Generating MNIST Digits using GANs

By:ANIRUDH S(EE18B073)

Generative Adversarial Networks(GANs) are used for training generative models such as deep convolutional layers.

Generative Adversarial Networks (GANs) can be broken down into three parts:

- **Generative:** To learn a generative model, which describes how data is generated in terms of a probabilistic model.
- **Adversarial:** The training of a model is done in an adversarial setting.
- **Networks:** Use deep neural networks for training purpose.

Any basic GAN consists of two parts: **Discriminator and Generator.**The Discriminator is used classify and differentiate between real and fake images. I have used the labels of 0 for fake images and 1 for real images and a generator model that uses inverse convolutional layers to transform an input to a full two-dimensional image of pixel values.

Difference between Simple GAN and DCGAN:

The basic difference between Simple GAN and DCGan is that in the **generator** part. The simple GAN uses a **fully connected layer** in the generator whereas the DCGan uses **Transposed Convolution** to do the Upsampling. Downsampling is done using **Convolutional Strides** and MAX Pooling is not used in case of DCGan.

For HyperParameters and activation functions:

I referred to the article by **JONATHAN HUI in Medium** which gave proper insights into what parameters to use for a better model. Article is Titled 'GAN-DCGAN(Deep Convolutional generative adversarial networks)

In that article it clearly mentions to use:

a)Leaky Relu instead of RELU with a slope of 0.2

b)Replace all MaxPooling with Conv. Strides

c)Use Adam with lr=0.0002 instead of normal 0.001 which was found to be very high.

d)Also the momentum term (beta) which is generally used as 0.9 is used as 0.5.This is because the value of 0.9 resulted in training oscillations and instability hence was reduced to 0.5 which provide stability.

My Discriminator model:

1)Layer 1: Conv2D layer with 64 filters and kernel size of (3,3) and strides of (2,2).Activation used is Leaky RELU with slope of 0.2 is used and Kernel weight initializer for symmetry breaking along with DropOut of 0.4(Reduces overfitting)

Generating MNIST Digits using GANs

By:ANIRUDH S(EE18B073)

2)Layer 2: Conv 2D layer with 128 filters and kernel size of (3,3) and strides of (2,2).Activation used is Leaky RELU with slope of 0.2 is used and Kernel weight initializer for symmetry breaking along with DropOut of 0.4(Reduces overfitting)

3)Flatten: This converts the 2D input into 1D input so that this can be fed into a dense layer as input.

4)Dense: This takes in input from the Flattened layer and uses sigmoid activation function. The weight initializer of 'Glorot_Uniform' is more appropriate for Tanh and sigmoid hence that is used instetad of 'he_uniform'.

Q)What is binary_crossentropy?

Loss function is used is **binary cross_entropy**. The definition of entropy comes from random variables. It is defined as the **expectation of your information content which is log(p).**so the normal cross entropy is (-1***sigma(p*logp)) .** In case of a true variable and predicted variable the cross entropy formula will give us how similar are the actual and predicted values. This is given by the formula (-1***sigma(p*log(q))**)

My Generator model:

a)INPUT: Latent_Dim is taken as input. It is generally taken to be 100. "**latent_dim**" is the number of nodes used as input of the generator. This gives a 100 element vector of Gaussian random numbers.

b)OUTPUT: Two-dimensional square grayscale image of 28×28 pixels with pixel values in range[0,1] (due to preprocessing step of dividing by 255)

1)Layer1: Dense layer as the first hidden layer that has enough nodes to represent a low-resolution version of the output image. I take a dense layer with 128*14*14 units . The activations from these nodes can then be reshaped into something image-like to pass into a convolutional layer, such as 128 different 14×14 feature maps.

2)Layer 2: We then use Conv2DTranspose layer. This is just the combination of UpSampling2D and Conv2D.This is used to increase the resolution of the image and increases its area. The layer increases the width and height dimensions by two(strides is used as two).So we end up getting a 28*28 layer at the end of these layers.

3)Layer3: The output layer of the model is a *Conv2D* with one filter and a kernel size of 7×7 and 'same' padding , designed to create a single feature map and preserve its dimensions at 28×28 pixels. A sigmoid activation is used to ensure output values are in the desired range of [0,1].

Generating_fake_images:

Generating MNIST Digits using GANs

By:ANIRUDH S(EE18B073)

This generator function can now be used to generate fake images.We need to first get the latent_dim(random points in the 100D space). For which I had used the function called generate_latent_points. This generated latent_points is then used to generate_fake_images.

GAN MODEL:

Then I have a model named GAN which is combination of discriminator and generator. This is basically generator(discriminator(x)). When training the generator via the GAN model, we want the discriminator to think that the samples output by the generator are real, not fake. Therefore, when the generator is trained as part of the GAN model, we will mark the generated samples as real .

The discriminator will then classify the generated samples as not real or a low probability of being real (0.3 or 0.5). The backprop process used to update the model weights will see this as a large error and will update the model weights (i.e. only the weights in the generator) to correct for this error, in turn making the generator better at generating good fake samples.

Generating real Samples:

This functions load_real_images and generate_real_images can now be used to generate real images.We load the data directly from keras.dataset.mnist's load data function .Then we randomly generate the numbers to get the required number of real images from the loaded dataset . This is then used to generate_real_images.

TRAINING THE MODEL:

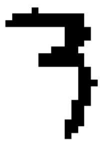
The discriminator model is updated once per batch by combining one half a batch of fake and real examples into a single batch via the vstack function. I took batch_size to be 256, so there will be 128 real and fake images each in a particular batch .I have labelled te fake images as zero and real images as one. This is then trained for 100 epochs. We find the individual loss for real and fake images using a another function called performance. That

Q)Why is that x1-x2 and x1+x2 of different inputs gives a superimposed kind of result?

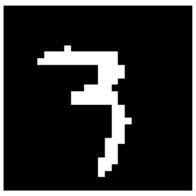
This is actually what is expected. I used plt.imshow to plot the images. That function plots the pixel values to get the image where 1 corresponds to white and 0 corresponds to black. But actually there is no restriction on pixel value. In most cases we use negative pixel values to leave out the particular function. It is also highly logical to think when I plot X and –X together everything should be black(AS they add up to zero).Each pixel in above case becomes zero individually. Similarly when we take two different inputs and do the same it does follow the pixel addition and subtraction to get the actual results.

Generating MNIST Digits using GANs

By:ANIRUDH S(EE18B073)



This is for -X

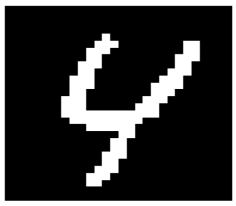


This is for X

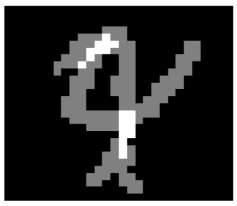
This is for X+(-1*X)

The plots clearly shows what happens if we add a linear combination of two different images .In case of two different images the locations where there is no exact cancellation of white and black results in grey colour. Common white remains white and common black remains black.

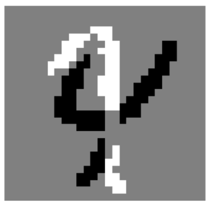
This is plain 1



This is plain 4



This is 1+4



This is 1-4

Q)What will happen if we interchange the labels for real and fake images?

In this model , the discriminator predicts 1 for real image and 0 for a generated image. When we modify the above labelling method and end up giving the opposite labels we still notice that model is trained on equal number of real and fake images in every batch. The data used for training is hence not skewed towards any class. The possible help by changing the labels is that the backprop algorithm becomes more robust. The weight updating is larger. Also I noticed for a fact that the accuracy of fake and real images were about 51% and 70% in the reversed labelled case whereas in the original case the discriminator accuracy was much lesser.