SKETCH-2-PAINT

Project Report

Eklavya Mentorship Programme

At

SOCIETY OF ROBOTICS AND AUTOMATION,
VEERMATA JIJABAI TECHNOLOGICAL
INSTITUTE, MUMBAI

OCTOBER 2021

ACKNOWLEDGMENT

We were able to explore a new domain-Deep learning and successfully work on it only because of the expert guidance provided to us by our mentors **Saurabh Powar** and **Chaitravi Chalke**.
We are extremely thankful to our mentors for their patience,motivation, enthusiasm and immense knowledge which they shared with us throughout the duration of project.
We would also like to thank all other mentors of SRA VJTI for their constant support and motivation and for giving us the opportunity to be a part of Eklavya 2021.

Our Team:

Neel Shah
neelshah29042002@gmail.com
+919653380766

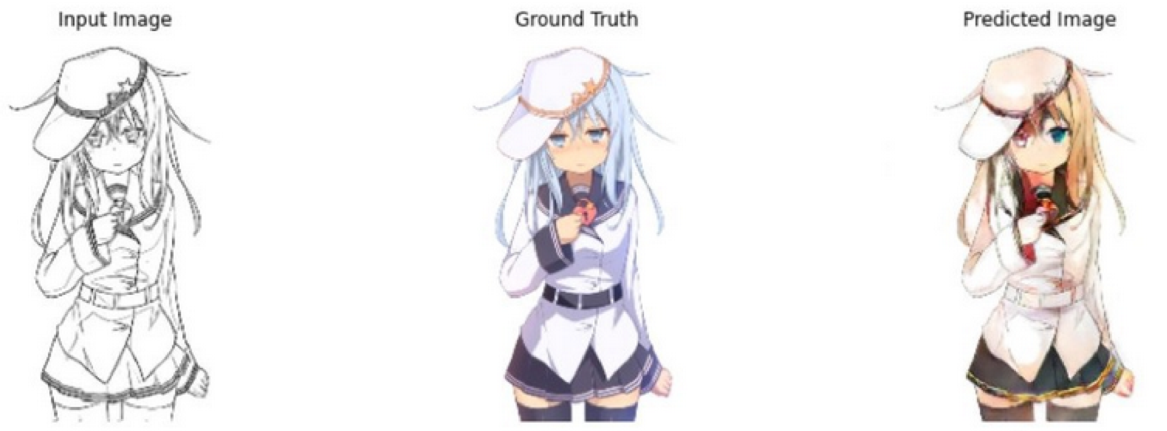
Kunal Agarwal
kunalagarwal1072002@gmail.com
+919820536577

TABLE OF CONTENTS

1. PROJECT OVERVIEW:

1.1 Description of Project:

GANs are a relatively recent invention in field of ML. The primary objective of GANs was to generate new samples from the given dataset. And since the invention of GANs, they have grown to accomplish this task with better results and added features. Sketch to Color Image generation is an image-to-image translation model using Conditional Generative Adversarial Networks. In this project we learnt to create a Conditional GAN to predict colorful images from the given black and white sketch inputs without knowing the actual ground truth.



1.2 Technology Used:

1. Tensorflow library-Python:

   TensorFlow is an end-to-end open source platform for machine learning. It has a comprehensive, flexible ecosystem of tools, libraries and community resources that lets researchers push the state-of-the-art in ML and developers easily build and deploy ML powered applications.

2. Keras-Python:

   Keras is an open-source neural-network library written in Python. It is capable of running on top of TensorFlow, Microsoft Cognitive Toolkit, R, Theano, or PlaidML. Designed to enable fast experimentation with deep neural networks, it focuses on being user-friendly, modular, and extensible

3. Matplotlib:

   Matplotlib is a comprehensive library for creating static, animated, and interactive visualizations in Python. It is a plotting library for the Python programming language and its numerical mathematics extension NumPy It provides an object-oriented API for embedding plots into applications using general-purpose GUI toolkits like Tkinter.

4. Google colab:

Colaboratory, or "Colab" for short, allows you to write and execute Python in your browser, with

1. Zero configuration required
2. Free access to GPUs
3. Easy sharing

If you are planning on using any cloud environments like Google Colab, you need to keep in mind that the training is going to take a lot of time as GANs are computationally quite heavy to run. Google Colab has an absolute timeout of 12 hours which means that the notebook kernel is reset so you'll need to consider some points like mounting the Google Drive and saving checkpoints after regular intervals so that you can continue training from where it left off before the timeout.

1.3 Brief Idea:

We are going to build a Conditional Generative Adversarial Network which accepts a 256x256 px black and white sketch image and predicts the colored version of the image without knowing the ground truth. The model will be trained on the Anime Sketch-Colorization Pair Dataset available on Kaggle which contains 14.2k pairs of Sketch-Color Anime Images.

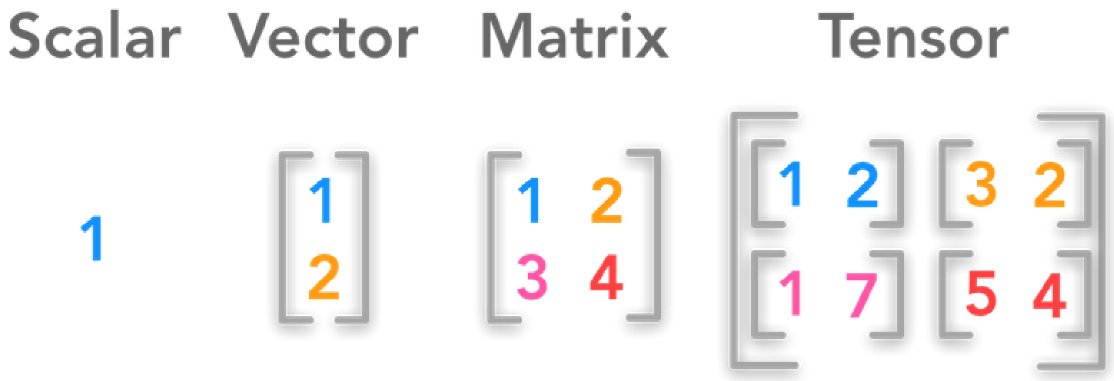
The model was trained on the Anime Sketch-Colorization Pair Dataset available on Kaggle which contains 14.2k pairs of Sketch-Color Anime Images.

2.THEORY

2.1 Linear Algebra:

Linear algebra is a form of continuous rather than discrete mathematics, many computer scientists have little experience with it. A good understanding of linear algebra is essential for understanding and working with many machine learning algorithms, especially deep learning algorithms.

The core data structures behind Deep-Learning are Scalars, Vectors, Matrices and Tensors. Programmatically, We solve all the basic linear algebra problems using these.



Scalars

Scalars are **single numbers** and are an example of a 0*th*-order tensor. The notation x ∈ ℝ states that x is a scalar belonging to a set of real-values numbers, ℝ.

There are different sets of numbers of interest in deep learning. ℕ represents the set of positive integers (1,2,3,…). ℤ designates the integers, which combine positive, negative and zero values. Q represents the set of rational numbers that may be expressed as a fraction of two integers.Few built-in scalar types are **int**, **float**, **complex**, **bytes**, **Unicode** in Python

Vectors

Vectors are ordered arrays of single numbers and are an example of 1st-order tensor. Vectors are fragments of objects known as vector spaces. A vector space can be considered of as the entire collection of all possible vectors of a particular length (or dimension). The three-dimensional real-valued vector space, denoted by ℝ^3 is often used to represent our real-world notion of three-dimensional space mathematically.

$$x = [x_1\ x_2\ x_3\ x_4\ \dots\ x_n]$$

To identify the necessary component of a vector explicitly, the i*th* scalar element of a vector is written as x[i].In deep learning vectors usually represent feature vectors, with their original components defining how relevant a particular feature is. Such elements could include the related importance of the intensity of a set of pixels in a two-dimensional image.

Matrices

Matrices are rectangular arrays consisting of numbers and are an example of 2*nd*-order tensors. If m and n are positive integers, that is $m, n \in \mathbb{N}$ then the m×n matrix contains m*n numbers, with m rows and n columns.The full m×n matrix can be written as:

$$A = \begin{bmatrix} a_{11} & a_{12} & a_{13} & \cdots & a_{1n} \\ a_{21} & a_{22} & a_{23} & \cdots & a_{2n} \\ a_{31} & a_{32} & a_{33} & \cdots & a_{3n} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ a_{m1} & a_{m2} & a_{m3} & \cdots & a_{mn} \end{bmatrix}$$

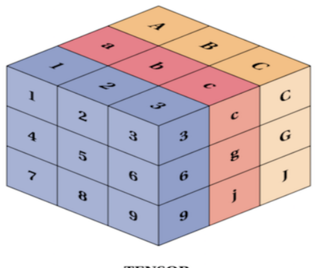
It is often useful to abbreviate the full matrix component display into the following expression:

$$A=[a_{ij}]_{m\times n}$$

In Python, We use numpy library which helps us in creating n dimensional arrays. Which are basically matrices, we use matrix method and pass in the lists and thereby defining a matrix.

Tensors

The more general entity of a tensor encapsulates the scalar, vector and the matrix. It is sometimes necessary — both in the physical sciences and machine learning — to make use of tensors with order that exceeds two.



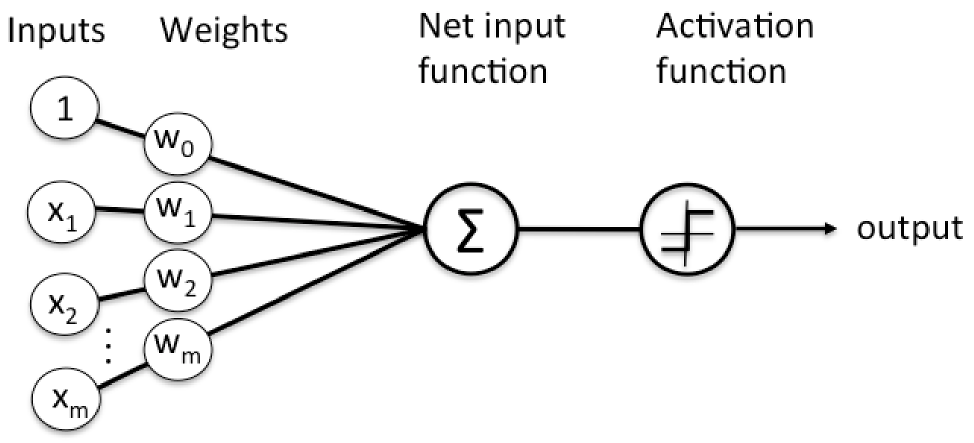
TENSOR

We use Python libraries like tensorflow or PyTorch in order to declare tensors, rather than nesting matrices.

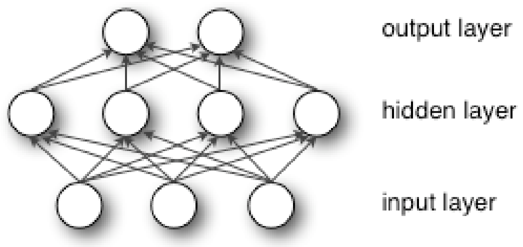
2.2 Neural Networks and Deep Learning

Deep learning is a class of machine learning algorithms that uses multiple layers to progressively extract higher-level features from the raw input. For example, in image processing, lower layers may identify edges, while higher layers may identify the concepts relevant to a human such as digits or letters or faces. In deep learning, each level learns to transform its input data into a slightly more abstract and composite representation. In an image recognition application, the raw input may be a matrix of pixels; the first representational layer may abstract the pixels and encode edges; the second layer may compose and encode arrangements of edges; the third layer may encode a nose and eyes; and the fourth layer may recognize that the image contains a face. Importantly, a deep learning process can learn which features to optimally place in which level *on its own*. This does not completely eliminate the need for hand-tuning; for example, varying numbers of layers and layer sizes can provide different degrees of abstraction.

Deep learning is the name we use for “stacked neural networks”; that is, networks composed of several layers.
The layers are made of *nodes*. A node is just a place where computation happens, loosely patterned on a neuron in the human brain, which fires when it encounters sufficient stimuli. A node combines input from the data with a set of coefficients, or weights, that either amplify or dampen that input, thereby assigning significance to inputs with regard to the task the algorithm is trying to learn; e.g. which input is most helpful is classifying data without error? These input-weight products are summed and then the sum is passed through a node’s so-called activation function, to determine whether and to what extent that signal should progress further through the network to affect the ultimate outcome, say, an act of classification. If the signal passes through, the neuron has been “activated.” Here’s a diagram of what one node might look like.



A node layer is a row of those neuron-like switches that turn on or off as the input is fed through the net. Each layer's output is simultaneously the subsequent layer's input, starting from an initial input layer receiving your data. Pairing the model's adjustable weights with input features is how we assign significance to those features with regard to how the neural network classifies and clusters input.



Activation Functions

An activation function in a neural network defines how the weighted sum of the input is transformed into an output from a node or nodes in a layer of the network.The choice of activation function has a large impact on the capability and performance of the neural network, and different activation functions may be used in different parts of the model.Technically, the activation function is used within or after the internal processing of each node in the network, although networks are designed to use the same activation function for all nodes in a layer.

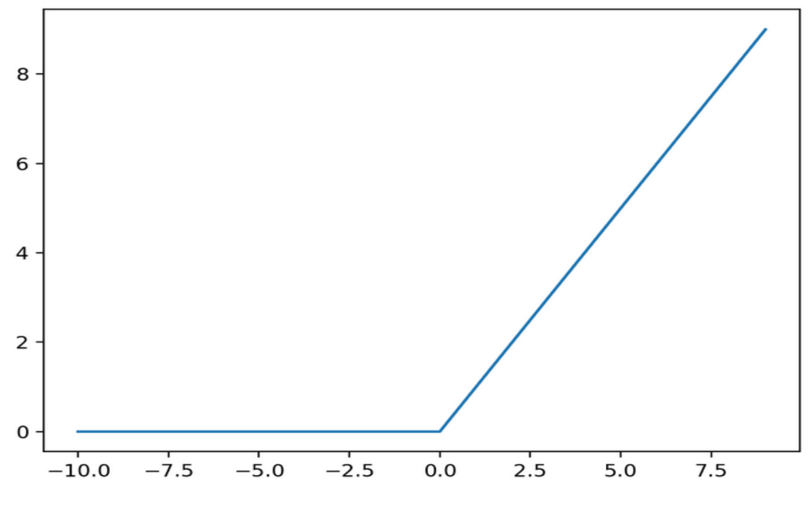
There are three activation functions you may want to consider for use in hidden layers; they are:

1) Rectified Linear Activation (**ReLU**)

It is the most common activation Function because it is both simple to implement and effective at overcoming the limitations of other previously popular activation functions, such as Sigmoid and Tanh. Specifically, it is less susceptible to vanishing gradients that prevent deep models from being trained, although it can suffer from other problems like saturated or "*dead*" units.

The ReLU function is calculated as follows:

- max(0.0, x)



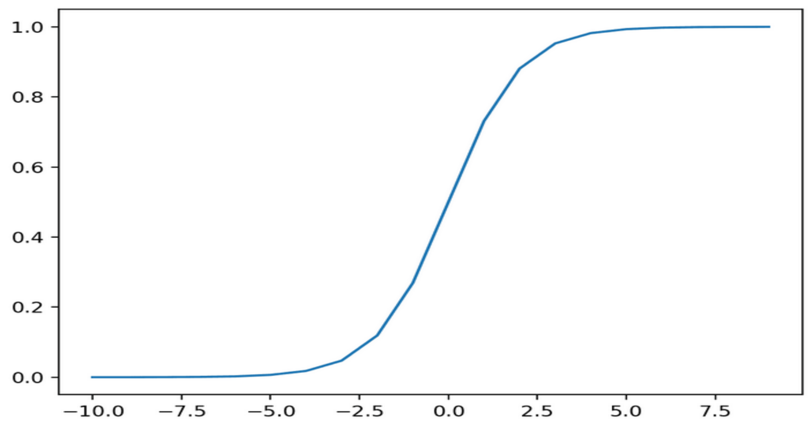
Plot of Inputs vs. Outputs for the ReLU Activation Function.

2) Logistic (**Sigmoid**)

It is the same function used in the logistic regression classification algorithm.The function takes any real value as input and outputs values in the range 0 to 1. The larger the input (more positive), the closer the output value will be to 1.0, whereas the smaller the input (more negative), the closer the output will be to 0.0.

The sigmoid activation function is calculated as follows:

- 1.0 / (1.0 + e^-x)



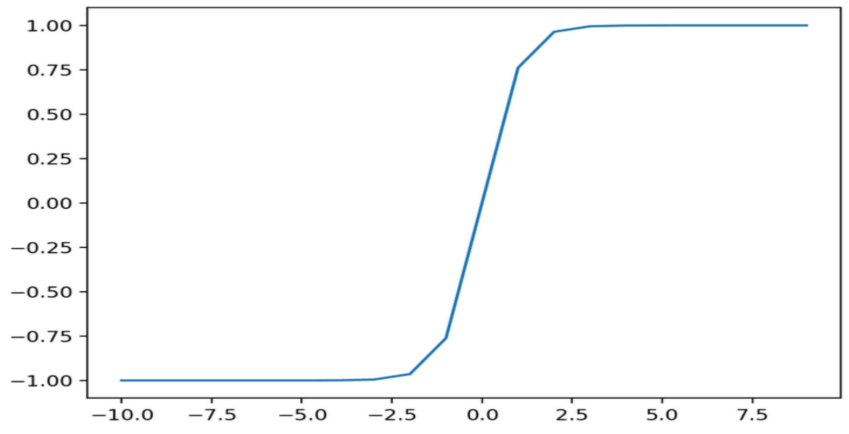
Plot of Inputs vs. Outputs for the Sigmoid Activation Function.

3) Hyperbolic Tangent (**Tanh**)

It is very similar to the sigmoid activation function and even has the same S-shape. The function takes any real value as input and outputs values in the range -1 to 1. The larger the input (more positive), the closer the output value will be to 1.0, whereas the smaller the input (more negative), the closer the output will be to -1.0.

The Tanh activation function is calculated as follows:

- (e^x – e^-x) / (e^x + e^-x)



Plot of Inputs vs. Outputs for the Tanh Activation Function.

Gradient Descent

Gradient Descent is a process that occurs in the **backpropagation** phase where the goal is to continuously resample the gradient of the model's parameter in the opposite direction based on the weight *w*, updating consistently until we reach the **global minimum** of function *J(w)*.To put it simply, we use gradient descent to minimize the cost function, *J(w)*.

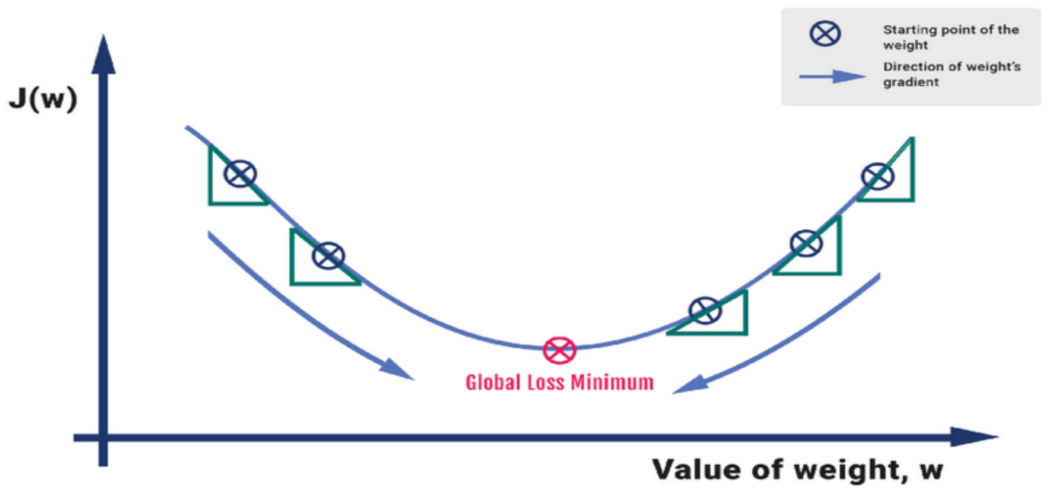


Fig 1: How Gradient Descent works for one parameter, w

Adam optimization

Adaptive Moment Estimation (Adam) is another method that computes adaptive learning rates for each parameter. In addition to storing an exponentially decaying average of past squared gradients vt like Adadelta and RMSprop, Adam also keeps an exponentially decaying average of past gradients mt, similar to momentum. Whereas momentum can be seen as a ball running down a slope, Adam behaves like a heavy ball with friction, which thus prefers flat minima in the error surface. We compute the decaying averages of past and past squared gradients mt and vt

respectively as follows:

mt=β1mt−1+(1−β1)gtvt=β2vt−1+(1−β2)g2t

mt and vt are estimates of the first moment (the mean) and the second moment (the uncentered variance) of the gradients respectively, hence the name of the method. As mt and vt are initialized as vectors of 0's, the authors of Adam observe that they are biased towards zero, especially during the initial time steps, and especially when the decay rates are small (i.e. β1 and β2 are close to 1).

They counteract these biases by computing bias-corrected first and second moment estimates:

^mt=mt1−βt1^vt=vt1−βt2

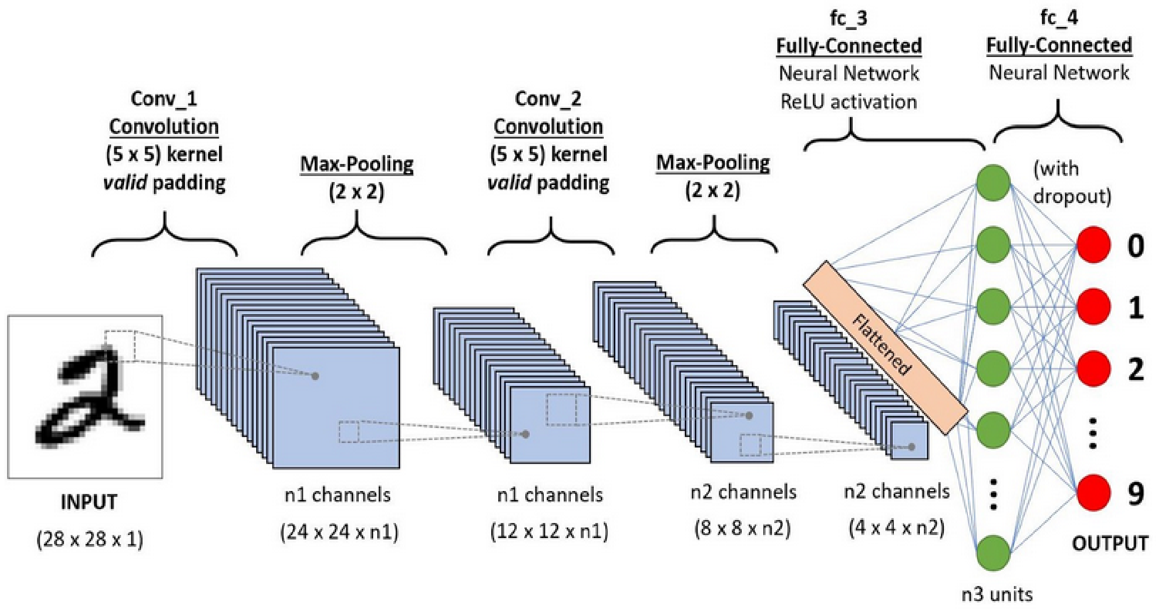
They then use these to update the parameters just as we have seen in Adadelta and RMSprop, which yields the Adam update rule:

θt+1=θt−η√^vt+ϵ^mt

The authors propose default values of 0.9 for β1, 0.999 for β2, and 10−8 for ϵ. They show empirically that Adam works well in practice and compares favorably to other adaptive learning-method algorithms.

2.3 Convolutional Neural Networks

A Convolutional Neural Network (ConvNet/CNN) is a Deep Learning algorithm which can take in an input image, assign importance (learnable weights and biases) to various aspects/objects in the image and be able to differentiate one from the other. The pre-processing required in a ConvNet is much lower as compared to other classification algorithms. While in primitive methods filters are hand-engineered, with enough training, ConvNets have the ability to learn these filters/characteristics.



A CNN sequence to classify handwritten digits

Convolutional neural networks are distinguished from other neural networks by their superior performance with image, speech, or audio signal inputs. They have three main types of layers, which are:

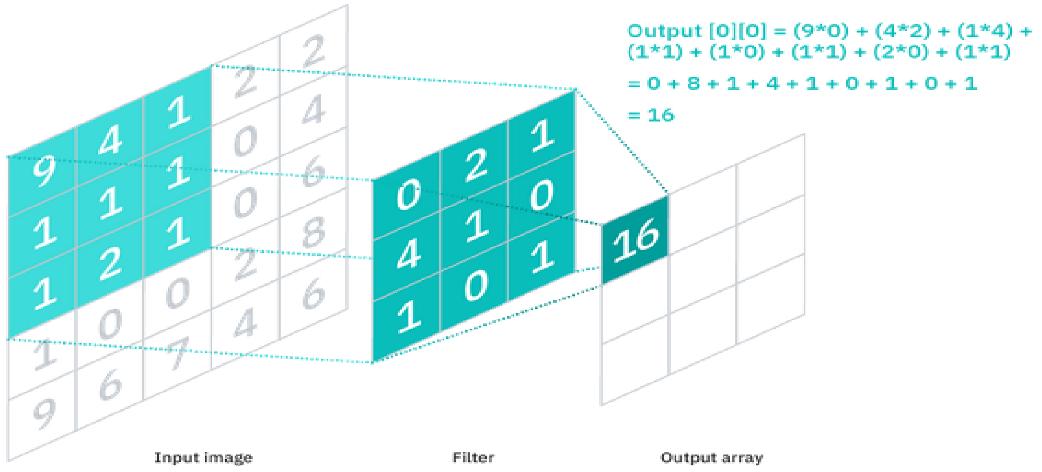
- Convolutional layer
- Pooling layer
- Fully-connected (FC) layer

The convolutional layer is the first layer of a convolutional network. While convolutional layers can be followed by additional convolutional layers or pooling layers, the fully-connected layer is the final layer. With each layer, the CNN increases in its complexity, identifying greater portions of the image. Earlier layers focus on simple features, such as colors and edges. As the image data progresses through the layers of the CNN, it starts to recognize larger elements or shapes of the object until it finally identifies the intended object.

Convolutional Layer

The convolutional layer is the core building block of a CNN, and it is where the majority of computation occurs. It requires a few components, which are input data, a filter, and a feature map. Let's assume that the input will be a color image, which is made up of a matrix of pixels in 3D. This means that the input will have three dimensions—a height, width, and depth—which correspond to RGB in an image. We also have a feature detector, also known as a kernel or a filter, which will move across the receptive fields of the image, checking if the feature is present. This process is known as a convolution.

The filter is applied to an area of the image, and a dot product is calculated between the input pixels and the filter. This dot product is then fed into an output array. Afterwards, the filter shifts by a stride, repeating the process until the kernel has swept across the entire image. The final output from the series of dot products from the input and the filter is known as a feature map, activation map, or a convolved feature.



There are three hyperparameters which affect the volume size of the output that need to be set before the training of the neural network begins. These include:

1. The **number of filters** affects the depth of the output. For example, three distinct filters would yield three different feature maps, creating a depth of three.

2. **Stride** is the distance, or number of pixels, that the kernel moves over the input matrix. While stride values of two or greater is rare, a larger stride yields a smaller output.

3. **Zero-padding** is usually used when the filters do not fit the input image. This sets all elements that fall outside of the input matrix to zero, producing a larger or equally sized output

<u>Pooling Layer</u>

Pooling layers, also known as downsampling, conducts dimensionality reduction, reducing the number of parameters in the input. Similar to the convolutional layer, the pooling operation sweeps a filter across the entire input, but the difference is that this filter does not have any weights. Instead, the kernel applies an aggregation function to the values within the receptive field, populating the output array. There are two main types of pooling:

- **Max pooling:** As the filter moves across the input, it selects the pixel with the maximum value to send to the output array. As an aside, this approach tends to be used more often compared to average pooling.
- **Average pooling:** As the filter moves across the input, it calculates the average value within the receptive field to send to the output array.

Fully-Connected Layer

The name of the full-connected layer aptly describes itself. As mentioned earlier, the pixel values of the input image are not directly connected to the output layer in partially connected layers. However, in the fully-connected layer, each node in the output layer connects directly to a node in the previous layer.

This layer performs the task of classification based on the features extracted through the previous layers and their different filters. While convolutional and pooling layers tend to use ReLu functions, FC layers usually leverage a softmax activation function to classify inputs appropriately, producing a probability from 0 to 1.

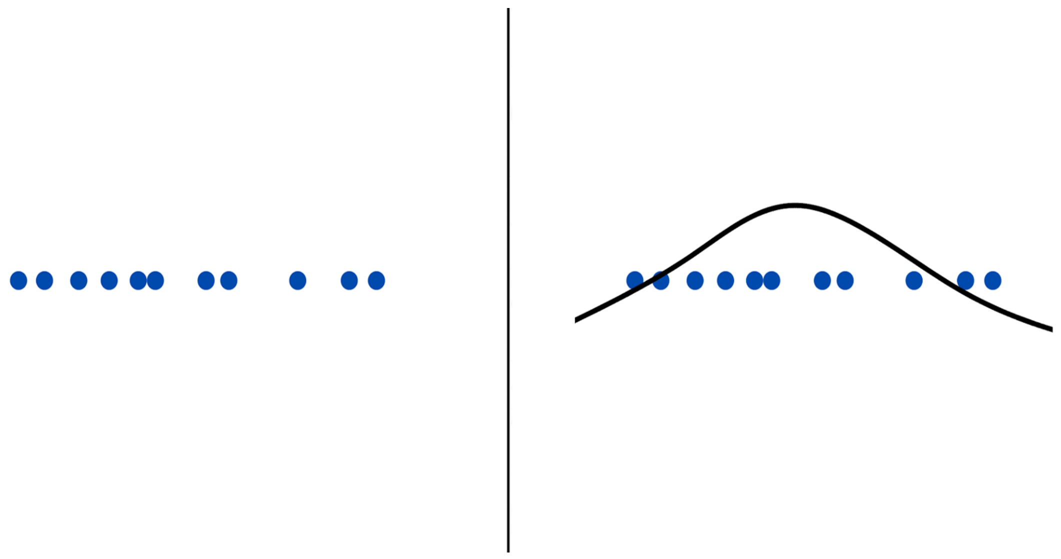
There are various architectures of CNNs available which have been key in building algorithms which power and shall power AI as a whole in the foreseeable future. Some of them have been listed below:

1. LeNet
2. AlexNet
3. VGGNet
4. GoogLeNet
5. ResNet
6. ZFNet

2.4 GANs and CGANs

GANs are a relatively recent invention in the field of ML.Deep Learning can be divided into two types of model objectives, which are

- **Discriminative Models**
  These models are used to map a single possible output from the given input data. The most common example you can think of, in this domain is a classifier. Their goal is to simply identify a class of the input data such as 'spam or not spam', or like handwritten character recognition, and others.
  These models capture the conditional probability P(y|x), which is 'Probability of y given x'.
- **Generative Models**
  These models are used to find a probability distribution of a dataset and generate similarly structured data.
  Generative models are primarily meant to find the **density function** from the given probability distribution of the data. As shown in the diagram below, the points represent the distribution of the data in a 1-dimensional axis which is fitted by the gaussian density in the right image.

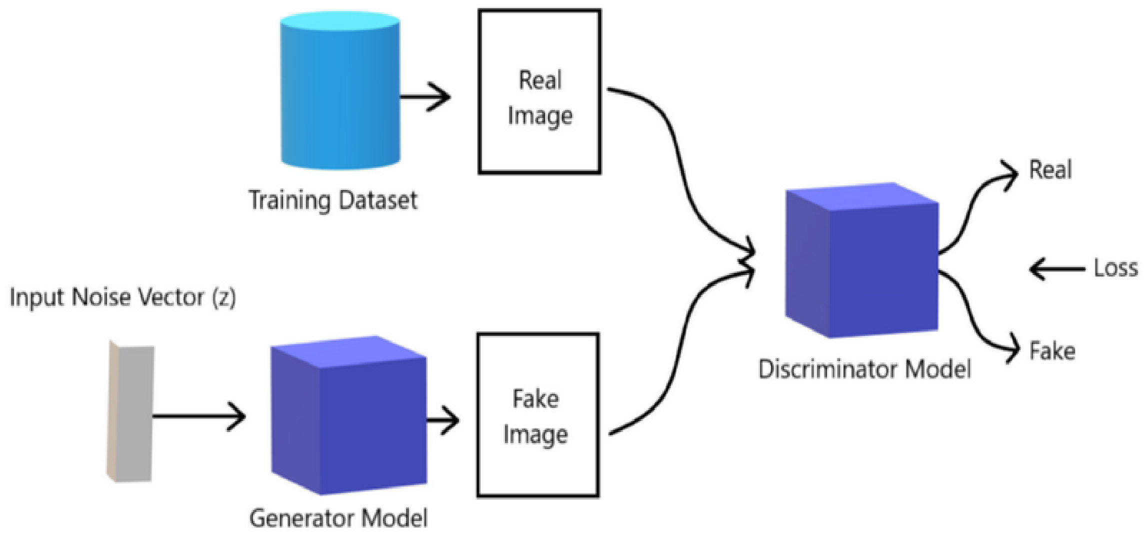


GANs do not focus on finding this density function accurately rather they observe the given dataset and generate new samples that fit the underlying structure in the given data samples by the help of two models which are **adversaries** of each other. Hence the name — ***Generative Adversarial Networks***

Working of GANs

GANs consist of two models, namely:

- **Generator**
  Its function is to take an input noise vector (z) and map it to an image that hopefully resembles the images in the training dataset.
- **Discriminator**
  The primary purpose of the discriminator model is to find out which image is from the actual training dataset and which is an output from the generator model.



Basic Structure of GANs consisting of the Generator and the Discriminator Models (Image by Author)

You can imagine the generator model to be counterfeiters who want to generate fake currency and fool everyone in believing that it is real, and the discriminator model is the police who want to identify the fake currency and catch the counterfeiters.

At the beginning the counterfeiters generate random currency that does not resemble the real currency at all. After being caught by the police, they learn from the mistakes [loss of the models in our case]and generate new currency which is better than the previous one.

100
100
Fake
Real

An example where the fake currency is not similar to the real one

This way, the police get better at discriminating the fake money from the real one, and simultaneously, the counterfeiters get better at generating money that looks similar to the real money.

The point where counterfeiters become well trained in generating fake money

This is a min-max 2-player game between the models where the generator model tries to minimize its loss and maximize the discriminator loss.As a result, the generator model maps the input vector (z) to an output which is similar to the data in the training dataset.

In the end, there comes a point when the discriminator can no longer identify the fake output, making its accuracy approximately 50%. This means that it is now making a random guess in discriminating between fake and real data. This is known as the point of Nash Equilibrium.

Cycle-GANs were able to map input classes to the desired output set. The training is done on two sets of data and with no other labels required. The model learns to transform one set of images into the other.

The most amazing thing about GANs is that it is a very simple implementation of a rather complex objective.If you have a basic understanding of how a Convolutional Neural Network works and the base of backpropagation, then you can pretty much start working on GANs.

3. IMPLEMENTATION:

3.1 Initialization

First, We initialized the parameters to configure the training of the model. We used the TensorFlow framework so we imported it using *import tensorflow as tf* .

The os module is used to interact with the Operating System. We used this for accessing and modifying the path variables to save checkpoints during training. The time module lets us display relative time and hence, we can check how much time each epoch took during the training.

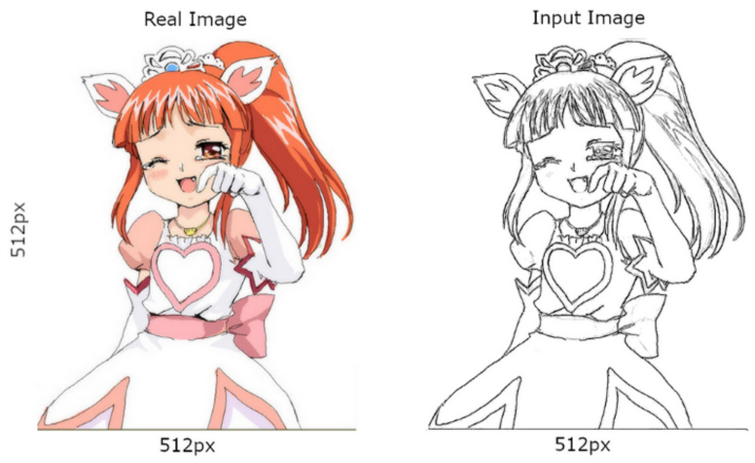
matplotlib is another cool python library which we used to plot and show images.

BUFFER_SIZE is used when we shuffle the data samples while training. Higher the value of this more will be the degree of shuffling, and hence, higher will be the accuracy of the model. But with large data, it takes a lot of processing power to shuffle the images.

BATCH_SIZE is used to divide the dataset into mini-batches for training. The higher this value is, the faster will be the process of training. But as you might have guessed already, higher batch size means a higher load on the machine.

3.2 Preprocessing

The dataset contains a single image of size 1024x512 px for one entry which has a colored image of size 512x512 px in the left and a black and white sketch image of size 512x512 px in the right.



Therefore we defined a function load() that takes the image path as a parameter and returns an input_image which is the black and white sketch that we'll give as an input to the model, and real_image which is the colored image that we want.

Then we did some preprocessing in order to prepare the data for the model.Given below are a few easy functions used for this purpose.

resize() function is used to return the images as 286x286 px. This is done in order to have a uniform image size if by chance there is a differently sized image in the dataset. And decreasing size from 512x512 px to half of it also helps in speeding up the model training as it is computationally less heavy.

random_crop() function returns the cropped input and real images which have the desired size of 256x256 px.

normalize() function, as the name suggests, normalizes images to [-1, 1].

In the random_jitter() function, all the previous preprocessing functions are put together and random images are flipped horizontally.

3.3 Loading the Train and Test Data

To load the train dataset we used the following functions:
load_image_train() function is used to put together all the previously seen

functions and output the final preprocessed image.

tf.data.Dataset.list_files() collects the path to all the png files available in the train/ folder of the dataset. Then the collection of these paths is mapped through and every path is sent individually as an argument to the load_image_train() function which returns the final preprocessed image and adds it to the train_dataset .

Finally, this train_dataset is shuffled using the BUFFER_SIZE and then divided into mini-batches.

To load the test dataset, we used a similar process except for a small change. Here we omitted the random_crop() and random_jitter() functions as there is no need to do this for testing the results. Also, we omittted to shuffle the dataset for the same reason.

3.4 Generator Model

Next we built the generator model, which takes an input black and white sketch image of 256x256 px and outputs an image that hopefully resembles the colored ground truth image in the training dataset.

The Generator model is a UNet Architecture Model and has skip connections to other layers than the intermediate one. It becomes complex to design such an architecture as the output and input shapes need to match to the connected layers.

The downsampling stack of layers has Convolutional layers which result in a decrease in the size of the input image. And once the decreased image goes through the upsampling stack of layers which has kind of “reverse” Convolutional layers, the size is restored back to 256x256 px. Hence, the output of the Generator Model is a 256x256 px image with 3 output channels.

3.5 Discriminator Model

The primary purpose of the discriminator model is to find out which image is from the actual training dataset and which is an output from the generator model.

This is not as complex as the Generator model as it's fundamental task is just to classify real and fake images.

3.6 Loss Functions

As we have two models with us, we required two different loss functions to calculate their loss independently.

The loss for the generator was calculated by finding the sigmoid cross-entropy loss of the output of the generator and an array of ones. This means that we are training it to trick the discriminator in outputting the value as 1, which means that it is a real image. Also, for the output to be structurally similar to the target image, we take L1 loss along with it. The value of LAMBDA is suggested to be kept 100 by authors of the *original paper*.

For discriminator loss, we took the same sigmoid cross-entropy loss of the real images and an array of ones and added it with the cross-entropy loss of the output images of the generator model and array of zeros.

3.7 Optimizers

Optimizers are algorithms or methods used to change the attributes of your neural network such as weights and learning rates in order to reduce the losses. Adam Optimizer is one of the best ones to use, in most of the use cases.

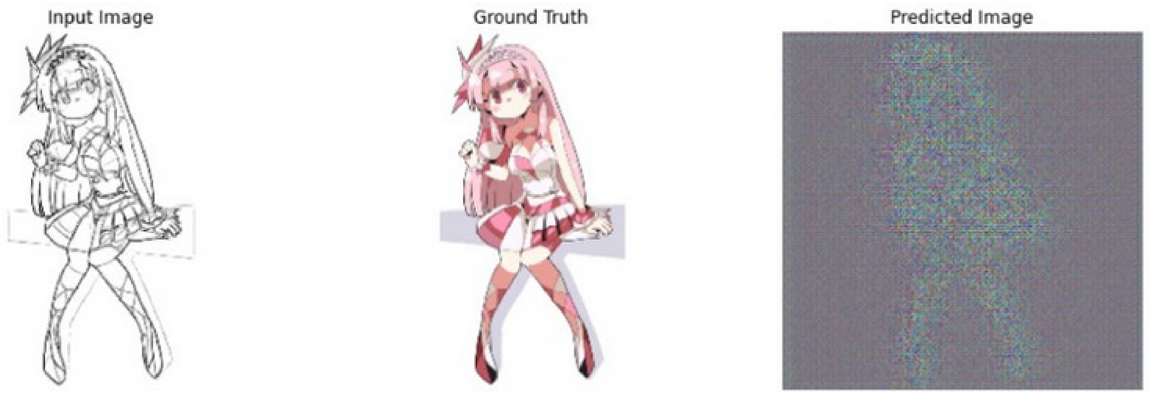
3.8 Creating Checkpoints

The cloud environments have a specific timeout which can interrupt the training process. Also, if you are using your local system, there may arise some cases where the training might be interrupted due to some reasons.

GANs take a very long time to train and are computationally expensive. So, it is best to keep saving checkpoints at regular intervals so that you can restore to the latest checkpoint and continue from there without losing the previously done hard work by your machines.

3.9 Displaying Output Images

Here we implemented a basic python function which used the pyplot module from matplotlib library to display the predicted images by the generator model.



3.10 Logging the Losses

Here we logged an important metric - losses in a file so that we could analyze it as the training progressed on Tensorboard.

3.11 Training

It consist of the following processes:

- The generator outputs a prediction
- The discriminator model is designed to have 2 inputs at a time. For the first time, it is given an input sketch image and the generated image. The next time it is given the real target image and the generated image.
- Now the generator loss and discriminator loss are calculated.
- Then, the gradients are calculated from the losses and applied to the optimizers to help the generator produce a better image and also to help discriminator detect the real and generated image with better insights.
- All the losses are logged using summary_writer defined previously using tf.summary .

Next we defined the model.fit() function where we iterate over for every epoch and assign the relative time to start variable. Then we display an example of the generated image by the generator model. This example helps us visualize how the generator gets better at generating better-colored images with every epoch. Then we call the train_step function for the model to learn from the calculated losses and gradients. And finally, we check if the epoch number is divisible by 5 to save a checkpoint. This means that we are saving a checkpoint after every 5 epochs of training are completed. After this entire epoch is completed, the start time is subtracted from the final relative time to count the time taken for that particular epoch.

Now all we had to do is run this one line of code and wait for the Model to do its magic on its own.

```
fit(train_dataset, EPOCHS, test_dataset)
```

3.12 Restoring Checkpoints

Before moving forward, we restored the latest checkpoint available in order to load the latest version of the trained model before testing it on the images.

3.13 Testing output and Saving Model

This randomly selects 5 images from the test_dataset and inputs them individually to the Generator Model. Now the model is trained well enough and predicts near-perfect colored versions of the input sketch images.

Finally we saved the entire model as a .H5 file which is supported by Keras models.

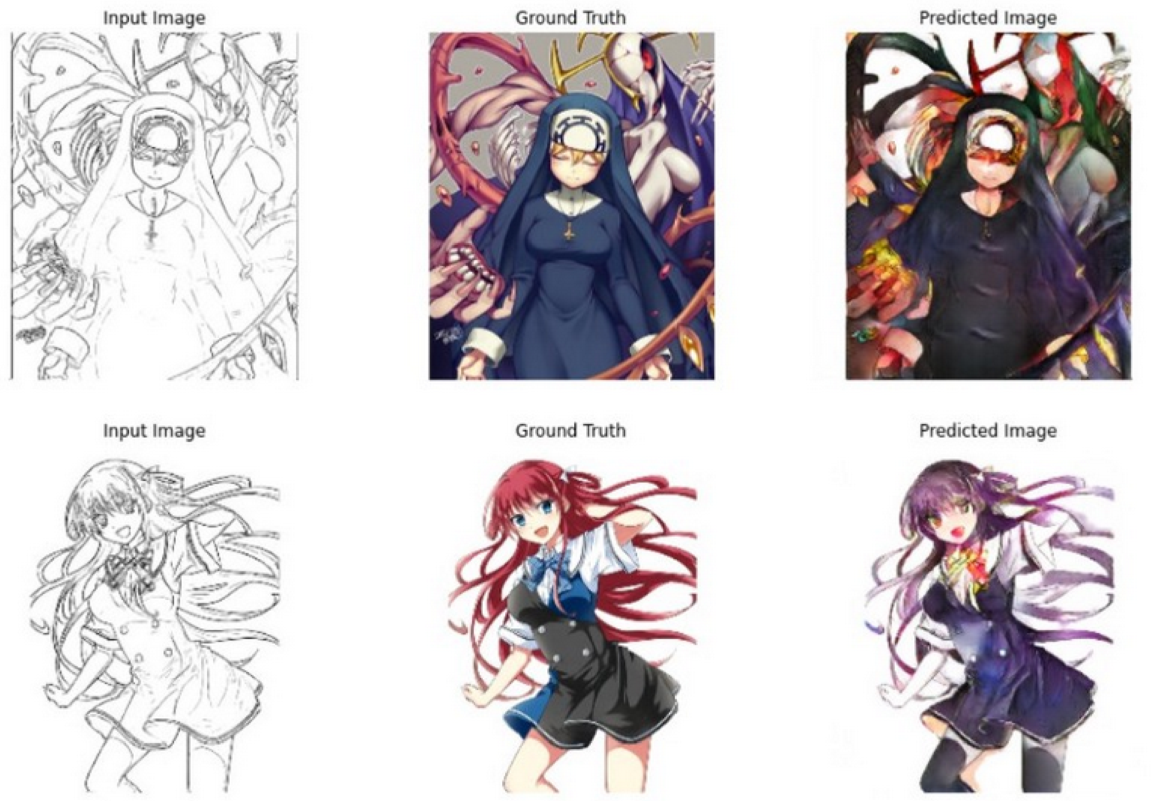
```
generator.save('AnimeColorizationModelv1.h5')
```

4.CONCLUSION AND FUTURE WORK

4.1 Conclusion and Efficiency

We have not only seen how a Conditional GAN works but also have successfully implemented it to predict colored images from the given black and white input sketch images.

When we trained the model on our system, we ran the model for 100 epochs which took approximately 23 hours on a single GeForce GTX 1060 4 GB Graphic Card and 8 GB RAM. After all that hard work and patience, the results were totally worth it!



4.2 Future Aspects

We enjoyed working on GANs during our project and plan to continue exploring the field for further applications and make new projects. Some of the points that We think this project can grow or be a base for are listed below.

1. Trying different databases to get an idea of preprocessing different types of images and building models specific to those input image types.
2. This is a project applied on individual Image to Image translation. Further the model can be used to process black and white sketch video frames to generate colored videos.
3. Converting the model from HDF5 to json and building interesting web apps using TensorFlow.Js.

5. REFERENCES

5.1 Useful links & Research Papers / Helpful Courses

[1] Generative Adversarial Networks (GANs) articles:
https://towardsdatascience.com/generative-adversarial-networks-gans-8fc303ad5fa1

https://machinelearningmastery.com/what-are-generative-adversarial-networks-gans/

[2] Deep Learning Courses:
https://www.coursera.org/specializations/deep-learning

[3] Research Paper-GANs: https://arxiv.org/pdf/1611.07004.pdf

[4] Sketch-2-Paint articles:

https://towardsdatascience.com/generative-adversarial-networks-gans-89ef35a60b69

https://www.tensorflow.org/tutorials/generative/pix2pix

[5] Linear Algebra:
https://www.youtube.com/playlist?list=PLZHQObOWTQDPD3MizzM2xVFitgF8hE_ab

[6] Unet Architecture:
https://towardsdatascience.com/understanding-semantic-segmentation-with-unet-6be4f42d4b47

[7] Tensorflow:
https://www.tensorflow.org/tutorials