Tutorial: Generative Adversarial Network

Background

In this tutorial, I will introduce, implement and investigate a Deep Convolutional Generative Adversarial Network, or DCGAN, proposed by Ian Goodfellow in 2014. (https://arxiv.org/abs/1406.2661)

Data: Binarized MNIST

We will use the MNIST dataset of 28×28 pixel images where each pixel is **either on or off**.

The binary variable $x_i \in \{0, 1\}$ indicates whether the i-th pixel is off or on.

Loading the Dataset

The following code loads the horizontally-placed MNIST dataset from the package MLDatasets and rotates the images to be vertical. Then data is binarized into {0, 1} labels and put into batches of 200.

```
using Flux
using BSON: @save
using MLDatasets: MNIST
using Plots
using Random
```

```
begin
    # load the original greyscale digits due to incompatibility with Flux
    train_digits, train_labels = MNIST.traindata()

    # Transform the horizontally-displayed data to vertically-displayed
    train_digits = permutedims(train_digits, (2, 1, 3))
    train_digits = train_digits[end:-1:1, :, :]

    greyscale_MNIST = reshape(train_digits, (28 * 28, size(train_labels)[1]))
```

```
    # binarize digits
    binarized_MNIST = greyscale_MNIST .> 0.5

    # partition the data into batches of size BS
    BS = 200

    # batch the data into minibatches of size BS
    batches = Flux.Data.DataLoader(binarized_MNIST, batchsize=BS)
end
```

GAN Overview

A DCGAN model consists of neural networks. The most common structure contains two conventional neural networks, which are refered to as a generator G and a discriminator D. We will refer to training data as X. (We will refer to the model as GAN onwards).

Generator

As the name suggests, the generator's main function is to generate data that resemble real data. We input random noise based on the latent representation of the data into G, where it outputs data that resembles entries in X.

Discriminator

As the name suggests, the discriminator's main function is to label whether the data is real (entries in X) or fake (generated by G). We will assign a label of 1 to real data, and label of 0 to fake data.

Training Process

We update G and D alternatively by holding the other parameter constant.

Firstly, we train the discriminator by inputing a batch of real data, and a batch of fake data. In this step, we penalizes the discriminator using cross entropy loss.

Secondly, we train the generator by penalizing its “incapacity” to “fool” the discriminator. Ideally, we want G to generate data that are indistinguishable from X.

Analogy

The training process is similar to attaining the Nash Equilibrium in a Game Theory problem. Ideally, at the point of convergence, the discriminator can only achieve an accuracy of 50% when given a batch of real data and fake data. At that point, we can say that the generator is powerful enough to “fool” the discriminator completely.

Model Parameter Definitions

Input Data X

Each element in the data $x \in X$ is a vector of 784 pixels.
Each pixel x_d is either on, $x_d = 1$ or off $x_d = 0$.

Each element corresponds to a handwritten digit $\{0, \ldots, 9\}$. These labels are not used.

Latent Variable z

We will introduce a latent variable $z \in \mathbb{R}^{28}$ to represent the input to the generator G. A higher dimension should lead to a more powerful model.

- **Assumption**: We assume that the digit’s latent representation is a multivariate standard normal distribution. $p(z) = \mathcal{N}(z \mid \mathbf{0}, \mathbf{1})$

Generator G

The `Generator` is a neural network with fully-connected architecture. The `Generator` accepts the latent variable z as input, a single hidden layer with 500 units and `tanh` nonlinearity, and a fully-connected output layer to the MNIST data space with `sigmoid` nonlinearity. The reason to use `sigmoid` in the output layer is to make sure that the output values can be binarized to resemble MNIST data.

Discriminator D

The `Discriminator` is a neural network with fully-connected architecture. The `Discriminator` accepts $x \in D$ as input. The first hidden layer has 500 units and `tanh` nonlinearity. The output layer has 2 units that undergo `softmax` transformation to make sure the output resembles probability. The second unit is dropped for convenience.

Implementation Dense, Chain function in Flux package are used to construct G and D

```
Dz, Dh, Ddata = 28, 500, 28^2

function softmax(x)
    return (exp.(x)./sum(exp.(x), dims=1))[1, :]
end

function binarize(x)
    return x .> 0.5
end

begin
    layer1 = Dense(Ddata, Dh, tanh)
    layer2 = Dense(Dh, 2, tanh)
    layer3 = softmax
    discriminator = Chain(layer1, layer2, layer3)
end

begin
    layer4 = Dense(Dz, Dh, tanh)
    layer5 = Dense(Dh, Ddata, sigmoid)
    generator = Chain(layer4, layer5)
end
```

Model Loss Functions

The output label is binary, so we use cross entropy loss with formula.
$-\sum_{x_i \in X} t_i log p(x_i) + (1 - t_i) log(1 - p(x_i))$

Discriminator D

The entries from X are assigned a label of 1. The entires generated by G, let's denote as G* are assigned by a label of 0. Therefore we can calculate the losses from X, and from G separately, and sum them up to attain Loss_D.

Loss_X $= -\sum_{x_i \in X} logp(x_i)$.

Loss_G* $= -\sum_{g_i \in G*} log(1 - p(G(g_i)))$.

Generator G

The entries from G are all assigned a label of 0. However, the purpose of the generator is to "trick" the discriminator into believing that the data is real. Therefore, we penalize when the output probability is further from 1.

Loss_G $= -\sum_{g_i \in G*} log(p(G(g_i))$

```
function discriminator_loss(x, z)
    real_loss = -sum(log.(discriminator(x)))
    fake_loss = -sum(log.(1 .- discriminator(generator(z))))
    return real_loss/size(x)[2] + fake_loss/size(z)[2]
end
function generator_loss(z)
    fake_loss = -sum(log.(discriminator(generator(z))))
    return fake_loss/size(z)[2]
end
```

Model Optimization

In the training process, we update the parameters D, G alternatively using backward propagation in each batch.

In order to better understand the training progress, we keep track of the loss per epoch and use an animation to show the gradual improvement over G's capabilities.

```
function train!(dis, gen, data; nepochs=2)
    params_dis = Flux.params(dis)
    params_gen = Flux.params(gen)
```

```
d_loss_lst = []
g_loss_lst = []

opt = ADAM()

num_samples = 15
anim_arr = reshape([], 0, 28 * num_samples)

for epoch in 1:nepochs
    num_batch = 0
    @info "Epoch: $epoch"
    # Include Training Progress to Visualize the Generator Capabilities
    epoch_arr = reshape([], 28, 0)
    for sample in 1:num_samples
        img_arr = reshape(generator(randn(Dz)), 28, 28)
        epoch_arr = cat(epoch_arr, img_arr, dims=2)
    end

    anim_arr = cat(epoch_arr, anim_arr, dims=1)

    for real_batch in data

        fake_batch = randn(Dz, size(real_batch)[2])

        # compute gradient wrt discriminator_loss
        grad = Flux.gradient(params_dis) do
        return discriminator_loss(real_batch, fake_batch)
        end
        # update discriminator
        Flux.update!(opt, params_dis, grad)

        # compute gradient wrt generator_loss
        grad = Flux.gradient(params_gen) do
        return generator_loss(fake_batch)
        end
        # update generator
        Flux.update!(opt, params_gen, grad)

        d = discriminator_loss(real_batch, fake_batch)
        g = generator_loss(fake_batch)
        # Keep track of losses and batches
        push!(d_loss_lst, d)
        push!(g_loss_lst, g)
```

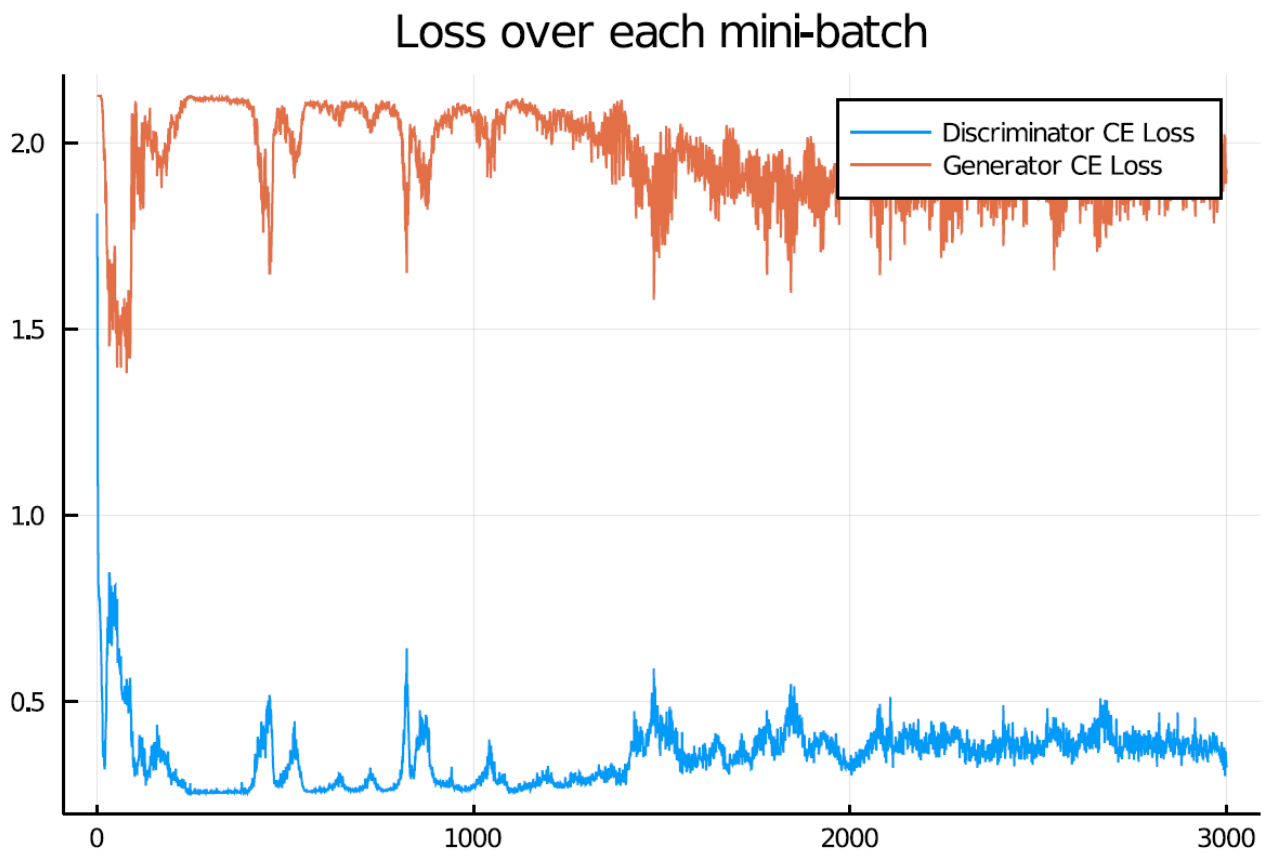
```
            @info "Discriminator: $d, Generator: $g"
            num_batch += 1
        end

    end
    return d_loss_lst, g_loss_lst, anim_arr
end


d_l, g_l, anim_l = train!(discriminator, generator, batches; nepochs=10)

@save "discriminator.bson" discriminator
@save "generator.bson" generator
```

Analysis

Training Loss

Contrary to conventional networks, the loss over each mini-batch does not decrease over time. It is hard to tell the progress of training from these loss plots. This is due to the alternative convergence that happens during training as GAN attempts to find an equilibrium between discriminator and generator.

```
begin
    plot(title="Loss over each mini-batch")
    plot!(d_l, label="Discriminator CE Loss")
    plot!(g_l, label="Generator CE Loss")
end
```

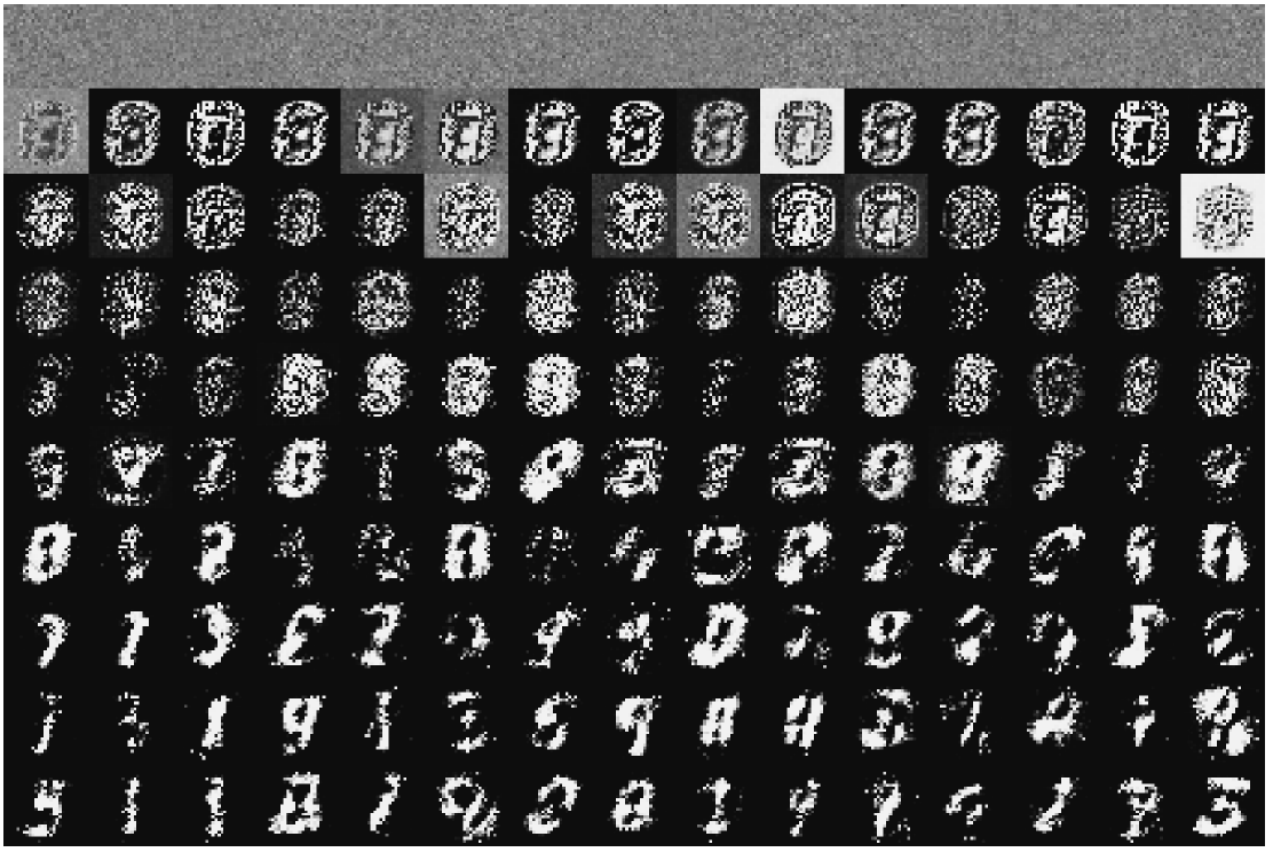


Generator Improvement over each Epoch

In each epoch, we sample 15 random latent inputs and visualize the generated image. Each horizontal line represents 1 epoch and we can clearly see G's training progress over time as the image it generates resemble closer to MNIST digits.

In the first iteration, the images resemble pure randomness. From the 2nd - 5th iteration, we can see that the generator starts to identify patterns in the middle of each image, although the shape is still not recognizable as digits. In later iterations, we can identify images that somewhat resemble the digit labels in MNIST.

```
heatmap(anim_l, color=:grays, aspect_ratio=1, grid=false, ticks=false, axis=fa
```



Discriminator Accuracy

Like most neural network structures, it is hard to quantify the exact point of convergence. In theory, the generator is well trained when the discriminator cannot distinguish between real and fake images. In other words, given equal amount of real and fake data, the discriminator accuracy should be close to 50%.

We will create a mini-batch of 400 images with 200 real and 200 fake, and loop over the entire dataset.

We see that the discriminator is able to acheive high accuracy, which indicates that the generator is not powerful enough and can be finetuned.

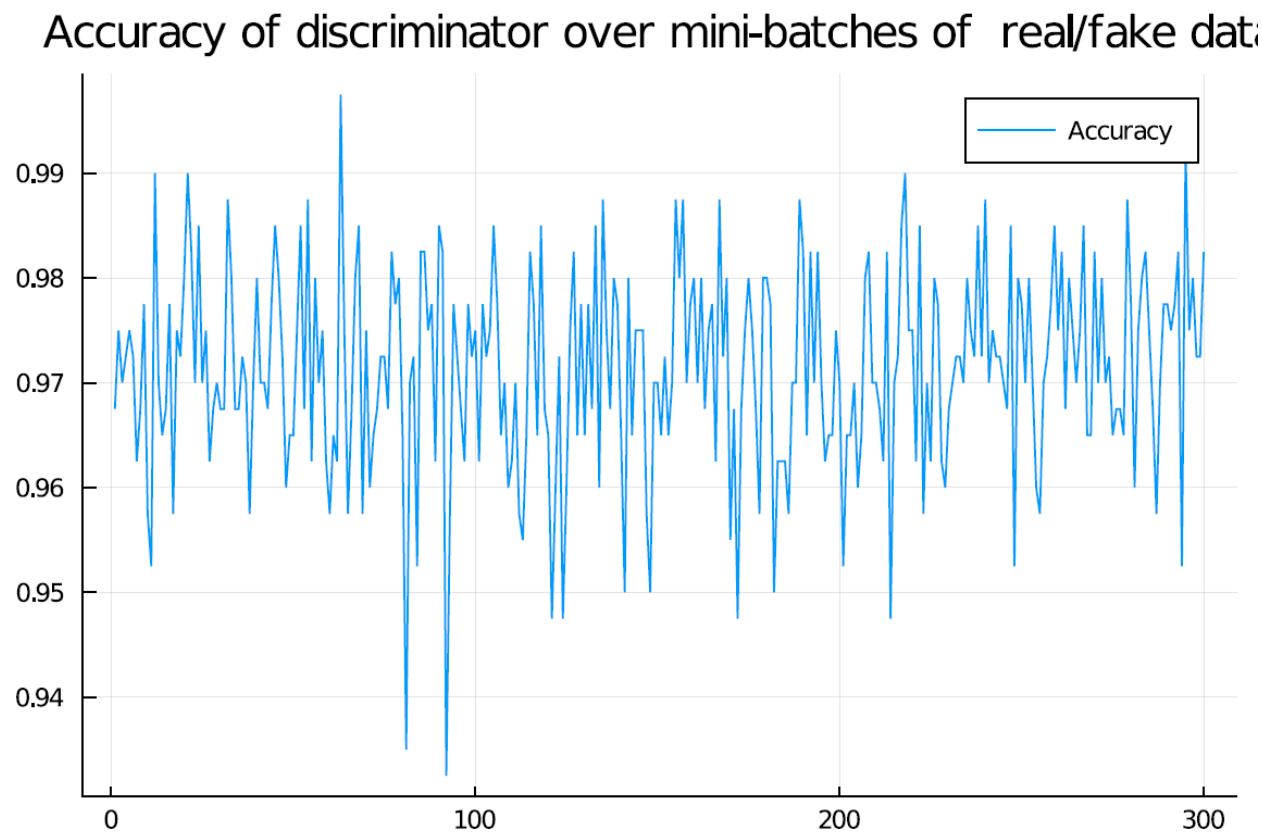
Discriminator Accuracy

Like most neural network structures, it is hard to quantify the exact point of convergence. In theory, the generator is well trained when the discriminator cannot distinguish between real and fake images. In other words, given equal amount of real and fake data, the discriminator accuracy should be close to 50%.

We will create a mini-batch of 400 images with 200 real and 200 fake, and loop over the entire dataset.

We see that the discriminator is able to acheive high accuracy, which indicates that the generator is not powerful enough and can be finetuned.

```
function discriminator_accuracy(data)
    accuracy_lst = []
    for batch in data
        fake_pred = discriminator(generator(randn(Dz, size(batch)[2])) .> 0.5)
        real_pred = discriminator(batch)

        total_correct = sum(fake_pred .< 0.5) + sum(real_pred .>= 0.5)
        push!(accuracy_lst, total_correct/2/size(batch)[2])
    end
    return accuracy_lst
end

begin
    plot()
    plot!(discriminator_accuracy(batches), label="Accuracy", title="Accuracy c
end
```



Challenge: Spot the Fake

We randomly generate a 6 x 6 grid where about 1/3 of the images are fake. Can you spot the fake ones easily from your naked eyes?

```
let
    dim1 = 6
    dim2 = 6

    verti_arr = reshape([], 0, 28 * dim2)
    for i in 1:dim1
        horiz_arr = reshape([], 28, 0)
        for j in 1:dim2

            choice = rand() .> 2/3

            if choice
                img_arr = reshape(generator(randn(Dz)), 28,28) .> 0.5
            else
                img_arr = reshape(binarized_MNIST[:, rand(1:size(binarized_MNI
            end
            horiz_arr = cat(horiz_arr, img_arr, dims=2)
        end
        verti_arr = cat(verti_arr, horiz_arr, dims=1)
    end
    heatmap(verti_arr, color=:grays, aspect_ratio=1, grid=false, ticks=false,
end
```

Future Improvements

GANs are difficult to train as the alternative gradient descent is inherently unstable. The improvement of the discriminator comes at the expense of the generator and vice versa. In this section, I will propose several ways to improve the model's stablility and robustness.

Higher Dimension Latent Input

Modeling the input with a 28 dimension multivariate normal distribution may too be idealistic. In general, the higher the input dimension, the more closer the generated data will resemble real data.

Dropout Layer

To avoid overfitting, adding a dropout layer is a common strategy used in neural networks. We can incorporate this in both neural networks so that the model does not depend its prediction on very few features.

Label Smoothing

Sometimes, the model gets too confident in its prediction so we can penalize when the discriminator outputs a value outside the range [0.1, 0.9]. The overfitting may occur if the discriminator uses very few features to identify real/fakeness and the generator picks it up and starts producing images that exploit these features only.

Loss Function

There are many loss functions that can be substituted for cross entropy. We can add regularization parameters as well.

References

Deep Convolutional Generative Adversarial Network : TensorFlow Core. (n.d.). Retrieved from https://www.tensorflow.org/tutorials/generative/dcgan

Goodfellow, I. (2017, April 03). NIPS 2016 Tutorial: Generative Adversarial Networks. Retrieved from https://arxiv.org/abs/1701.00160

Goodfellow, I. J., Pouget-Abadie, J., Mirza, M., Xu, B., Warde-Farley, D., Ozair, S., . . . Bengio, Y. (2014, June 10). Generative Adversarial Networks. Retrieved from https://arxiv.org/abs/1406.2661

A Beginner's Guide to Generative Adversarial Networks (GANs). (n.d.). Retrieved from https://wiki.pathmind.com/generative-adversarial-network-gan#:~:text=Generative adversarial networks (GANs) are,video generation and voice generation.

ProbabilisticLearning. (n.d.). ProbabilisticLearning/Assignment-3. Retrieved from https://github.com/ProbabilisticLearning/Assignment-3