Neural Style Transfer and more

Project Report

EKLAVYA MENTORSHIP PROGRAM
AT

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

AUGUST 2022

ACKNOWLEDGEMENT

We are extremely grateful to our mentors Neel Shah and Pratham Shah for their support and guidance throughout the duration of the project.

We would also like to thank all the members of SRA VJTI for their timely support as well as for organizing Eklavya and giving us a chance to work on this project.

Labib Asari

labeebasari@gmail.com

Lakshaya Singhal

singhallakshaya@gmail.com

OVERVIEW

Deep learning is a machine learning subset that makes computers do what comes naturally to humans: learn by example.

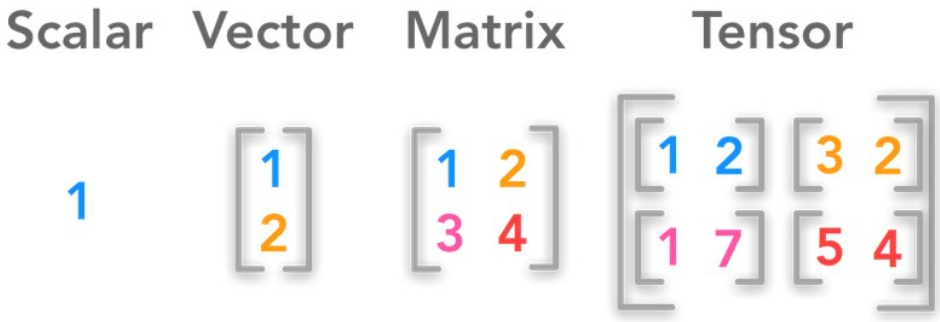
Deep neural networks have already surpassed human level performance in tasks such as object recognition and detection. However, deep networks were lagging far behind in tasks like generating artistic artifacts having high perceptual quality until recent times. Creating better quality art using machine learning techniques is imperative for reaching human-like capabilities, as well as opens up a new spectrum of possibilities. And with the advancement of computer hardware as well as the proliferation of deep learning, deep learning is right now being used to create art.

Neural style transfer is an optimization technique used to take two images — a content image and a style reference image (such as an artwork by a famous painter) — and blend them together so the output image looks like the content image, but "painted" in the style of the style reference image.

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.



1.1 Scalars

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

There are different sets of numbers of interest in deep learning. $\mathbb{N}$ represents the set of positive integers (1,2,3,…). $\mathbb{Z}$ 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

1.2 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 $\mathbb{R}$^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.

1.3 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:

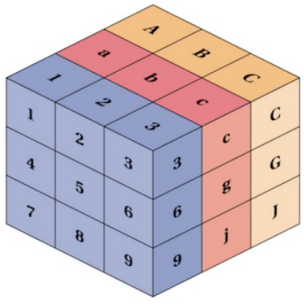
$$A = \begin{bmatrix} a_{11} & a_{12} & a_{13} & \cdots & a_{1n} \\ a_{21} & a_{22} & a_{23} & \cdots & a_{2n} \\ a_{31} & a_{32} & a_{33} & \cdots & a_{3n} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ a_{m1} & a_{m2} & a_{m3} & \cdots & a_{mn} \end{bmatrix}$$

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

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.

1.4 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.

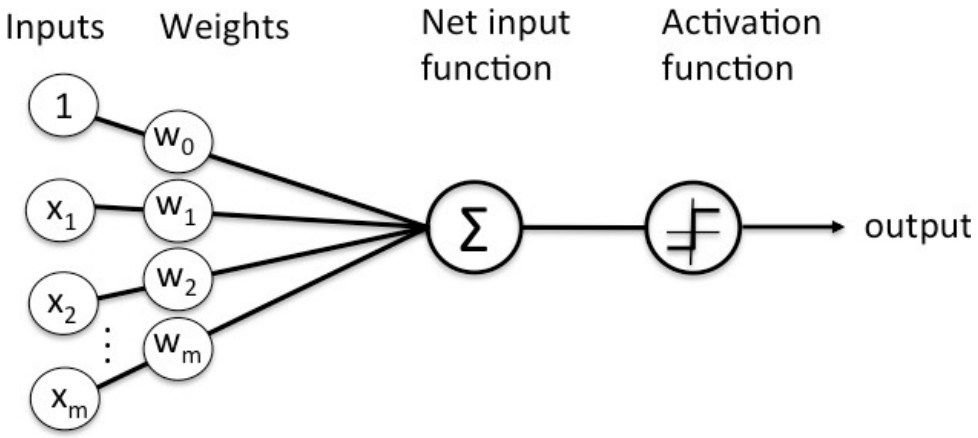


TENSOR

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

2. 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."

2.1 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. The activation function is used within or after the internal processing of each node in the neural network. The biggest role activation functions play is in the top layer or the final layer where the correct activation function helps give us probabilities in a classification problem by plotting the output of the neural network between 0 and 1.

Why is an activation function necessary?

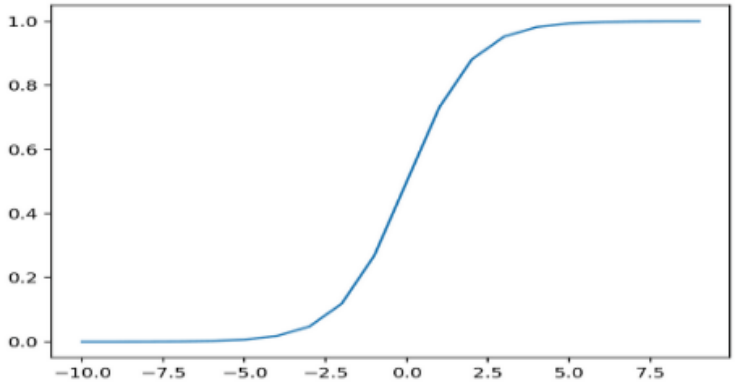
Activation functions are necessary to prevent linearity. Without them, the data would pass through the nodes and layers of the network only going through linear functions (a*x+b). The composition of these linear functions is again a linear function and so no matter how many layers the data goes through, the output is always the result of a linear function. Also, they aid in classification problems by plotting the output of the neural network between 0 and 1.

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

1) 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. Its calculated as follows:

$$\frac{1}{1+e^{-x}}$$



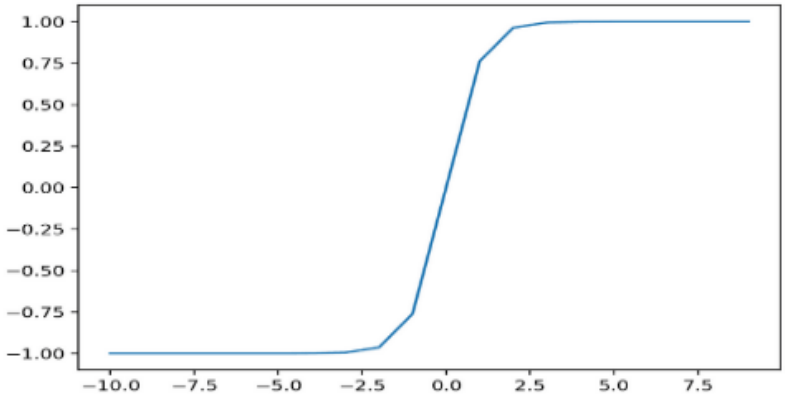
Plot of Inputs vs. Outputs for the Sigmoid Activation Function.

2) 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.

It is calculated as follows:

$$\frac{e^{x} - e^{-x}}{e^{x} + e^{-x}}$$



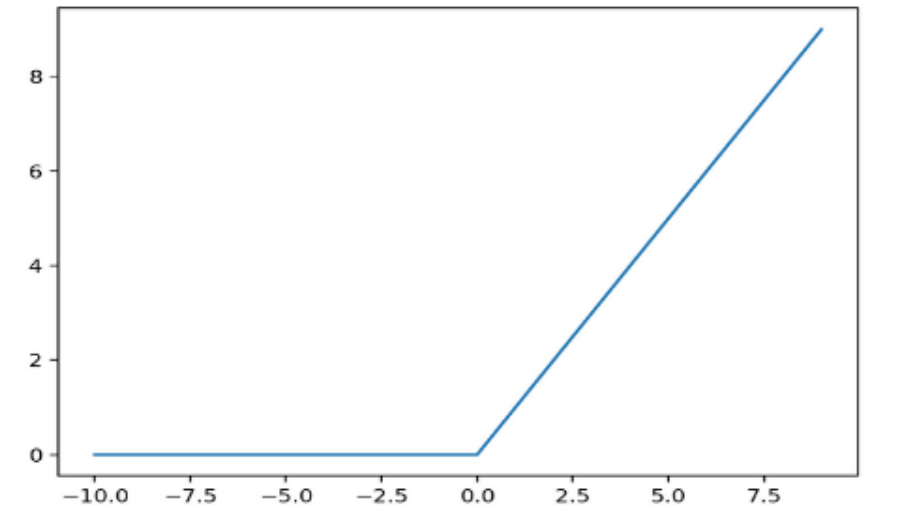
Plot of Inputs vs. Outputs for the Tanh Activation Function.

3) 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.

It can be calculated as follows:

$$\max(0,x)$$



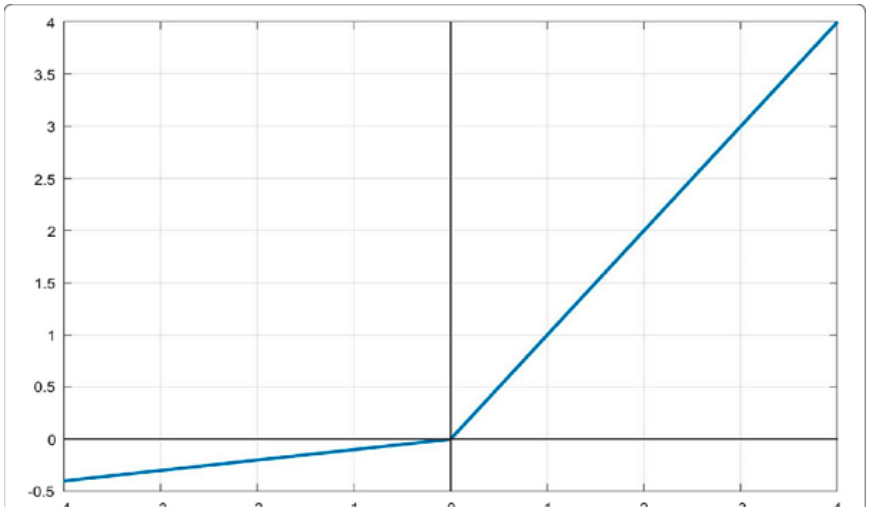
Plot of Inputs vs. Outputs for the ReLU Activation Function.

4) Leaky ReLU:

Leaky Rectified Linear Unit, or Leaky ReLU, is a type of activation function based on a ReLU, but it has a small slope for negative values instead of a flat slope. The slope coefficient is determined before training, i.e. it is not learnt during training.

It can be calculated as follows:

$$\max(0.01*x,x)$$



2.2 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). Gradient Descent is the algorithm that facilitates the search of parameters values that minimize the cost function towards a local minimum or optimal accuracy. To put it simply, we use gradient descent to minimize the cost function, J(w).

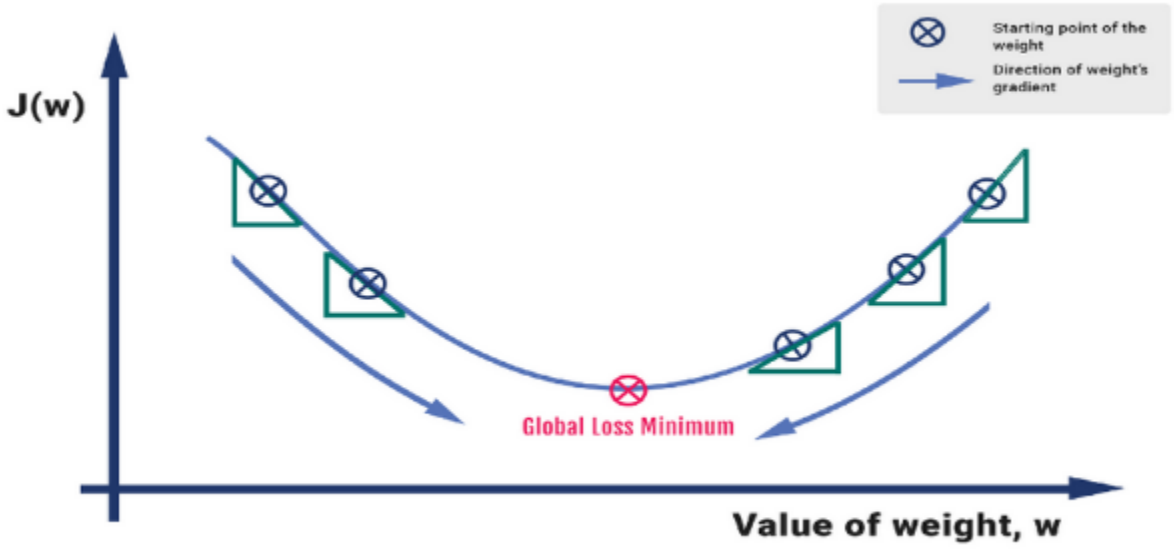


Fig 1: How Gradient Descent works for one parameter, w

2.3 Neural Network Implementation using Numpy

We trained a shallow neural network on the MNIST dataset for digit recognition. As this implementation was from scratch, we made functions for all our tasks. These functions are listed below:

1) **sigmoid() , leakyrelu() , softmax()** are the activation functions.
2) **initialize_parameters()** sets the weights to random num,bers and biases to 0.
3) **linear_forward()** perform forward propagation for one layer.
4) **linear_activation_forward()** applies activation function to the output of forward propagation.
5) **L_forward()** produces forward propagation result for the entire neural network.
6) **compute_cost()** computes the cost of the neural network.
7) **linear_backward() , softmax_backward() , leakyrelu_backward()** finds derivatives for a particular layer to perform backward propagation.
8) **linear_activation_backward()** finds derivatives for all the layers
9) **L_model_backward()** uses derivatives from the previous functions to find changes to be made to the weights and biases.
10) **update_parameters()** updates the weights and biases.
11) **model()** uses all the above functions to train a model.

We trained a model having 4 layers with 100, 50, 25 and 10 nodes in each layer respectively. The last layer with 10 nodes used softmax activation to classify the input digits into 10.

Upon training this model, we were able to achieve a train accuracy of 97% and a test accuracy of 95.5%.

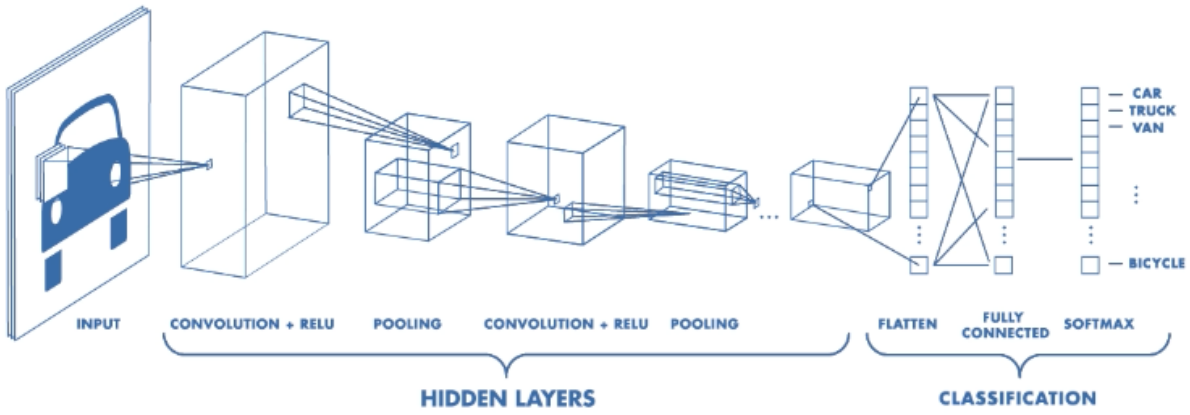
It is important to note that the general consensus right now is that the MNIST dataset is not a reliable way to learn neural networks since it is very easy to train. Infact, a simple convolutional neural network can achieve 99.7% accuracy. It also might paint an unrealistic picture of how simple the field of deep learning can be.

3. Convolutional Neural Networks

A Convolutional Neural Network, also known as CNN or ConvNet, is a class of neural networks that specializes in processing data that has a grid-like topology, such as an image. A digital image is a binary representation of visual data. It contains a series of pixels arranged in a grid-like fashion that contains pixel values to denote how bright and what color each pixel should be.

The human brain processes a huge amount of information the second we see an image. Each neuron works in its own receptive field and is connected to other neurons in a way that they cover the entire visual field. Just as each neuron responds to stimuli only in the restricted region of the visual field called the receptive field in the biological vision system, each neuron in a CNN processes data only in its receptive field as well. The layers are arranged in such a way so that they detect simpler patterns first (lines, curves, etc.) and more complex patterns (faces, objects, etc.) further along. By using a CNN, one can enable sight to computers.

A CNN typically has three layers: a convolutional layer, a pooling layer, and a fully connected layer.

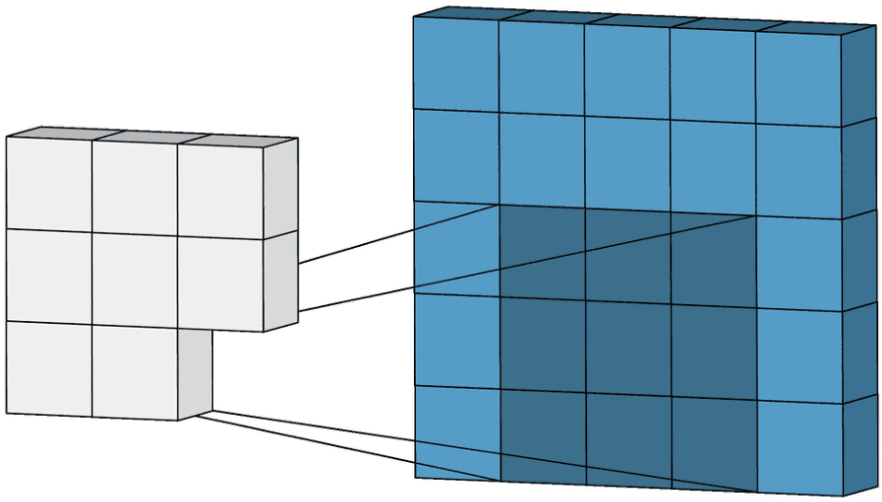


3.1 Convolutional Layer

The convolution layer is the core building block of the CNN. It carries the main portion of the network's computational load.

This layer performs a dot product between two matrices, where one matrix is the set of learnable parameters otherwise known as a kernel, and the other matrix is the restricted portion of the receptive field. The kernel is spatially smaller than an image but is more in-depth. This means that, if the image is composed of three (RGB) channels, the kernel height and width will be spatially small, but the depth extends up to all three channels.

During the forward pass, the kernel slides across the height and width of the image-producing the image representation of that receptive region. This produces a two-dimensional representation of the image known as an activation map that gives the response of the kernel at each spatial position of the image. The sliding size of the kernel is called a stride.



Convolution leverages three important ideas that motivated computer vision researchers: sparse interaction, parameter sharing, and equivariant representation.

1) Sparse Interaction:

   Trivial neural network layers use matrix multiplication by a matrix of parameters describing the interaction between the input and output unit. This means that every output unit interacts with every input unit. However, convolution neural networks have *sparse interaction.* This is achieved by making kernel smaller than the input e.g., an image can have millions or thousands of pixels, but while processing it using kernel we can detect meaningful information that is of tens or hundreds of pixels. This means that we need to store fewer parameters that not only reduces the memory requirement of the model but also improves the statistical efficiency of the model.

2) Parameter Sharing:

   If computing one feature at a spatial point (x1, y1) is useful then it should also be useful at some other spatial point say (x2, y2). It means that for a single two-dimensional slice i.e., for creating one activation map, neurons are constrained to use the same set of weights. In a traditional neural

network, each element of the weight matrix is used once and then never revisited, while convolution network has *shared parameters* i.e., for getting output, weights applied to one input are the same as the weight applied elsewhere.

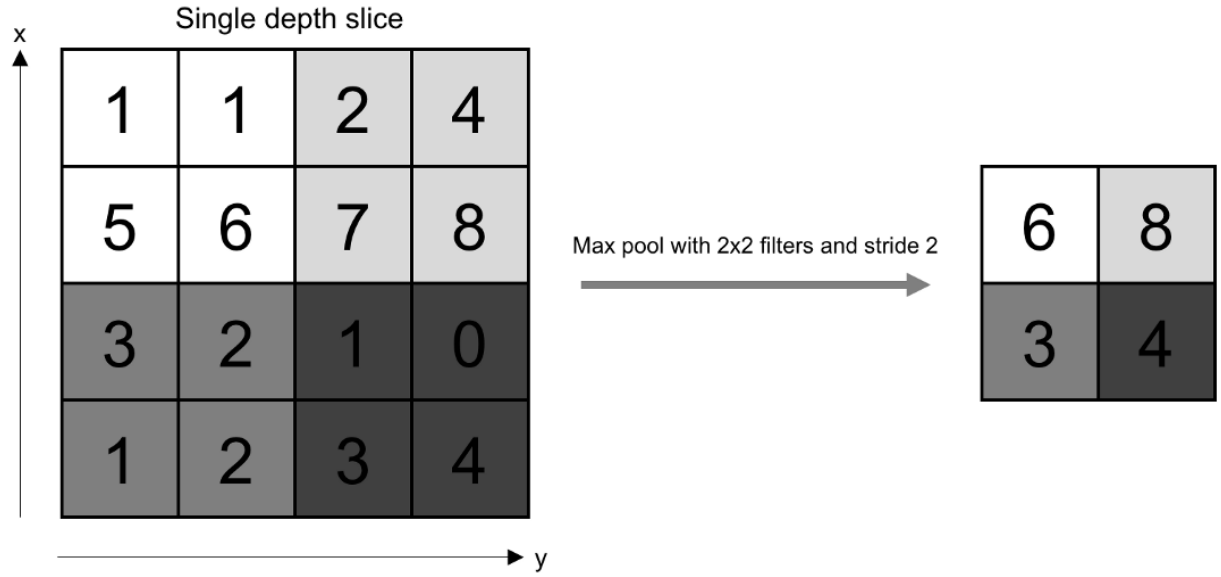
3) Equivariant Representation:

Due to parameter sharing, the layers of convolution neural network will have a property of *equivariance to translation*. It says that if we changed the input in a way, the output will also get changed in the same way.

3.2 Pooling Layer

The pooling layer replaces the output of the network at certain locations by deriving a summary statistic of the nearby outputs. This helps in reducing the spatial size of the representation, which decreases the required amount of computation and weights. The pooling operation is processed on every slice of the representation individually.

There are several pooling functions such as the average of the rectangular neighborhood, L2 norm of the rectangular neighborhood, and a weighted average based on the distance from the central pixel. However, the most popular process is max pooling, which reports the maximum output from the neighborhood.



3.3 Fully Connected Layer

These are layers of nodes in which each node of a layer is connected with every node of the previous and the next layer. These are also accompanied by an activation function (sigmoid, tanh, ReLU, LakyReLU).

These layers along with their features and components have been discussed in the previous section.

3.4 CNN Models

Various architectures of CNN have been studied and published in papers, a few of which are listed below:

1. LeNet
2. ALexNet
3. VGG Net
4. GoogLeNet
5. ResNet
6. ZFNet

Each model has its pros and cons with some models suited for a particular task more than others. To study and to understand further the features of a model that affects the performance of the model, we implemented AlexNet and VGG Net using Tensorflow.

1) AlexNet:

1) One of the largest neural networks on the subsets of ImageNet

2) 5 convolutional layers and 3 fully connected layers

3) Uses ReLU non-linearity because it does not saturate which makes sure the gradient descent doesn't slow down.

4) Trained on 2 GTX 580, 3GB.

5) The parallelization scheme puts half the kernels on each GPU. ALso the GPUs communicate only in some layers.

6) ReLU doesn’t need Normalization to prevent saturating but local normalization scheme aids generalization. Uses Local Response Normalization.

7) Pooling is generally done without overlapping where stride = filter size, but here overlapping pooling was used and it was found that top1 and top5 error rates reduced by 0.4% and 0.3% respectively.

8) Output is given by a 1000 way Softmax layer.

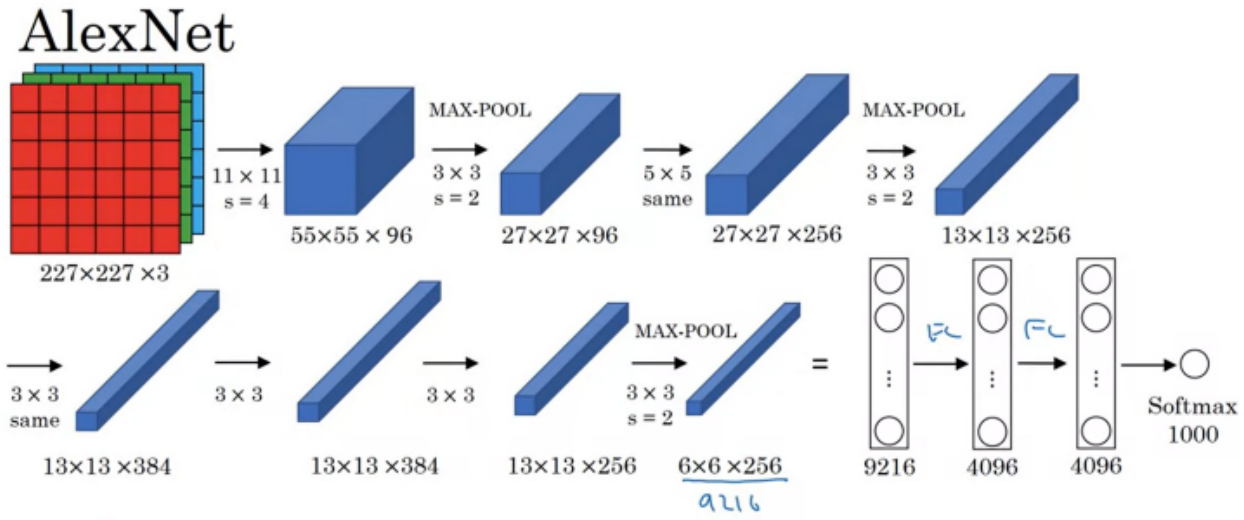
9) The kernels of the second, fourth, and fifth convolutional layers are connected only to those kernel maps in the previous layer which reside on the same GPU.

10) Overfitting prevention measures:

a. **Data Augmentation**: From each 256x256 image, 5 224x224 patches were taken(4 corners and 1 center) and they were also flipped horizontally. Also, PCA was performed on the set of RGB pixel values.

b. **Dropout**: For each training epoch, nodes were randomly dopped. This technique reduces complex co-adaptations of neurons, since a neuron cannot rely on the presence of particular other neurons. Used only in first 2 fully connected layers here.

11) Stochastic gradient descent was used with a batch size of 128, momentum of 0.9.



2) VGG Net

1. Input size is 224x224 RGB image. The preprocessing done is that the mean RGB value is subtracted, computed on the training set on each pixel.

2. Passed through multiple filters, all being 3x3. In one configuration 1x1 was also used.

3. Convolution stride is fixed at 1.

4. Padding is such that the size of the image doesn't change or 'same' type of padding.

5. Spatial pooling follow some of the conv. layers. MAx pooling is performed over 2x2 window with a stride of 2

6. 3 fully connected layers follow - 4096,4096, 1000.

7. Dropout with a probability of 0.5 is used after the first 2 FC layers.

8. The final layer is the softmax layer.

9. All hidden layers use ReLU activation.

10. Only one of the networks use local response normalization and the parameters are same as AlexNet.

11. The training was regularized by weight decay (the L2 penalty multiplier set to $5 \cdot 10^{-4}$)

12. The initialisation of the network weights is important, since bad initialisation can stall learning due to the instability of gradient in deep nets. To circumvent this problem, training began with the configuration A (Table 1), shallow enough to be trained with random initialisation. Then, when training deeper architectures, the first four convolutional layers and the last three fully- connected layers were initialized with the layers of net A (the intermediate layers were initialized randomly). The learning rate for the pre-initialised layers was not decreased, allowing them to change during learning.

ConvNet Configuration					
A	A-LRN	B	C	D	E
11 weight layers	11 weight layers	13 weight layers	16 weight layers	16 weight layers	19 weight layers
input (224 × 224 RGB image)					
conv3-64	conv3-64 **LRN**	conv3-64 **conv3-64**	conv3-64 conv3-64	conv3-64 conv3-64	conv3-64 conv3-64
maxpool					
conv3-128	conv3-128	conv3-128 **conv3-128**	conv3-128 conv3-128	conv3-128 conv3-128	conv3-128 conv3-128
maxpool					
conv3-256 conv3-256	conv3-256 conv3-256	conv3-256 conv3-256	conv3-256 conv3-256 **conv1-256**	conv3-256 conv3-256 **conv3-256**	conv3-256 conv3-256 conv3-256 **conv3-256**
maxpool					
conv3-512 conv3-512	conv3-512 conv3-512	conv3-512 conv3-512	conv3-512 conv3-512 **conv1-512**	conv3-512 conv3-512 **conv3-512**	conv3-512 conv3-512 conv3-512 **conv3-512**
maxpool					
conv3-512 conv3-512	conv3-512 conv3-512	conv3-512 conv3-512	conv3-512 conv3-512 **conv1-512**	conv3-512 conv3-512 **conv3-512**	conv3-512 conv3-512 conv3-512 **conv3-512**
maxpool					
FC-4096					
FC-4096					
FC-1000					
soft-max					

3.5 AlexNet and VGG Net implementation using Tensorflow

What is Tensorflow?

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. It is a free and open-source software library for machine learning and artificial intelligence. It can be used across a range of tasks but has a particular focus on training and inference of deep neural networks.

To further understand models of Convolutional Neural Networks and to learn about Tensorflow, we implemented AlexNet and VGG Net in tensorflow.

Both the networks were trained on the CIFAR-10 dataset. The models were originally trained on a subset of the ImageNet dataset but training the model on our GPU with that dataset would have taken too long. Since we only intended to understand the structure of the models and to learn Tensorflow, we chose to go with a smaller dataset.

1. Import all the required libraries including Tensorflow.
2. Load the dataset. We used Keras' functions to import CIFAR-10
3. Build the model using keras sequential api.
4. Add the layers according to the model to be trained. We also need to add the activation functions, Batch Normalization function, Max Pooling, etc. Tensorflow makes it really easy to add layers.
5. The last layer uses the softmax activation function to classify images into 10 classes.
6. Buil dthe model.
7. Compile the model with appropriate loss function, optimizer, etc
8. Train the model using model.fit() function.

For each model, different configurations of layers was used.

After training, we were able to achieve 97.6% train accuracy and 82.27% test accuracy on the AlexNet architecture. On the VGG architecture, we achieved 97.42% train accuracy and 75.37% test accuracy.

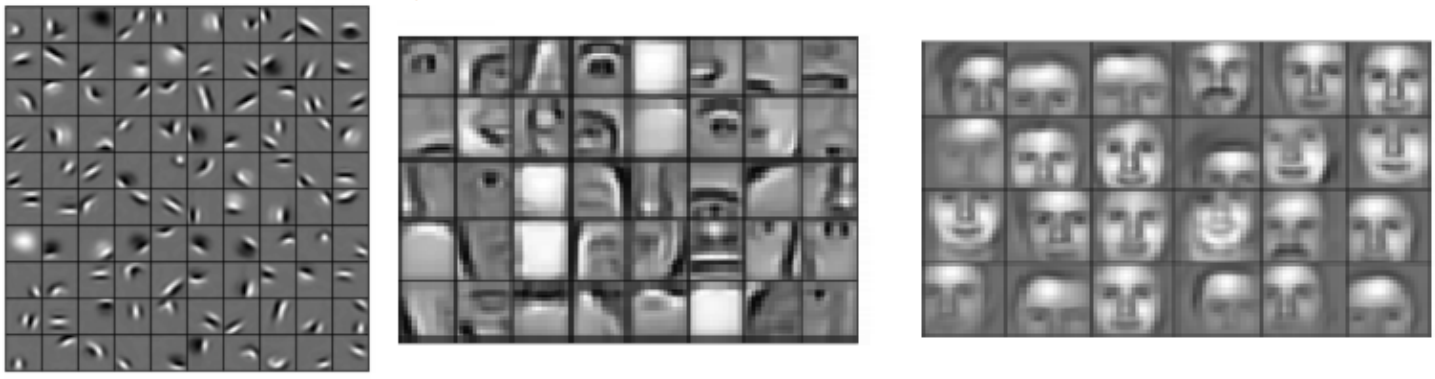
It is important to note that these models faced overfitting due to the size and complexity of the models compared to the size of our data.

4. Neural Style Transfer

Neural style transfer is an optimization technique used to take two images—a content image and a style reference image (such as an artwork by a famous painter)—and blend them together so the output image looks like the content image, but “painted” in the style of the style reference image.This is implemented by optimizing the output image to match the content statistics of the content image and the style statistics of the style reference image. These statistics are extracted from the images using a convolutional network.

The principle of neural style transfer is to define two distance functions, one that describes how different the content of two images are, Lcontent, and one that describes the difference between the two images in terms of their style, Lstyle. Then, given three images, a desired style image, a desired content image, and the input image (initialized with the content image), we try to transform the input image to minimize the content distance with the content image and its style distance with the style image.

Look at the image below carefully and examine the feature maps at different layers.



What you must have noticed is that- the maps in the lower layers look for low level features such as lines or blobs (gabor filters). As we go to the higher layers, out features become increasingly complex. Intuitively we can think of it this way- the lower layers capture low level features such as lines and blobs, the layer above that builds up on these low level features and calculates slightly more complex features, and so on…

Thus, we can conclude that ConvNets develop a hierarchical representation of features.

This property is the basis of style transfer.

While doing style transfer, we are not training a neural network. Rather, what we're doing is — we start from a blank image composed of random pixel values, and we optimize a cost function by changing the pixel values of the image. In simple terms, we start with a blank canvas and a cost function. Then we iteratively modify each pixel so as to minimize our cost function. To put it in another way, while training neural networks we update our weights and biases, but in style transfer, we keep the weights and biases constant, and instead, update our image.

For doing this, it is important that our cost function correctly represents the problem. The cost function has two terms- a style loss term and a content loss term, both of which are explained below.

Content Loss

$$\mathcal{L}_{content}(\vec{p}, \vec{x}, l) = \frac{1}{2} \sum_{i,j} \left(F_{ij}^{l} - P_{ij}^{l} \right)^2$$

This is based on the intuition that images with similar content will have similar representation in the higher layers of the network.

P^l is the representation of the original image and F^l is the representation of the generated image in the feature maps of layer l.

Style Loss

$$E_l = \frac{1}{4N_l^2 M_l^2} \sum_{i,j} \left(G_{ij}^{l} - A_{ij}^{l} \right)^2$$

$$\mathcal{L}_{style}(\vec{a}, \vec{x}) = \sum_{l=0}^{L} w_l E_l$$

Here, A^l is the representation of the original image and G^l is the representation of the generated image in layer l. Nl is the number of feature maps and Ml is the size of the flattened feature map in layer l. wl is the weight given to the style loss of layer l.

By style, we basically mean to capture brush strokes and patterns. So we mainly use the lower layers, which capture low level features. Also note the use of gram matrix here. Gram matrix of a vector X is X.X_transpose. The intuition behind using gram matrix is that we're trying to capture the statistics of the lower layers.

You don't have to necessarily use a Gram matrix, though. Some other statistics (such as mean) have been tried and have worked pretty well too.

Total Loss

$$\mathcal{L}_{total}(\vec{p}, \vec{a}, \vec{x}) = \alpha \mathcal{L}_{content}(\vec{p}, \vec{x}) + \beta \mathcal{L}_{style}(\vec{a}, \vec{x})$$

where alpha and beta are weights for content and style, respectively. They can be tweaked to alter our final result. So our total loss function basically represents our problem- we need the content of the final image to be similar to the content of the content image and the style of the final image should be similar to the style of the style image.

Now all we have to do is to minimize this loss. We minimize this loss by changing the input to the network itself. We basically start with a blank grey canvas and start altering the pixel values so as to minimize the loss. Any optimizer can be used to minimize this loss.

4.1 Implementation

Understanding VGG Net was even more important as we used it to implement Neural Style Transfer. We used a pre-trained VGG model. This implementation was done in Tensorflow.

For Neural Style Transfer to work, we need a trained neural network. For this implementation, we imported a trained VGG-19 network using Tensorflow.

Libraries used - **Numpy, Tensorflow, Keras, IPython.display**

First we defined the path for the input images (Style image and Content Image) and the final generated image using **keras.utils.get_file**.

We defined some parameters including **style weight** and **content weight** which allows us to control how stylized we want the image ot be.

We defined a function **preprocess_image()** to preprocess the input image to convert to tensor datatype.

We defined a function **deprocess_image()** to convert a tensor back to an image.

A function called **gram_matrix()** allows us to create a matrix that helps us quantify the styles in an image. The gram matrix of the output image and the input style image will be compared.

style _loss(), **content_loss()** and **total_variation_loss()** were defined to calculate loss.

A dictionary containing all the layers we want the output is required. The layers are extracted using **keras.Model()**.

compute_loss() is used to finally calculate loss by adding style loss, content loss and total variation loss.

compute_loss_and_grads() is udes to calculate the gradient allowing us to backpropagate the output image which is the input to the neural network.

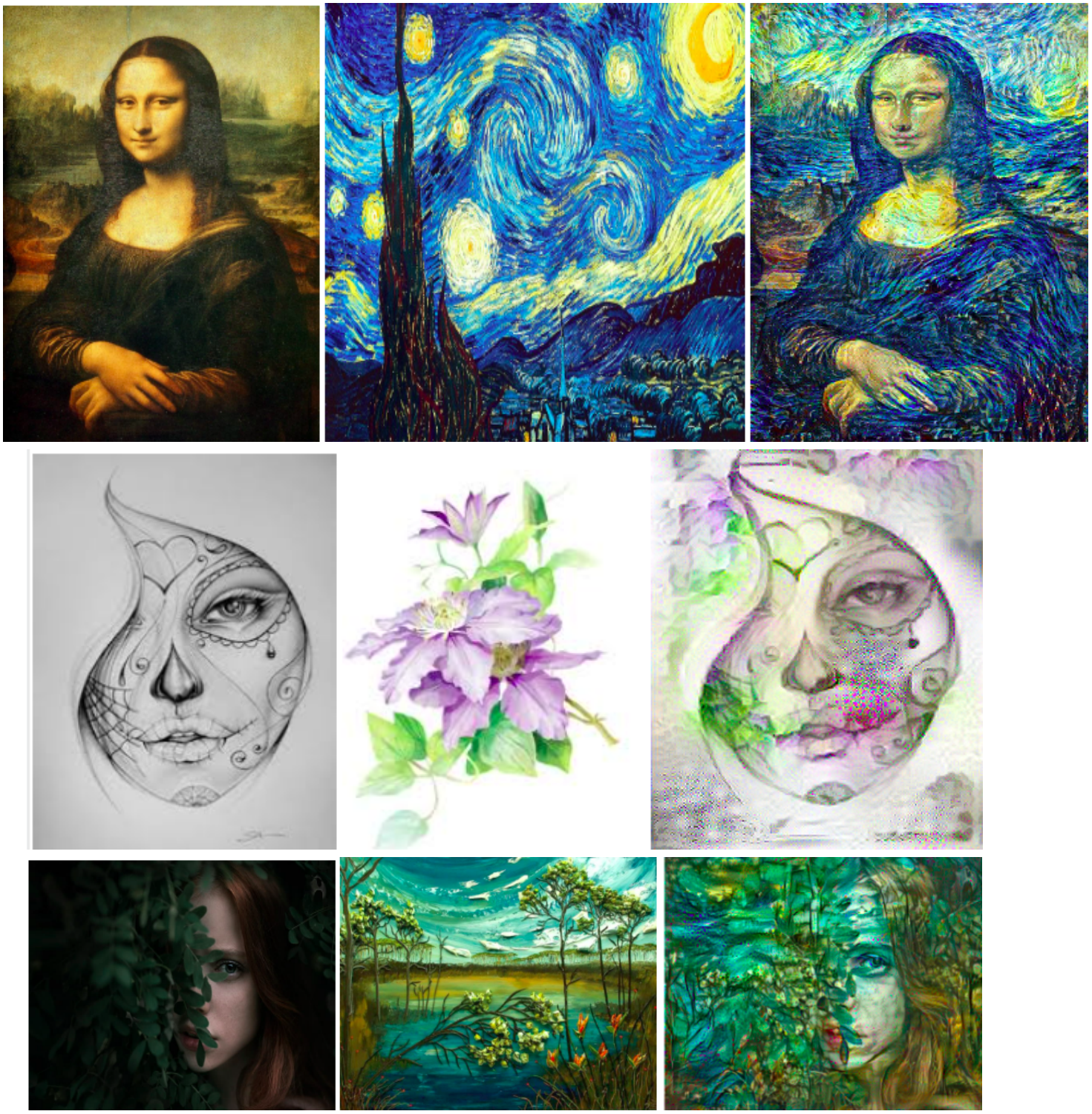
Finally, the image is iterated multiple times and it slowly gets closer to both the style and content image.

In general, we found that anywhere between 2000-4000 iterations were enough and any further iterations either produced extremely small changes or none at all.

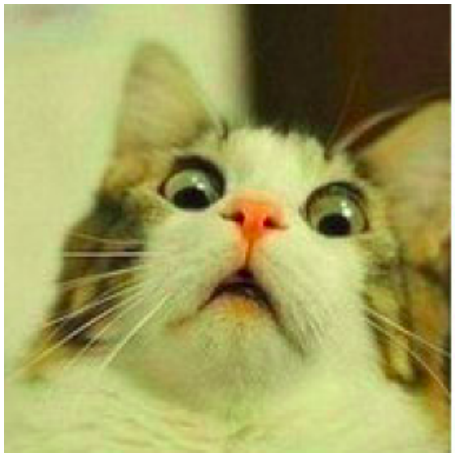
Also, it is important to note that the input style and content images severely affected the quality of the output image. We found that some style images were better suited to create pleasing images almost each time.

Content image with a clear and properly framed subject and style image with repetitive patterns were better suited to neural style transfer.

4.2 NST Result




www.atomiccircus-tattoo.com

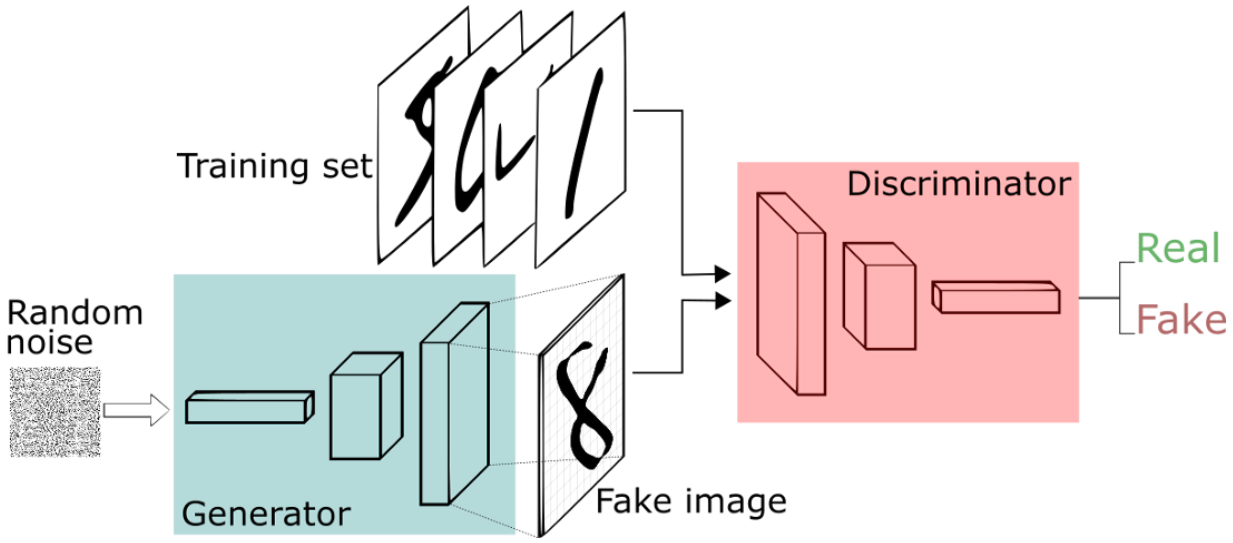



5. GANs and CycleGANs

5.1 GANs

Generative Adversarial Networks takes up a game-theoretic approach, unlike a conventional neural network. The network learns to generate from a training distribution through a 2-player game. The two entities are Generator and Discriminator. These two adversaries are in constant battle throughout the training process. Since an adversarial learning method is adopted, we need not care about approximating intractable density functions.

As you can identify from their names, a generator is used to generate real-looking images and the discriminator's job is to identify which one is a fake. The entities/adversaries are in constant battle as one(generator) tries to fool the other(discriminator), while the other tries not to be fooled. To generate the best images you will need a very good generator and a discriminator. This is because if your generator is not good enough, it will never be able to fool the discriminator and the model will never converge. If the discriminator is bad, then images which make no sense will also be classified as real and hence your model never trains and in turn you never produces the desired output. The input, random noise can be a Gaussian distribution and values can be sampled from this distribution and fed into the generator network and an image is generated. This generated image is compared with a real image by the discriminator and it tries to identify if the given image is fake or real.



Since a game-theoretic approach is taken, our objective function is represented as a minimax function. The discriminator tries to maximize the objective function, therefore we can perform gradient ascent on the objective function. The generator tries to minimize the objective function, therefore we can perform gradient descent on the objective function. By alternating between gradient ascent and descent, the network can be trained.

$$\min_{\theta_g} \max_{\theta_d} \left[\mathbb{E}_{x \sim p_{data}} \log \underbrace{D_{\theta_d}(x)}_{\text{Discriminator output for real data x}} + \mathbb{E}_{z \sim p(z)} \log(1 - \underbrace{D_{\theta_d}(G_{\theta_g}(z))}_{\text{Discriminator output for generated fake data G(z)}}) \right]$$

But when applied, it is observed that optimizing the generator objective function does not work so well, this is because when the sample is generated is likely to be classified as fake, the model would like to learn from the gradients but the gradients turn out to be relatively flat. This makes it difficult for the model to learn. Therefore, the generator objective function is changed as below.

$$\max_{\theta_g} \mathbb{E}_{z \sim p(z)} \log(D_{\theta_d}(G_{\theta_g}(z)))$$

Instead of minimizing the likelihood of discriminator being correct, we maximize the likelihood of discriminator being wrong. Therefore, we perform gradient ascent on generator according to this objective function.

5.2 GAN Implementation - Face Generation

For Face Generation to work, we need a dataset of images of faces. We used **facemask_lite/wihtoutmask** dataset and trained out GAN using the unmasked dataset which contains 9896 images.

Libraries used - **Numpy, Tensorflow, Keras, IPython.display, OpenCV, os, re, tqdm, glob, imageio, PIL**

First we processed ther images to resize it to 256x256. We made an array of all images using **keras.preprocessing.image.img_to_array()**

We defined a function **Generator()** to create a model for our generator. We used **Keras Sequential API** to define our model. Our generator model is inspired from the **U-Net** model and has multiple **convolution** and **transpose convolution** layers. The input to our generator model is randomly generated data and the size chosen for this input is **100.** We used **ReLU** as our activation function throughout.

Next, we defined a function **Discriminator()** to create a model for our discriminator. Again, we used **Keras Sequential API** to define our model. Our discriminator is a simpler model having **6 convolutional layers** and **one Dense layer.** The input to the discriminator are images from the real dataset and the fake dataset coming from the generator. All the input images are of the 256x256.

The randomized input for the generator was created using the numpy function **np.random.normal()**.

We used the **RMSprop** optimizer and the loss function used was **Binary Cross Entropy**.

SInce training takes a lot of time, it is important to save checkpoints incase training is ever interrupted. We saved checkpoints every 2 epochs and every last checkpoint was saved too.

Functions for calculation of loss were defined - **generator_loss()** and **discriminator_loss()**

A function **train_steps()** was defined to train the generator and the discriminator.

plot_generated_images() is simply used to plot and print the images of the generator.

A function **generate_and_save_images()** was created that saved the images so they can be accessed later.

Finally, a function **train()** uses the above functions to train the generator and the discriminator such that the generator slowly learns to generate more realistic faces.

For training, we went with a **batch size of 32.**

We trained the GAN for 28 epochs which took around 8 hours on DGX A100.

5.3 Face Generation Result

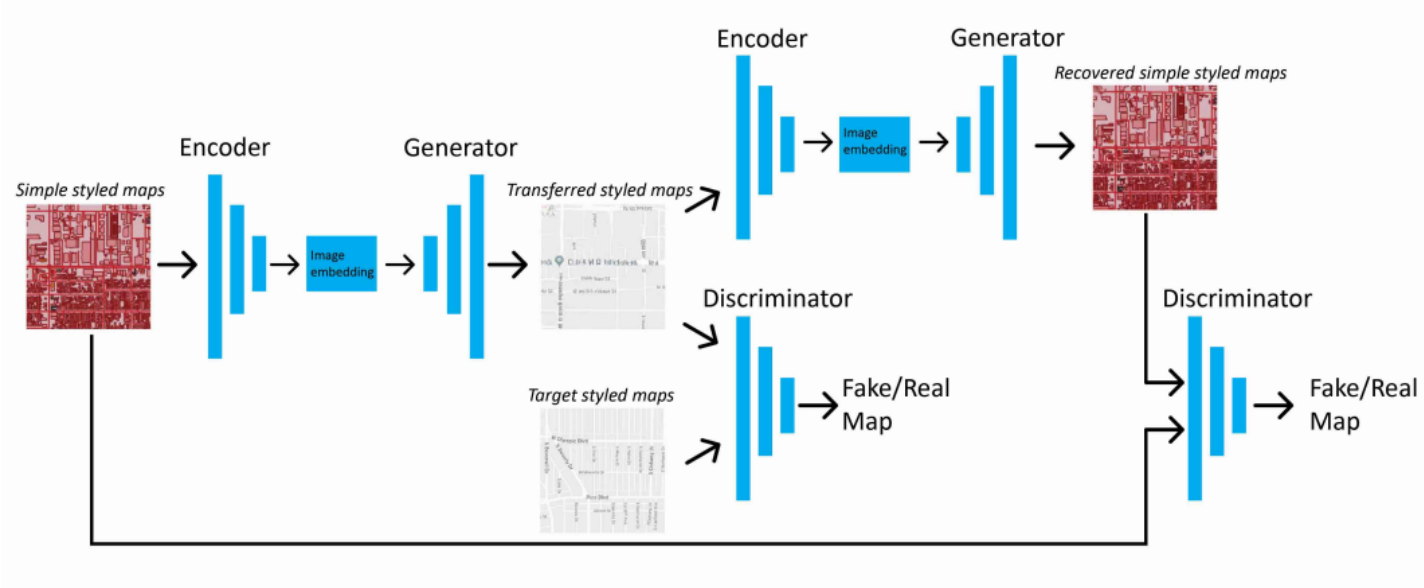
5.3 CycleGANs

The Cycle Generative Adversarial Network, or CycleGAN, is an approach to training a deep convolutional neural network for image-to-image translation tasks. The Network learns mapping between input and output images using unpaired dataset. For Example: Generating RGB imagery from SAR, multispectral imagery from RGB, map routes from satellite imagery, etc.

The model architecture is comprised of two generator models: one generator (Generator-A) for generating images for the first domain (Domain-A) and the second generator (Generator-B) for generating images for the second domain (Domain-B).

Domain-B -> Generator-A -> Domain-A

Domain-A -> Generator-B -> Domain-B



Each generator has a corresponding discriminator model (Discriminator-A and Discriminator-B). The discriminator model takes real images from Domain and generated images from Generator to predict whether they are real or fake.

Domain-A -> Discriminator-A -> [Real/Fake]

Domain-B -> Generator-A -> Discriminator-A -> [Real/Fake]

Domain-B -> Discriminator-B -> [Real/Fake]

Domain-A -> Generator-B -> Discriminator-B -> [Real/Fake]

How is loss calculated?

The loss used to train the Generators consists of three parts:

1. **Adversarial Loss**: We apply Adversarial Loss to both the Generators, where the Generator tries to generate the images of it's domain, while its corresponding discriminator distinguishes between the translated samples and real samples. Generator aims to minimize this loss against its corresponding Discriminator that tries to maximize it.

2. **Cycle Consistency Loss**: It captures the intuition that if we translate the image from one domain to the other and back again we should arrive at where we started. Hence, it calculates the L1 loss between the original image and the final generated image, which should look same as original image. It is calculated in two directions:

Forward Cycle Consistency: Domain-B -> Generator-A -> Domain-A -> Generator-B -> Domain-B

Backward Cycle Consistency: Domain-A -> Generator-B -> Domain-B -> Generator-A -> Domain-A

3. **Identity Loss**: It encourages the generator to preserve the color composition between input and output. This is done by providing the generator an image of its target domain as an input and calculating the L1 loss between input and the generated images.

Domain-A -> Generator-A -> Domain-A

Domain-B -> Generator-B -> Domain-B

As all of these loss functions play critical roles in arriving at high-quality results. Hence, both the generator models are optimized via combination of all of these loss functions.

5.4 CycleGANs Implementation - Style Transfer

For Face Generation to work, we need a dataset of images of zebras and horses. We used tensorflow dataset to import a dataset call **cycle_gan/horse2zebra** It has 2 classes of images - zebras and horses.

Libraries used - **Tensorflow, tensorflow_addons, Keras, tensorflow_datasets**

First we loaded the dataset using **tfds.load()**. The dataset has test and train images for both zebras and horses.

We defined some parameters - **BUFFER_SIZE = 1000, BATCH_SIZE = 1, IMG_WIDTH = 256, IMG_HEIGHT = 256**

Since the images from the dataset have different sizes, we randomly cropped a 256x256 area from each image using **random_crop()**.

A function **random_jitter()** resized all images to 286x286 from which a 256x256 sections was cropped using **random_crop()**. The image was also randomly flipped. This function allows us to do data augmentation which expands our dataset.

Functions **preprocess_image_train()** and **preprocess_image_test()** apply data augmentation to the dataset using **random_jitter()**.

define_discriminator() was defined to create a model for our discriminator. We used **Keras Sequential API** to define our model. Our discriminator is a simpler model having **6 convolutional layers.** The model was compile using the **Adam Optimizer.**

Since we need 2 discriminators, we created **disc_x** and **disc_y** using the above function.

define_generator() was defined to create a model for our generator. We used **Keras Sequential API** to define our model. Our generator model has **4 convolutional layers** and **2 transpose convolutional layers**. We used **Batch Normalization** after every layer which differs from

Since we need 2 generators, we created **gen_g** and **gen_f** using the above function.

gen_g is used to convert horse to zebra and **gen_f** is used to convert zebra to horse.

The loss function used is **Binary Cross Entropy.**

discriminator_loss() calculates loss for the discriminator by adding loss for real images and loss for generated images.

generator_loss() calculates how many times the generated image fooled the discriminator.

calc_cycle_loss() calculates the difference between the input image and the cycled image

identity_loss() calculates the identity loss for both the generators and discriminators.

For all the discriminators and generators, we used **Adam Optimizers**.

Since training takes a lot of time, it is important to save checkpoints in case training is ever interrupted. We saved checkpoints every 2 epochs and every last checkpoint was saved too.

train_step() was created which allows us to train the generators and discriminators.

Finally, the generators and discriminators were trained.

The training was trained for 128 epochs on DGX A100 which took around 24 hours.

5.5 Style Transfer Result














Major Problems Faced

- Alpha/beta ratio during initial NST implementation
- Noise in generated images
- Out of memory while loading VGG
- Tensorflow and Cuda
- Cuda
- Cuda
- Training a GAN model
- Colab disconnecting runtime
- Getting permission to use DGX A100
- Memory leak in Tensorflow

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.

REFERENCES

1) LA Playlist:
https://www.youtube.com/playlist?list=PLZHQObOWTQDPD3MizzM2xVFitgF8hE_ab

2) 3B1B Neural network playlist:
https://www.youtube.com/playlist?list=PLZHQObOWTQDNU6R1_67000Dx_ZCJB-3pi

3) Coursera course(Deep learning-All 5):-
https://www.coursera.org/specializations/deep-learning

4) Coursera course youtube playlist:
https://youtube.com/playlist?list=PLpFsSf5Dm-pd5d3rjNtlXUHT-v7bdaEIe

5) Neural Style Transfer Paper
https://arxiv.org/abs/1508.06576

6) Photorealistic Neural Style Transfer Paper
https://arxiv.org/abs/1912.02398

7) GANs paper
https://arxiv.org/abs/1406.2661

8) Article on GANs:
https://towardsdatascience.com/generative-adversarial-networks-gans-8fc303ad5fA1
https://medium.com/analytics-vidhya/gans-a-brief-introduction-to-generative-adversarial-networks-f06216c7200e
https://medium.com/nerd-for-tech/face-generation-using-generative-adversarial-networks-gan-6d279c2d5759

9) GANs playlist:
https://www.youtube.com/playlist?list=PLdxQ7SoCLQAMGgQAIAcyRevM8VvygTpCu

10) Intuition of GANs:

https://youtu.be/Sw9r8CL98N0

11) Face Generation Using GANs implementation :
https://thispersondoesnotexist.com
https://github.com/topics/thispersondoesnotexist

12) CycleGANs paper :
https://arxiv.org/abs/1703.10593

13)CycleGANs articles :
https://jonathan-hui.medium.com/gan-cyclegan-6a50e7600d7
https://medium.com/analytics-vidhya/the-beauty-of-cyclegan-c51c153493b8
https://medium.datadriveninvestor.com/style-transferring-of-image-using-cyclegan-3cc7aff4fe61

14) CycleGANs implementation by authors :
https://github.com/junyanz/pytorch-CycleGAN-and-pix2pix

15) Tensorflow :
https://www.tensorflow.org/learn