727.
     total [#########################################...] 83.20%
this epoch [################################...] 64.00%
      7800 iter, 16 epoch / 20 epochs
    25.046 iters/sec. Estimated time to finish: 0:01:02.884261.
     total [##########################################...] 84.27%
this epoch [##########################################...] 85.33%
      7900 iter, 16 epoch / 20 epochs
    25.017 iters/sec. Estimated time to finish: 0:00:58.960830.
17          0.0237644   0.117908              0.992771       0.975672                         319.488
     total [##########################################...] 85.33%
this epoch [###...]  6.67%
      8000 iter, 17 epoch / 20 epochs
    24.921 iters/sec. Estimated time to finish: 0:00:55.174564.
     total [###########################################...] 86.40%
this epoch [##############...] 28.00%
      8100 iter, 17 epoch / 20 epochs
    24.861 iters/sec. Estimated time to finish: 0:00:51.285655.
     total [###########################################...] 87.47%
this epoch [########################...] 49.33%
      8200 iter, 17 epoch / 20 epochs
    24.807 iters/sec. Estimated time to finish: 0:00:47.366349.
     total [############################################...] 88.53%
this epoch [###################################...] 70.67%
      8300 iter, 17 epoch / 20 epochs
    24.791 iters/sec. Estimated time to finish: 0:00:43.363023.
     total [############################################...] 89.60%
this epoch [##############################################...] 92.00%
      8400 iter, 17 epoch / 20 epochs
    24.754 iters/sec. Estimated time to finish: 0:00:39.386819.
18          0.0223618   0.133069              0.993653       0.9732                     341.504
     total [#############################################...] 90.67%
this epoch [######...] 13.33%
      8500 iter, 18 epoch / 20 epochs
    24.672 iters/sec. Estimated time to finish: 0:00:35.464687.
     total [#############################################...] 91.73%
this epoch [#################...] 34.67%
      8600 iter, 18 epoch / 20 epochs
    24.592 iters/sec. Estimated time to finish: 0:00:31.514571.
     total [##############################################...] 92.80%
this epoch [############################...] 56.00%
      8700 iter, 18 epoch / 20 epochs
    24.525 iters/sec. Estimated time to finish: 0:00:27.522851.
     total [##############################################...] 93.87%
this epoch [######################################...] 77.33%
      8800 iter, 18 epoch / 20 epochs
    24.507 iters/sec. Estimated time to finish: 0:00:23.462836.
     total [###############################################...] 94.93%
this epoch [#################################################.] 98.67%
      8900 iter, 18 epoch / 20 epochs
    24.479 iters/sec. Estimated time to finish: 0:00:19.404549.
19          0.0204874   0.114019              0.99427        0.981013                   364.441
     total [################################################..] 96.00%
this epoch [#########...] 20.00%
      9000 iter, 19 epoch / 20 epochs
    24.385 iters/sec. Estimated time to finish: 0:00:15.378441.
     total [################################################..] 97.07%
this epoch [####################...] 41.33%
      9100 iter, 19 epoch / 20 epochs
    24.375 iters/sec. Estimated time to finish: 0:00:11.282188.
     total [#################################################.] 98.13%
this epoch [###############################...] 62.67%
      9200 iter, 19 epoch / 20 epochs
    24.343 iters/sec. Estimated time to finish: 0:00:07.189067.
     total [#################################################.] 99.20%
```

```
this epoch [#########################################...] 84.00%
      9300 iter, 19 epoch / 20 epochs
      24.3 iters/sec. Estimated time to finish: 0:00:03.086451.
20          0.0194203   0.121105              0.994224       0.981013                  386.637
```

```
In [ ]:
```

Adversarial images

January 10, 2017

```
In [1]: GPU_DEVICE = -1#Please set the GPU device you want to use to execute this notebook -1 me
```

1 Adversarial images

In this notebook we will have a look at the vulnerability for adversarial examples of the networks learned in 'training GAN' and 'training supervised network' notebooks. First we will use the supervised model to generate adversarial examples and see how vulnerable this network is to these examples. Next we will use the same examples to test the vulnerability of the descriminator learned by the GAN.

```
In [2]: import math
        import json
        import numpy as np
        import chainer, os, collections, six, math, random, time, copy,sys
        from chainer import cuda, Variable, optimizers, serializers, function, optimizer, initia
        from chainer.utils import type_check
        from chainer import functions as F
        from chainer import links as L
        import matplotlib
        import matplotlib.pyplot as plt
        import pandas as pd
        from tqdm import tqdm
        %matplotlib inline
```

1.0.1 Helper functions and constants

Here i will define some helper functions that we might want to use for both the supervised network and the gan network

```
In [3]: # These are the epsilons value we will calculate adversarial images of i also generate t
        # This gives me the original images and make my life a little easier when obtaining the
        GRADIENT_STEPS = [0.0,.1,.25,1.]

        # Column names for the to be generated csvs
        COLS = ['index','org_index','loss','eps','correct','found_label','confidence'] + ['prob_

        def _build_labels(p, label,eps):
```

```
    """ Helper function to build label for under the image, shows the certainty
    with which the image is classified as a certain integer
    """
    return ("%d(%.2f%%)" % (label, p*100.0), "$\epsilon$:%.2f" % eps)

def _plot_image(i,ax, text_a,text_b):
    """ Plots a gray scale image with some text under it
    text_a is first line
    text_b is second line
    """
    i = i.reshape(28,28)
    ax.imshow(i,cmap='gray')
    ax.axis('off')
    ax.annotate(text_a, xy=(0, 0), xytext=(2, 32))
    ax.annotate(text_b, xy=(0, 0), xytext=(7, 36))

def plot_images(imgs,confidence,labels,eps=GRADIENT_STEPS):
    """ Plots multiple images with the probablity of the correct class and the probablit
    """
    ni = len(imgs)
    fig, axarr = plt.subplots(ncols=ni)
    for i in range(ni):
        correct_label = labels[i]
        p = confidence[i]
        text_a,text_b = _build_labels(p, correct_label,eps[i])
        _plot_image(imgs[i,:],axarr[i],text_a,text_b )


def statistics_from_df(df):
    """ Gets statistics( average confidence and accuracy) for every possible value of ep
    return als list of tuples (eps, average confidence, accuracy)
    """
    data = []
    for eps in GRADIENT_STEPS:
        # get all fields for this
        s = df[df.eps == eps]
        p = s[s.correct == s.found_label]

        acc = float(p.shape[0])/s.shape[0]
        data.append((eps, acc, np.average(s.confidence.tolist())))
    return data

def statistics_to_latex_table(data):
    """ Can be used to print data as table
    """
    print "\\begin{tabular}{|l|l|l|}\hline"
    print " & ".join(["\\textbf{"+x+"}" for x in ['type','accuracy','avg. confidence']])
    for d in data:
        X = list(d)
        print  " & ".join(["$\epsilon = %.2f$" % d[0]] +["%.3f" % x for x in X[1:]]) + "
    print "\end{tabular}"
```

1.1 Supervised network

Now load the normal supervised neural network and see how it works out ### Loading model

```
In [4]: # Network definition
        class MLP(chainer.Chain):

            def __init__(self, n_out):
                super(MLP, self).__init__(
                    # the size of the inputs to each layer will be inferred
                    l1=L.Linear(None, 1000),  # n_in -> n_units
                    l2=L.Linear(None, 500),  # n_units -> n_units
                    l3=L.Linear(None, 250),  # n_units -> n_units
                    l4=L.Linear(None, n_out),  # n_units -> n_out
                )

            def __call__(self, x):
                h1 = F.elu(self.l1(x))
                h2 = F.elu(self.l2(h1))
                h3 = F.elu(self.l3(h2))
                return self.l4(h3)

In [5]: model = L.Classifier(MLP( 10))
        resume = 'result/model.npz'
        if GPU_DEVICE >= 0:
            chainer.cuda.get_device(GPU_DEVICE).use()  # Make a specified GPU current
            model.to_gpu()  # Copy the model to the GPU

        chainer.serializers.load_npz(resume, model)
```

1.2 Some helper functions

```
In [6]: # Helper functions that help to translate between chainer variable and numpy variables
        def _to_variable(x):
            if isinstance(x, Variable) == False:
                x = Variable(x)
                if GPU_DEVICE > -1:
                    x.to_gpu()
            return x

        def _to_numpy(x):
            if isinstance(x, Variable) == True:
                x = x.data
            if isinstance(x, cuda.ndarray) == True:
```

```
        x = cuda.to_cpu(x)
    return x

def pass_forward_sup(ims, target_labels, calc_gradients = False):
    """ This passes images forward trough the network. Using the target_labels
    it will also calculate the loss. If calc_gradients is set to True it will
    calculate the gradiens with respect to the input

    Returns a tuple containing loss, probabilities and optionally gradients
    """
    ims = _to_variable(ims)

    target = _to_variable(target_labels)
    # predict loss
    loss = model(ims, target)
    if calc_gradients:
        # back propagate error
        loss.backward()

        return _to_numpy(loss), _to_numpy(F.softmax(model.y)), ims.grad
    else:
        # model.predictor contains our original model, without the classifier class
        return _to_numpy(loss), _to_numpy(F.softmax(model.predictor(ims)))

def create_adv_images(ims, sgrad, eps):
    """ Given original images, signed gradient matrix and a epsilon will
    calculate the new adversarial images.
    Will also make sure the matrix is constraint to a minimum of 0 and a max of 1
    """
    return np.minimum(
        np.ones(shape=ims.shape, dtype=np.float32),
        np.maximum(
            np.zeros(shape=ims.shape,dtype=np.float32),
            ims + np.multiply(eps,sgrad)
        )
    )
```

1.2.1 Generate adversarial images and a csv file with the results

We will now create the following numpy array: - images array all generated adversarial images, for every possible ϵ value - all_probs holding all found probalities for every class for every adversarial image - org_probs holding all originally found probabilties, only for the non adversarial images

We will also create a csv using pandas with the following columns: - index: Index in the images and all_probs array - org_index: index in the org_probs array and the MNIST dataset of the original image - loss: loss for this step - eps: Used epsilon value for this adversarial image -

correct: original label for this image - found_label: the label with the highest probablity for this image - Confidence: the probability of the found label - prob_0 t/m prob_10: probabilities found for all classes

This csv can be used to obtain results.

```
In [7]: # download mnist dataset
        train, test = chainer.datasets.get_mnist()

        # Number of original images to walk over
        N_IMAGES = len(test)
        #keep data for csv dataframe
        data = []

        #Keep index of last created adversarial image
        image_idx  = 0

        #empty array that can hold all generated adversarial images
        images = np.zeros((len(GRADIENT_STEPS)*N_IMAGES,28**2))
        #empty array that hold all probabiliteis for all genareted images
        all_probs = np.zeros((len(GRADIENT_STEPS)*N_IMAGES,10),dtype=np.float32)

        #empty array that keeps track of all original probabilities
        org_probs = np.zeros((N_IMAGES,10),dtype=np.float32)

        #Walk over the training image in batches of 128
        for i in range(0,int(math.ceil(N_IMAGES/128.))):
            #start indexes of this batch
            s = i * 128
            # ends s+128 or N_images
            e = min(s+128,N_IMAGES)
            n = e-s

            #dataset consists of list of tuples, first element of tuple contains image
            #second contains label
            ins = test[s:e]
            imgs = np.array([i[0] for i in ins])
            labs = np.array([i[1] for i in ins],dtype=np.int32)

            # Calculate original probs and gradient with respect to input
            loss, probs, gradients = pass_forward_sup(imgs, labs, True)

            # Adversarial images are generated using the signed gradient
            sgrad = np.sign(gradients)

            # Save original probabilities, for later display
            org_probs[s:e,:] = probs

            # we will generate the adversarial images in
            for step in GRADIENT_STEPS:
                # build images
                advi = create_adv_images(imgs, sgrad, step)
                # passes forward, giving probabilities and loss
                loss, probs = pass_forward_sup(advi,labs)
                # Loss is sum of loss of all different samples: should be corrected for batch si
                loss = loss / n
                # found labels with max probability
                found_labels = np.argmax(probs,axis=1)
                #walk over the 128 images seperately
                for x in range(n):
                    #save iamges and probablities
                    images[image_idx,:] = advi[x]
                    all_probs[image_idx,:] = probs[x]

                    #to csv
                    data.append([image_idx,s+x,loss, step, labs[x], found_labels[x],probs[x,foun
                    image_idx +=1
        #save to csv
        df = pd.DataFrame(data=data, columns=COLS)
        df.to_csv('supervised_images.csv', encoding='utf8')

        #save to npz files
        np.savez('supervised_advs.npz',images=images,org_probs=org_probs,all_probs=all_probs)
```

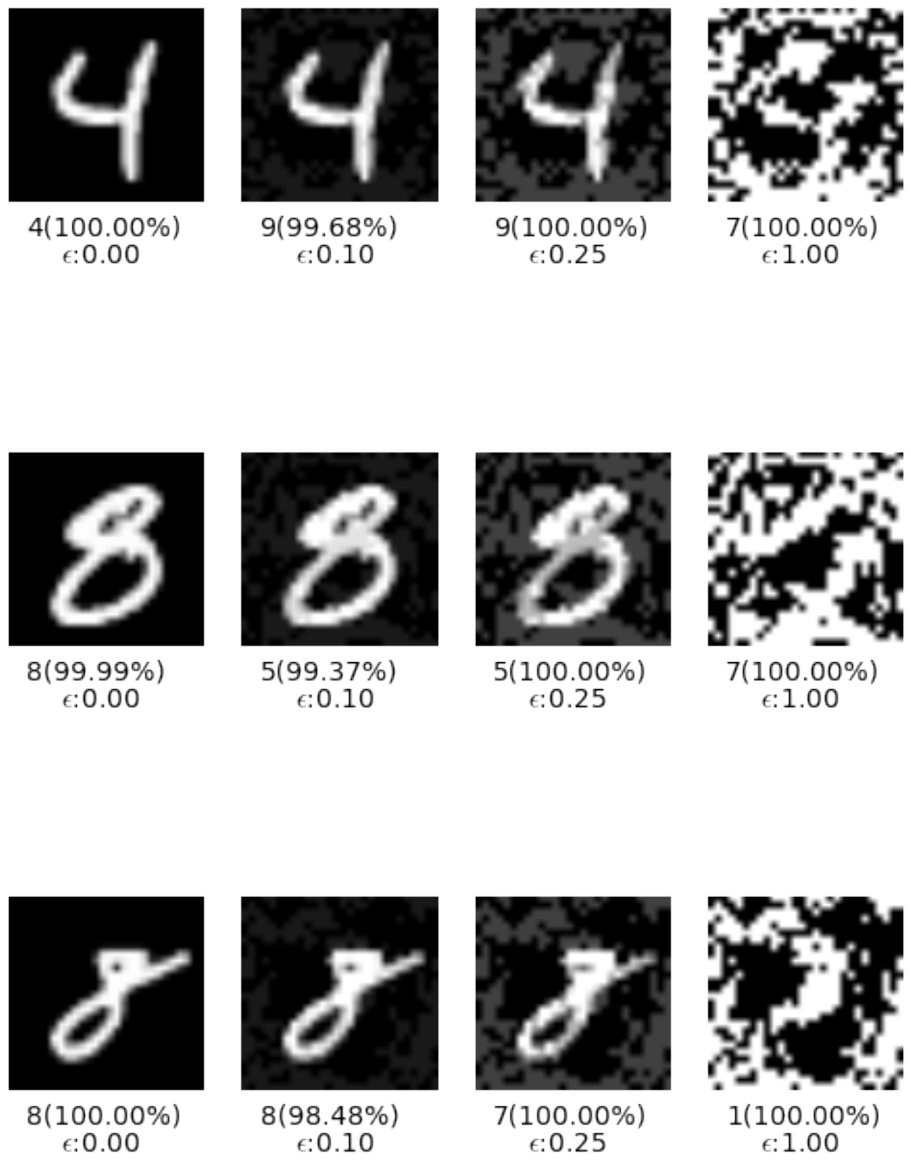
1.2.2 Showing adversarial images

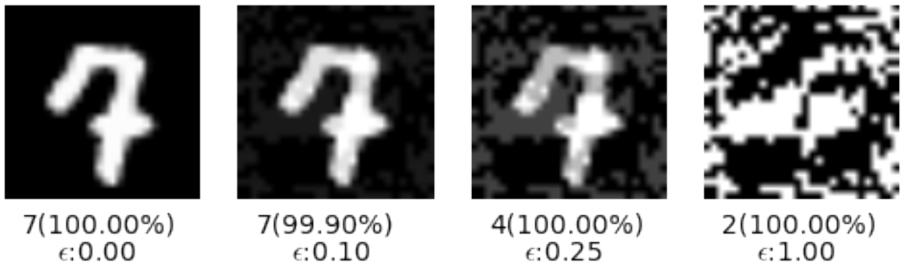
I plot all versions of the two images that are wrongly classified with $\epsilon \geq 0.1$ and two that are wrongly classified with $\epsilon \geq 0.25$.

```
In [8]: def _plot_for(org_ids):
            for org_id in org_ids:
                selected_img = df[df.org_index == org_id].sort_values('eps')
                imgs = np.take(images,selected_img.index.tolist(),axis=0)
                conf = selected_img.confidence.tolist()
                found_labels = selected_img.found_label.tolist()
                plot_images(imgs, conf, found_labels)

        incorrect = df[df.found_label != df.correct] #select that are classified correclty

        # incorrectly classified anyway
        incorrect_eps_0 = incorrect[incorrect.eps == 0.0]

        # Incorrectly classified with eps=0.1
        incorrect_eps_1 = incorrect[(incorrect.eps == 0.1) & (incorrect.org_index.isin(incorrect
        idx = incorrect_eps_1.sample(2).org_index.tolist()
        print "Wrongly classified eps >= 0.1", idx
        _plot_for(idx)
```

```
# Incorrecly classfied with eps=0.25
incorrect_eps_25 = incorrect[(incorrect.eps == 0.25)   & (incorrect.org_index.isin(incor
idx = incorrect_eps_25.sample(2).org_index.tolist()
print "Wrongly classified eps >= 0.25", idx
_plot_for(idx)
```

```
Wrongly classified eps >= 0.1 [9344, 7597]
Wrongly classified eps >= 0.25 [9780, 5866]
```





```
In [9]: stats = statistics_from_df(df)
        statistics_to_latex_table(stats)
```

```
\begin{tabular}{|l|l|l|}\hline
\textbf{type} & \textbf{accuracy} & \textbf{avg. confidence}\\\hline
$\epsilon = 0.00$ & 0.981 & 0.994\\\hline
$\epsilon = 0.10$ & 0.537 & 0.956\\\hline
$\epsilon = 0.25$ & 0.306 & 0.965\\\hline
$\epsilon = 1.00$ & 0.142 & 0.988\\\hline
\end{tabular}
```

1.3 GAN

To keep this notebook readable we import the scripts directly from python. Please see training GAN for more details about the GAN code used. We will doe this experiment twice. Once with a 250 epoch trained network once with discriminator trained for a 1000 epochs. In order to reproduce you would have to change lines below

```
In [10]: repo_root = os.path.join(os.path.split(os.getcwd())[0],'improved-gan')
         mnist_root = os.path.join(repo_root,'train_mnist')

         # Overwrite system arguments such that argparse in train_mnist/args.py works correctly
         #1000 epochs
         #sys.argv = ['main','-g',str(GPU_DEVICE),'-m',os.path.join(mnist_root,'mnist'),'-p',os.
         #250 epoch
         sys.argv = ['main','-g',str(GPU_DEVICE),'-m',os.path.join(mnist_root,'mnist_250'),'-p',

In [11]: # add the imported repository to the path both the general code and the train_mnist spe
         sys.path.append(repo_root)
         sys.path.append(os.path.join(repo_root,'train_mnist'))

         from gan import *
         from params import *
         # This will load the gan and the parameters learned during training
         from model import gan
         import mnist_tools
```

```
/Users/stijnvoss/Documents/UNI/CCN/git/end-assignment
loading /Users/stijnvoss/Documents/UNI/CCN/git/end-assignment/improved-gan/train_mnist/mnist_250
loading /Users/stijnvoss/Documents/UNI/CCN/git/end-assignment/improved-gan/train_mnist/mnist_250
loading /Users/stijnvoss/Documents/UNI/CCN/git/end-assignment/improved-gan/train_mnist/mnist_250
loading /Users/stijnvoss/Documents/UNI/CCN/git/end-assignment/improved-gan/train_mnist/mnist_250
```

1.4 Load adversarial images

```
In [12]: data = np.load('supervised_advs.npz')
         images = data['images']
         supervised_df = pd.read_csv('supervised_images.csv', encoding='utf8')
```

```
In [13]: def pass_forward_desc(ims, target_classes):
             """ Will pass a batch of images trough the descriminator, will calculate loss and p
             """
             # converting to chainer variable makes sure we can obtain gradient
             ims = gan.to_variable(ims)

             #Now pass forward trough the network and get probs, we do not use softmax since cha
             #to provide a cross entropy softmax loss function
             probs, activations = gan.discriminate(ims,test=True,apply_softmax=False)
             # our cross entropy function that will calculate the loss also need chainer var
             labels = gan.to_variable(target_classes)

             # Here we calculate the loss
             loss = F.softmax_cross_entropy(probs, labels)

             return gan.to_numpy(loss),gan.to_numpy(F.softmax(probs))
```

We will walk over the dataframe created at the adversarial image generation step. We walk over the images 1-by-1 because it's faster to implement

```
In [14]: def calculate_results_gan(title):
             """ Calculates loss and probabiliteis of generative adversarial network
             """
             data = []

             gan_probs = np.zeros((len(images),10), dtype=np.float32)
             counter = 0
             b = tqdm(total=len(images))
             for t in supervised_df.itertuples():
                 #we have to multiple by 255. since our network is expecting in range 0-225.
                 img =  255. * np.take(images, t.index,axis=0).reshape(1,784).astype(np.float32)
                 # Forward pass
                 loss, probs = pass_forward_desc(img,np.array([t.correct],dtype=np.int32))

                 # save
                 probs = probs[0]
```

```
                gan_probs[t.index,:] = probs

                # found label
                found_label = np.argmax(probs)

                # add row for csv
                data.append([t.index, t.org_index, float(loss), t.eps, t.correct, found_label,
                b.update()

            df = pd.DataFrame(data=data, columns=COLS)
            df.to_csv('gan_results_%s.csv' % title,encoding='utf8')
            statistics_to_latex_table(statistics_from_df(df))
```

```
In [15]: calculate_results_gan('250_epochs')
```

```
100%|| 39996/40000 [01:30<00:00, 450.84it/s]

\begin{tabular}{|l|l|l|}\hline
\textbf{type} & \textbf{accuracy} & \textbf{avg. confidence}\\\hline
$\epsilon = 0.00$ & 0.768 & 1.000\\\hline
$\epsilon = 0.10$ & 0.698 & 1.000\\\hline
$\epsilon = 0.25$ & 0.525 & 1.000\\\hline
$\epsilon = 1.00$ & 0.115 & 1.000\\\hline
\end{tabular}


100%|| 40000/40000 [01:50<00:00, 450.84it/s]
```

```
In [ ]:
```