Anirudh Manjunath

50289832

Image-to-Image Translation with Conditional Adversarial Networks

Generative modeling is the use of artificial intelligence (AI), statistics and probability in applications to produce a representation or abstraction of observed phenomena or target variables that can be calculated from observations.

Generative modeling is used in unsupervised machine learning as a means to describe phenomena in data, enabling computers to understand the real world. An example of a generative model might be one that is trained on collections of images from the real world in order to generate similar images.

Generative modeling contrasts with discriminative modeling, which identifies existing data and can be used to classify data. Generative modeling produces something; discriminative modeling recognizes tags and sorts data. While discriminative models care about the relation between y and x, generative models care about "how you get x." They allow you to capture p(x|y), the probability of x given y, or the probability of features given a label or category.

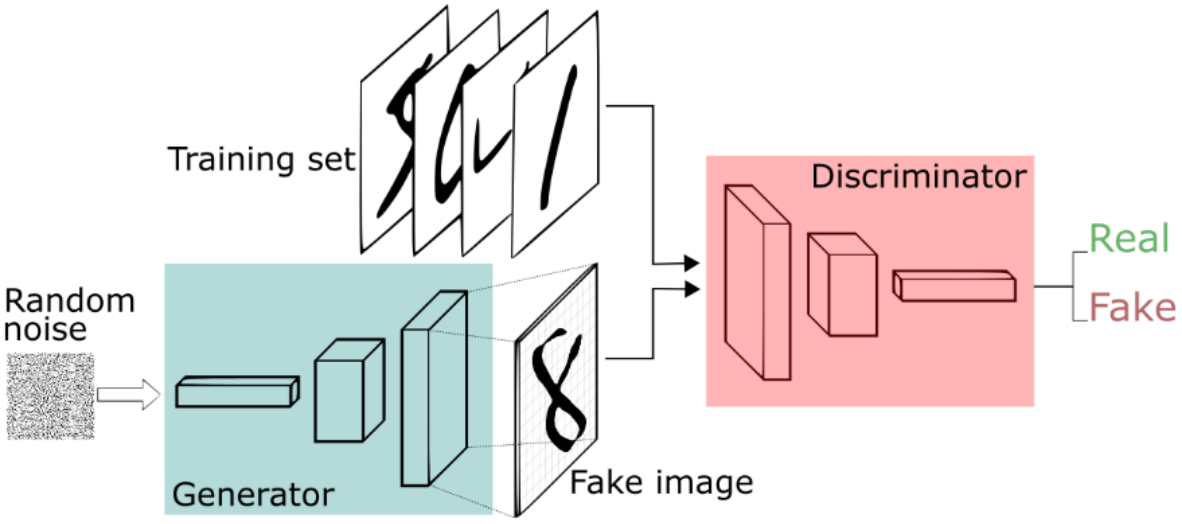
In GANs, one neural network, called the generator, generates new data instances, while the other, the discriminator, evaluates them for authenticity; i.e. the discriminator decides whether each instance of data that it reviews belongs to the actual training dataset or not.

We can understand the working of GAN with an example. We will take generating images of numbers when we input a number. We can use MNIST data for this.

The generator takes in random numbers and returns an image. This generated image is fed into the discriminator alongside a stream of images taken from the actual, ground-truth dataset. The discriminator takes in both real and fake images and returns probabilities, a number between 0 and 1, with 1 representing a prediction of authenticity and 0 representing fake.

The discriminator is in a feedback loop with the ground truth of the images, which we know. The generator is in a feedback loop with the discriminator.

The generator is an inverse convolutional network, in a sense: While a standard convolutional classifier takes an image and downsamples it to produce a probability, the generator takes a vector of random noise and upsamples it to an image. The first throws away data through downsampling techniques like maxpooling, and the second generates new data.



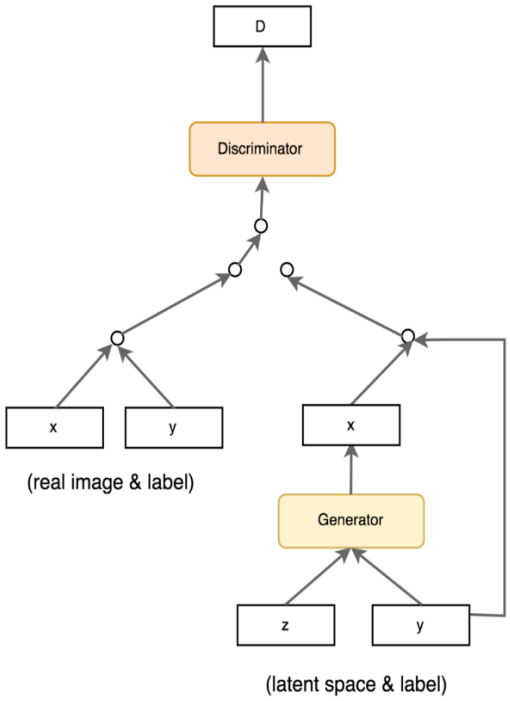
Both nets are trying to optimize a different and opposing objective function, or loss function, in a zero-zum game. This is essentially an actor-critic model. As the discriminator changes its behavior, so does the generator, and vice versa. Their losses push against each other.

In GAN, training models are non-trivial. It can take extra help from the labels to make the model performing better. In CGAN (Conditional GAN), labels act as an extension to the latent space z to generate and discriminate images better. In GAN, there is no control over modes of the data to be generated. The conditional GAN changes that by adding the label y as an additional parameter to the generator and hopes that the corresponding images are generated. We also add the labels to the discriminator input to distinguish real images better.

In MNIST, we sample the label y from a uniform distribution to generate a number from 0 to 9. We encode this value into a 1-hot vector. For example, the value 3 will be encoded as (0, 0, 0, 1, 0, 0, 0, 0, 0, 0). We feed the vector and the noise z to the generator to create an image that resembles “3”. For the discriminator, we add the supposed label as a one-hot vector to its input. The cost function for CGAN is the same as GAN.

$$\min_G \max_D V(D,G) = \mathbb{E}_{\boldsymbol{x}\sim p_{\text{data}}(\boldsymbol{x})}[\log D(\boldsymbol{x}|\boldsymbol{y})] + \mathbb{E}_{\boldsymbol{z}\sim p_z(\boldsymbol{z})}[\log(1 - D(G(\boldsymbol{z}|\boldsymbol{y})))].$$

D(x|y) and G(z|y) demonstrates we are discriminating and generating an image given a label y. (It is the same as D(x, y) and G(z, y) in other diagrams.).



In this approach, we alternate between one gradient descent step on D, then one step on G. As suggested in the original GAN paper, rather than training G to minimize log(1 − D(x, G(x, z)), we instead train to maximize log D(x, G(x, z)). , z)), we instead train to maximize log D(x, G(x, z)). In addition, we divide the objective by 2 while optimizing D, which slows down the rate at which D learns relative to G. We use minibatch SGD and apply the Adam solver, with a learning rate of 0.0002, and momentum parameters β1 = 0.5, β2 = 0.999.

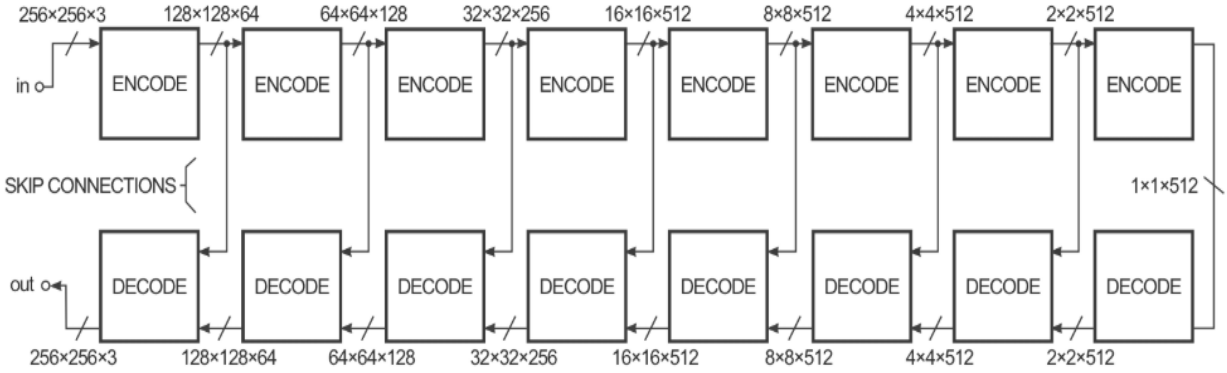
At inference time, we run the generator net in exactly the same manner as during the training phase. This differs from the usual protocol in that we apply dropout at test time, and we apply batch normalization.

OBJECTIVE

In this project, we apply cGANs to convert Black and White images to colour. I have used to pix2pix TensorFlow code and have obtained the results. The dataset used was MIT Places 365, which contains 10 different types of dataset like attic, field, office, etc. I have considered equal number of samples from each type of the dataset and trained the model. In total, Places contains more than 10 million images comprising 400+ unique scene categories. The dataset features 5000 to 30,000 training images per class, consistent with real-world frequencies of occurrence.

For training I have used 2000 images and trained the model for varying epochs.

In order to improve the performance of the image-to-image transform in the paper, the authors used a "U-Net" instead of an encoder-decoder. The skip connections give the network the option of bypassing the encoding/decoding part if it doesn't have a use for it.



To train the model, the pix2pix code can take inputs in two formats. The first one is combining the original and target side by side and training the model to use one image as input and the other as the target to adjust the weights. I have used the second method where the input is a colour image and a method converts it into black and white to train the model and uses the input as the target.

```
python pix2pix.py
--mode train
--output_dir photos_train
--max_epochs 200
--input_dir photos/train
--lab_colorization
```

To train this network, there are two steps: training the discriminator and training the generator.

To train the discriminator, first the generator generates an output image. The discriminator looks at the input/target pair and the input/output pair and produces its guess about how realistic they look. The weights of the discriminator are then adjusted based on the classification error of the input/output pair and the input/target pair. The generator's weights are then adjusted based on the output of the discriminator as well as the difference between the output and target image.

RESULTS

In the testing, we can observe that increasing the epochs doesn't necessarily improve the results. This is because of the wide variety of input data we are considering.

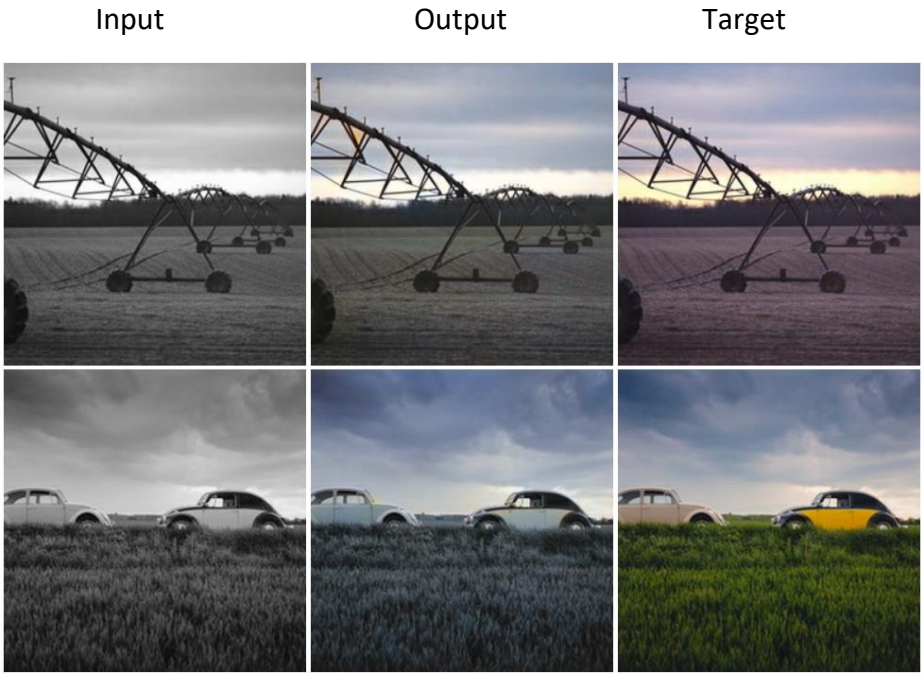
30 epochs

Input Output Target

100 epochs

But that is not the case for all the images. We can see that if the testing images contains more features from the training set, the model trained for 100 epochs performs better.

30 epochs



100 epochs

We can observe that the model cannot colorize all types of black and white images but only the types that is was being trained on. To obtain even higher accuracy, we have to train it with much bigger dataset containing wider types of images. For this, the training epochs must be much higher.

FUTURE WORKS

Noting the advantages of colorizing images, I experimented with converting old black and white movies to color. To train this, we should use color images from the 60's ad 70's because, much of the clothing, lighting, and architecture would be similar to the even older films which we can remaster.

```
for i in range(len(images)):
    filename=images[i]
    name.append(filename)
    img = cv2.imread(filename)
    height, width, layers = img.shape
    size = (width,height)

    #inserting the frames into an image array
    frame_array.append(img)
out = cv2.VideoWriter(pathOut,cv2.VideoWriter_fourcc(*'DIVX'), fps, size)
for i in range(len(frame_array)):
    # writing to a image array
    out.write(frame_array[i])
out.release()
```

As a sample, I tried the same model that I used for images to do this. The results were fairly good for many scenes.

Input

Output