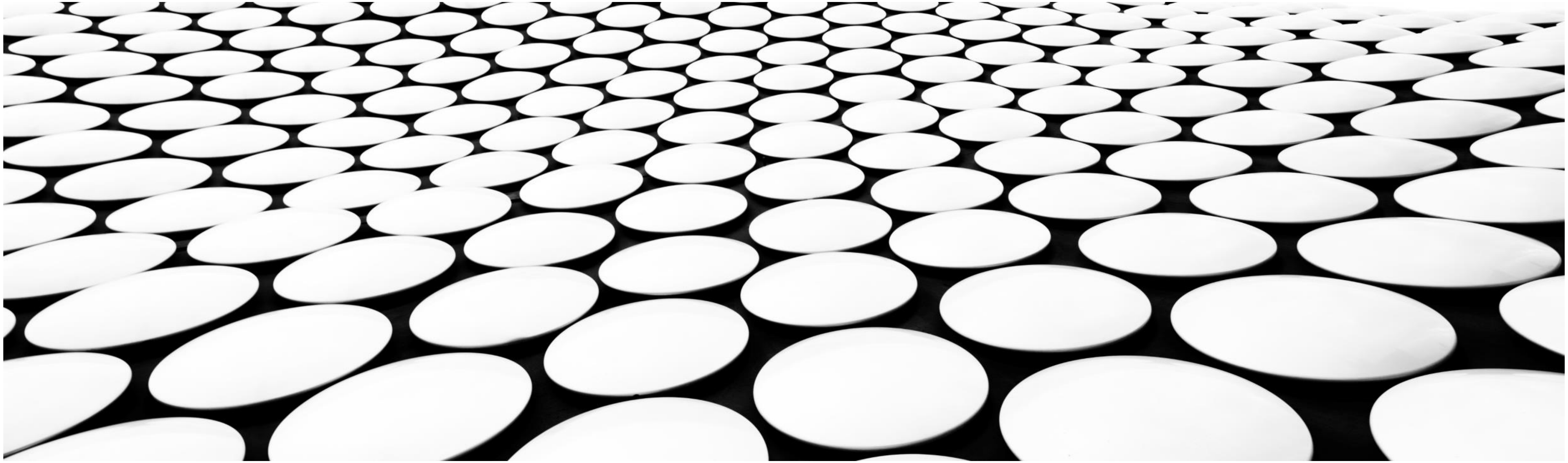
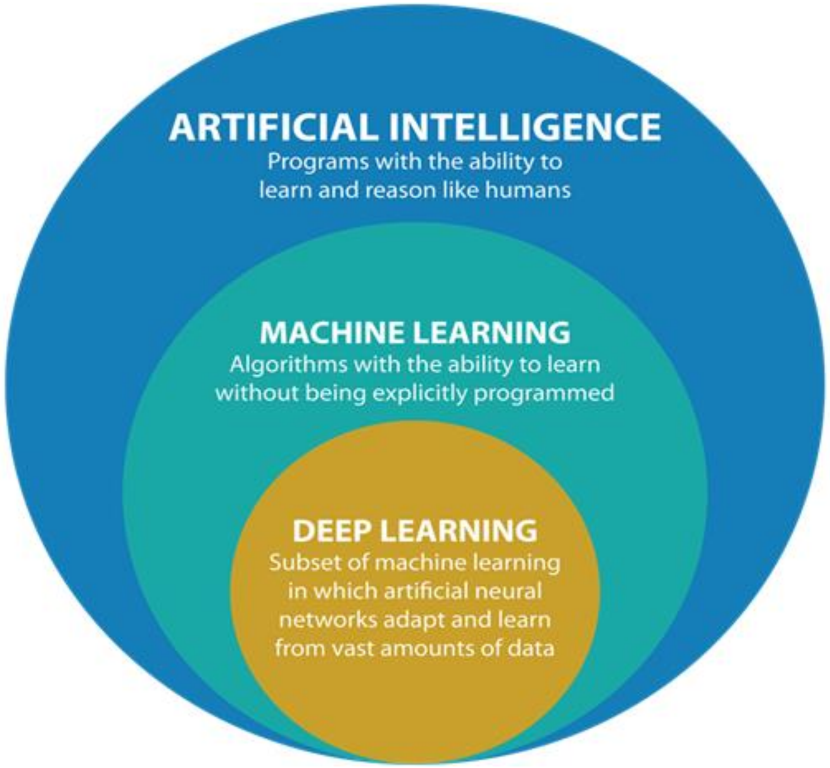
NEURAL NETWORKS

DEBAPRIYA HAZRA



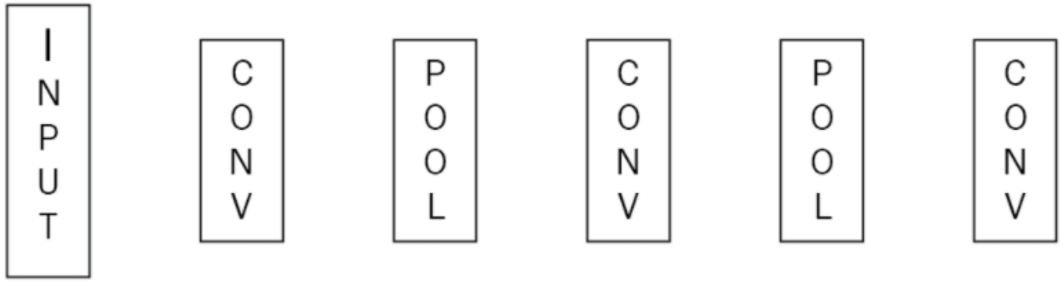
HISTORY OF ARTIFICIAL INTELLIGENCE (AI)

- Dartmouth Conference of 1956 - AI and its mission were defined.
- 1970s – First AI winter
- The first robot was constructed in 1972.
- 1990s - Second AI winter machine learning
- 1997 - IBM’s Deep Blue



CONVOLUTIONAL NEURAL NETWORKS (CNN)

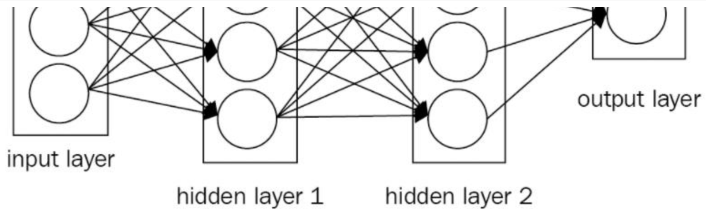
How a simple ConvNet looks like



Fully Connected Layer

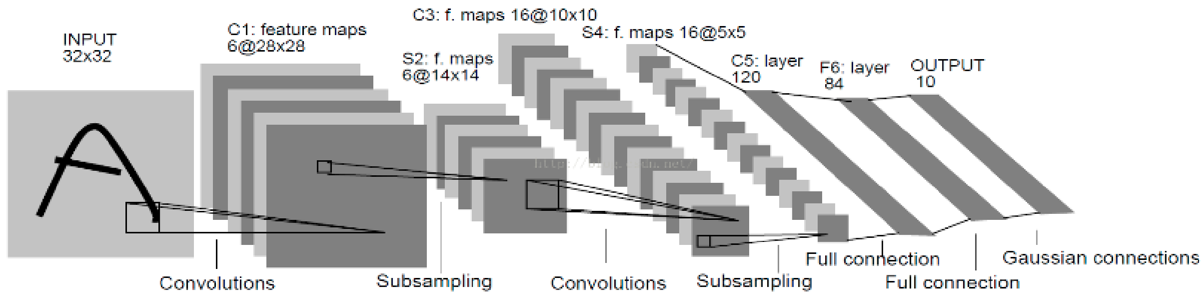
- … a three dimensional matrix
- Logistic is used for binary classification and Softmax is for multi-class classification.
- It reduces the number of parameters to learn and the amount of computation performed in the network.

- `filters` : The number of filters.
- `kernel_size` : A number specifying both the height and width of the (square) convolution window. There are also some additional optional arguments that you might like to tune.
- `strides` : The stride of the convolution. If you don't specify anything, this is set to one.
- `padding` : This is either `valid` or `same` . If you don't specify anything, the padding is set to `valid` .
- `activation` : This is typically `relu` . If you don't specify anything, no activation is applied. You are strongly encouraged to add a ReLU activation function to every convolutional layer in your networks.



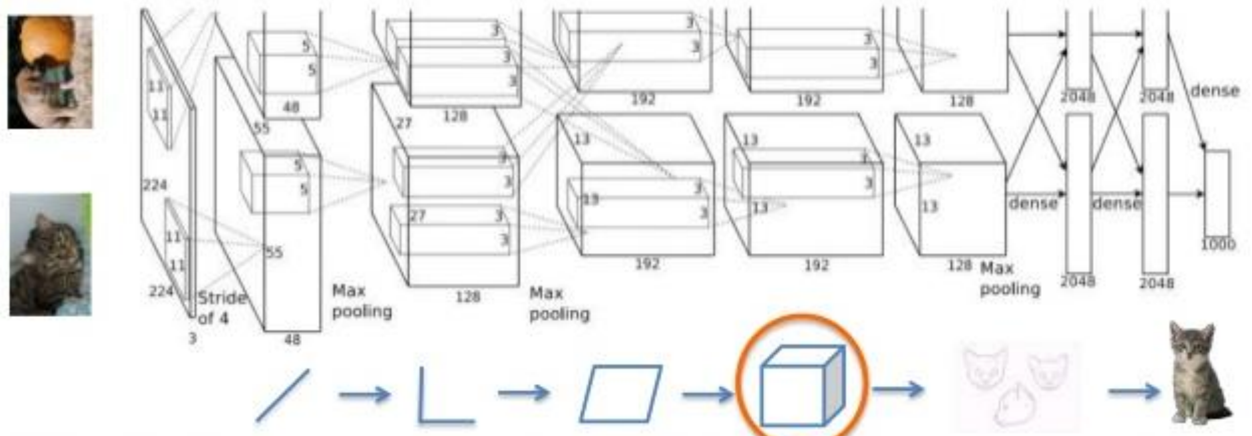
LENET (2010)

- This network takes a grayscale 32 x 32 image as input
- It goes to the convolution layers (C1) and then to the subsampling layer (S2)
- Nowadays the subsampling layer is replaced by a pooling layer.
- Then, there is another sequence of convolution layers (C3) followed by a pooling (that is, subsampling) layer (S4)
- Finally, there are three fully connected layers, including the OUTPUT layer at the end
- This network was used for zip code recognition in post offices



ALEXNET (2012)

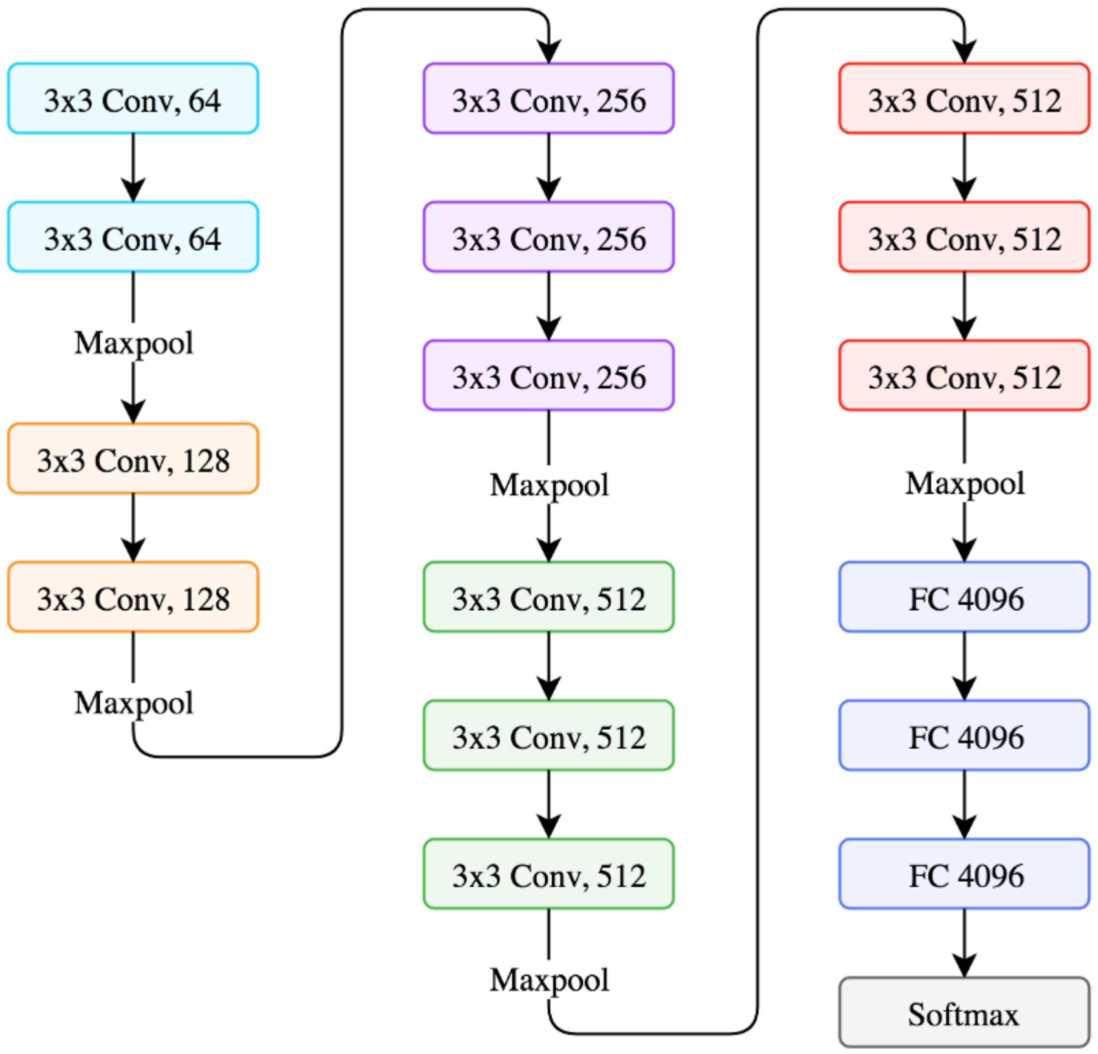
- A ReLU activation function and a dropout of 0.5 were used in this network to fight overfitting
- There is a normalization layer used in the architecture, but this is not used in practice anymore as it used heavy data augmentation
- Simple structure
- AlexNet is trained on the ImageNet database using two separate GPUs, possibly due to processing limitations with inter-GPU connections at the time



When AlexNet is processing an image, this is what is happening at each layer.

VGGNET (2014)

- The default input size of an image for this model is 224 x 224 x 3.
- The image is passed through a stack of convolution layers with a stride of 1 pixel and padding of 1.
- It uses 3 x 3 convolution throughout the network.
- Max pooling is done over a 2 x 2 pixel window with a stride of 2
- The first two fully connected layers have 4,096 neurons each
- The third fully connected layers are responsible for classification with 1,000 neurons
- All hidden layers are built with the ReLU activation function



VGGNET

- It is painfully slow to train.
- The network architecture weights themselves are quite large (in terms of disk/bandwidth).

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					

VGGNET

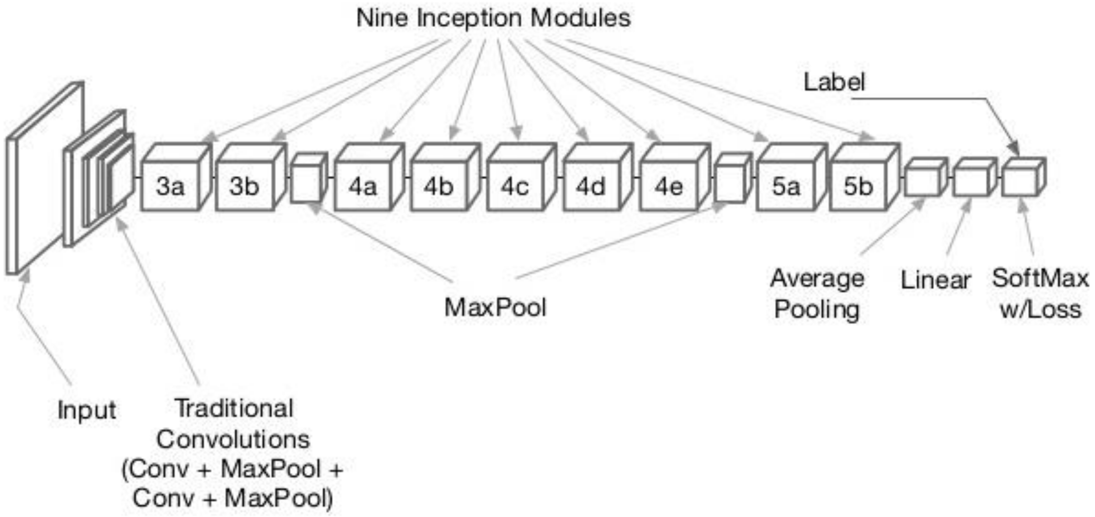
Keras

- The Keras Applications module has pre-trained neural network models, along with its pre-trained weights trained on ImageNet.
- These models can be used directly for prediction, feature extraction, and fine-tuning.
- The first time it executes the preceding script, Keras will automatically download and cache the architecture weights to disk in the ~/.keras/models directory
- From next time it will be faster.

```
#import VGG16 network model and other necessary libraries
from keras.applications.vgg16 import VGG16
from keras.preprocessing import image
from keras.applications.vgg16 import preprocess_input
import numpy as np
#Instantiate VGG16 and returns a vgg16 model instance
vgg16_model = VGG16(weights='imagenet', include_top=False)
#include_top: whether to include the 3 fully-connected layers at the top of the network.
#This has to be True for classification and False for feature extraction. Returns a model
#instance weights:'imagenet' means model is pre-training on ImageNet data.
model = VGG16(weights='imagenet', include_top=True)
model.summary()
#image file name to classify
image_path = 'jumping_dolphin.jpg'
#load the input image with keras helper utilities and resize the image.
#Default input size for this model is 224x224 pixels.
img = image.load_img(image_path, target_size=(224, 224))
#convert PIL (Python Image Library??) image to numpy array
x = image.img_to_array(img)
print (x.shape)
#image is now represented by a NumPy array of shape (224, 224, 3),
# but we need to expand the dimensions to be (1, 224, 224, 3) so we can
# pass it through the network -- we'll also preprocess the image by
# subtracting the mean RGB pixel intensity from the ImageNet dataset
#Finally, we can load our Keras network and classify the image:
x = np.expand_dims(x, axis=0)
print (x.shape)
preprocessed_image = preprocess_input(x)
preds = model.predict(preprocessed_image)
print('Prediction:', decode_predictions(preds, top=2)[0])
```

GOOGLENET (2014)

- The main attractive feature of GoogLeNet is that it runs very fast due to the introduction of a new concept called inception module.
- GoogLeNet has 22 layers
- The main advantage of GoogLeNet is that it is more accurate than VGG, while using much fewer parameters and less compute power.



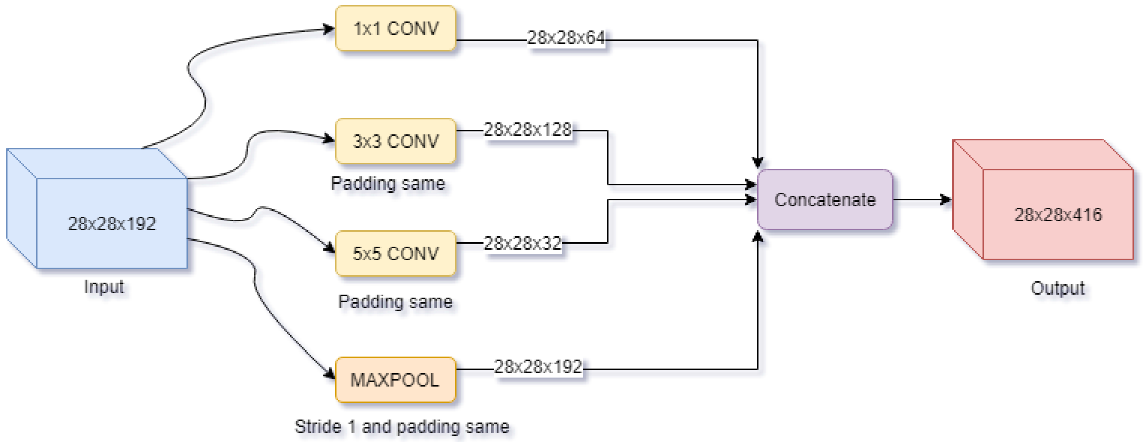
GOOGLENET

GoogleNet Architecture

- Vanishing gradient problem is avoided by adding auxiliary classifiers to intermediate layers
- Since fully connected layers are prone to overfitting, it is replaced with a global average pooling (GAP layer)
- GoogLeNet added a linear layer for the ease of transfer learning

Inception Module

- In addition to creating deeper networks, the inception block introduces the idea of parallel convolutions.
- In-parallel convolutions of different sizes are performed on the output of the previous layer.



GOOGLENET

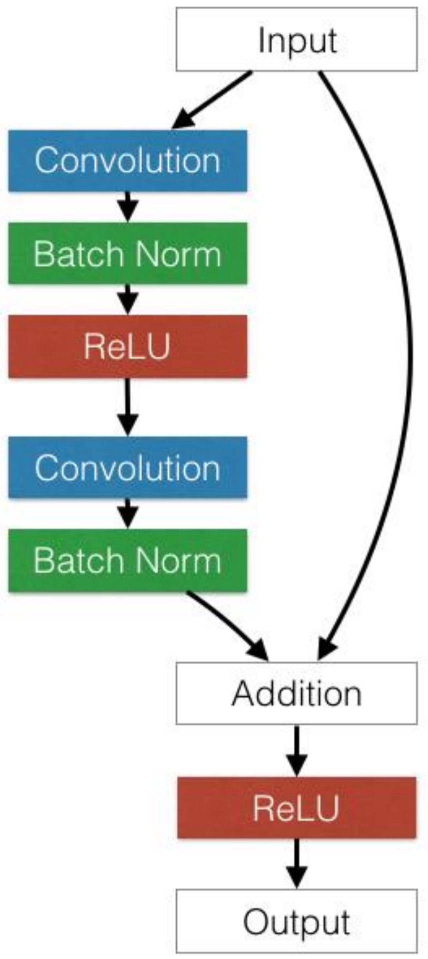
- Googlenet will be just many inception blocks in cascade
- The main disadvantage is still the gradient vanishing that would occur if we begin stacking lots and lots of inception layers
- Multiple branches and losses

Inception Block

```
import tensorflow as tf
def inception_block_a(x, name='inception_a' ):
# num of channels: 384 = 96*4
with tf.variable_scope(name):
# Pooling part
b1 = tf.layers.average_pooling2d(x, [3,3], 1, padding='SAME' )
b1 = tf.layers.conv2d(inputs=b1, filters=96, kernel_size=[1, 1], padding="same", activation=tf.nn.relu)
# 1x1 part
b2 = tf.layers.conv2d(inputs=x, filters=96, kernel_size=[1, 1], padding="same", activation=tf.nn.relu)
# 3x3 part
b3 = tf.layers.conv2d(inputs=x, filters=64, kernel_size=[1, 1], padding="same", activation=tf.nn.relu)
b3 = tf.layers.conv2d(inputs=b3, filters=96, kernel_size=[3, 3], padding="same", activation=tf.nn.relu)
# 5x5 part
b4 = tf.layers.conv2d(inputs=x, filters=64, kernel_size=[1, 1], padding="same", activation=tf.nn.relu)
# 2 3x3 in cascade with same depth is the same as 5x5 but with less parameters
# b4 = tf.layers.conv2d(inputs=b4, filters=96, kernel_size=[5, 5], padding="same", activation=tf.nn.relu)
b4 = tf.layers.conv2d(inputs=b4, filters=96, kernel_size=[3, 3], padding="same", activation=tf.nn.relu)
b4 = tf.layers.conv2d(inputs=b4, filters=96, kernel_size=[3, 3], padding="same", activation=tf.nn.relu)
concat = tf.concat([b1, b2, b3, b4], axis=-1)
return concat
```

RESIDUAL NETWORKS (2015)

- After a certain depth, adding additional layers to feed-forward convNets results in a higher training error and higher validation error.
- When adding layers, performance increases only up to a certain depth, and then it rapidly decreases.
- The ResNet team added connections that can skip layers
- The residual block, which adds the output of the previous layer to the output of the next layer



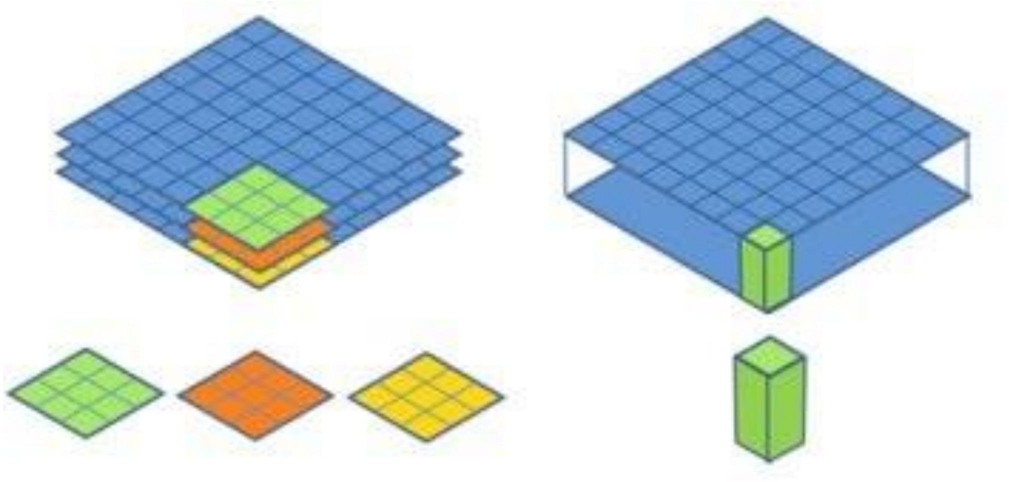
RESIDUAL NETWORKS

- It has different depth variations, such as 34, 50, 101, and 152 layers
- The ResNet architecture is a stack of residual blocks.
- Every residual block has 3 x 3 convolution layers
- After the last conv layer, a GAP layer is added.
- There is only one fully connected layer to classify 1,000 classes
- For a deeper network, say more than 50 layers, it uses the bottleneck features concept to improve efficiency.
- No dropout is used in this network.

```
import tensorflow as tf
from collections import namedtuple
# Configurations for each bottleneck group.
BottleneckGroup = namedtuple('BottleneckGroup' ,
['num_blocks', 'num_filters', 'bottleneck_size' ])
groups = [
   BottleneckGroup(3, 128, 32), BottleneckGroup(3, 256, 64), BottleneckGroup(3, 512, 128),
BottleneckGroup(3, 1024, 256)
]
# Create the bottleneck groups, each of which contains `num_blocks`
# bottleneck groups.
for group_i, group in enumerate(groups):
for block_i in range(group.num_blocks):
                name = 'group_%d/block_%d' % (group_i, block_i)
# 1x1 convolution responsible for reducing dimension
with tf.variable_scope(name + '/conv_in' ):
                conv = tf.layers.conv2d(
                 net,
filters=group.num_filters,
kernel_size=1,
padding='valid' ,
activation=tf.nn.relu)
                 conv = tf.layers.batch_normalization(conv, training=training) with
tf.variable_scope(name + '/conv_bottleneck' ):
                conv = tf.layers.conv2d(
                                conv,
filters=group.bottleneck_size,
kernel_size=3,
padding='same' ,
activation=tf.nn.relu)
                conv = tf.layers.batch_normalization(conv, training=training)
# 1x1 convolution responsible for restoring dimension
with tf.variable_scope(name + '/conv_out' ):
                input_dim = net.get_shape()[-1].value
                conv = tf.layers.conv2d(
                                conv,
filters=input_dim,
kernel_size=1,
padding='valid' ,
activation=tf.nn.relu)
                conv = tf.layers.batch_normalization(conv, training=training)
# shortcut connections that turn the network into its counterpart
# residual function (identity shortcut)
net = conv + net
```

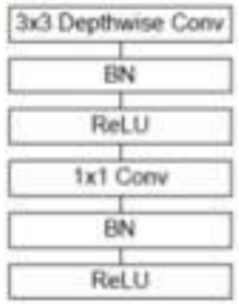
MOBILENETS (2017)

- Works faster on mobile devices.
- Created by Google, MobileNet's key feature is that it uses a different "sandwich" form of convolution block.
- Instead of the usual (CONV, BATCH_NORM,RELU), it splits 3x3 convolutions up into a 3x3 depthwise convolution, followed by a 1x1 Pointwise CONV.
- They call this block a depthwise separable convolution.



Depthwise Convolutional Filters

Pointwise Convolutional Filters



Depthwise Separable Convolution

MOBILENETS

Depthwise Separable Convolution

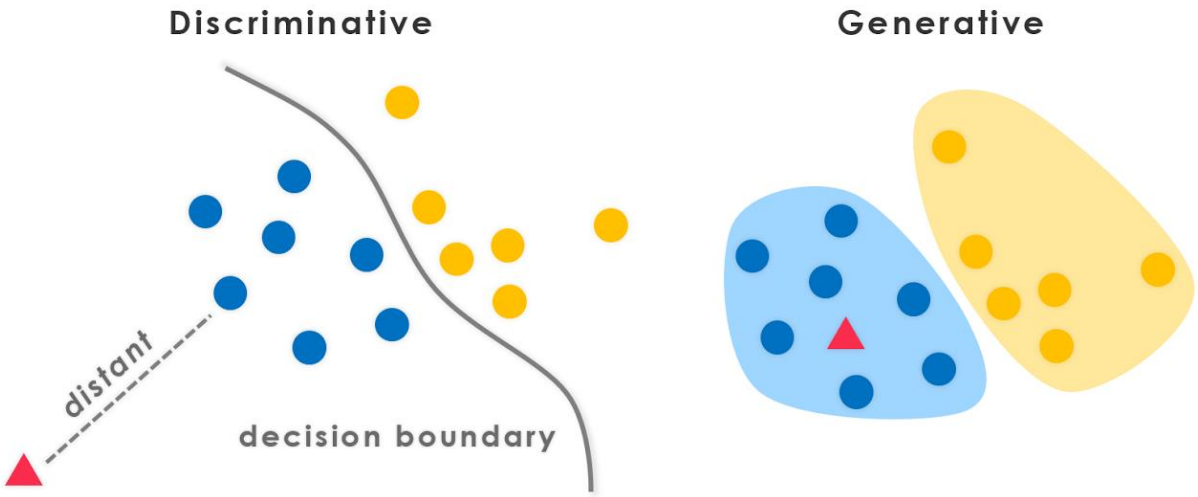
- The new convolution block (tf.layers.separable_conv2d) consists of two main parts:
  - a depthwise convolution layer
  - followed by a 1x1 pointwise convolution layer.
- However, it only filters input channels, and it does not combine them to create new features.

Control Parameters

- MobileNets uses two hyperparameters to help control the trade-off between accuracy and speed
- Width Multiplier: Controls the Depthwise CONVs accuracy by uniformly reducing the number filters used throughout the network
- Resolution Multiplier: Simply scales down the input image to different sizes

WHY GENERATIVE MODELS ?

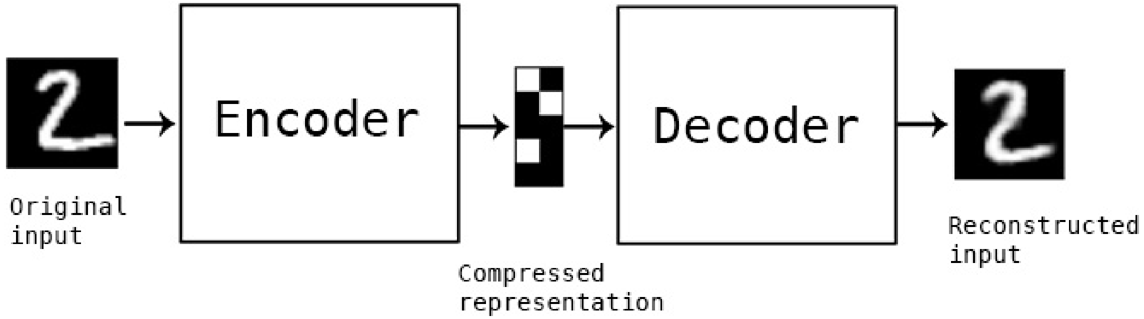
- With discriminative models, we generally try to find ways of separating or "discriminating" between different classes in our data
- However, with generative models, we try to find out the probability distribution of our data



- Pretrain a model with unlabeled data
- Augment your dataset (in theory, if you capture the probability distribution of your data, you can generate more data)
- Compress your data (lossy)
- Create some sort of simulator (for example, a quadcopter can be controlled with four inputs; if you capture this data and train a generative model on it, you can learn the dynamics of the drone)

AUTOENCODERS

- An autoencoder is a regular neural network, an unsupervised learning model that takes an input and produces the same input in the output layer.
- The idea is that the encoder part will compress your input into a smaller dimension.
- From this smaller dimension, it then tries to reconstruct the input using the decoder part of the model.
- An encoder can be a fully connected neural network or a convolutional neural network (CNN).
- A decoder also uses the same kind of network as an encoder.



AUTOENCODERS

Uses

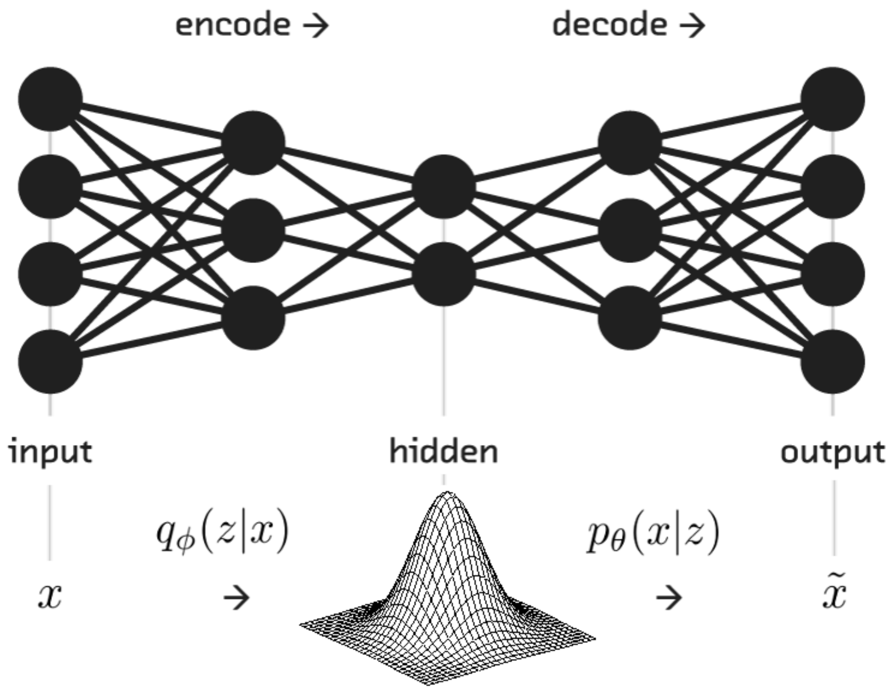
- The use of autoencoders can be good for pretraining
- Once trained, you can use the weights of the encoder and then fine-tune them to your intended task.
- Another use is as a form of compression for your data if it isn't too complicated.
- You can use the autoencoder to reduce the dimensionality down to two or three dimensions and then try to visualize your inputs in the latent space to see whether it shows you anything useful.

Limitations

- They cannot be used to generate more data for us.
- This is because we don't know how to create new latent vectors to feed to the decoder

VARIATIONAL AUTOENCODERS (VAE)

- The first generative model that could **create more data that resembles the training data**
- VAE has a new constraint
- When we want to use our VAE to generate new data, we can just **create sample vectors that come from a unit Gaussian distribution and give them to the trained decoder.**



VARIATIONAL AUTOENCODERS (VAE)

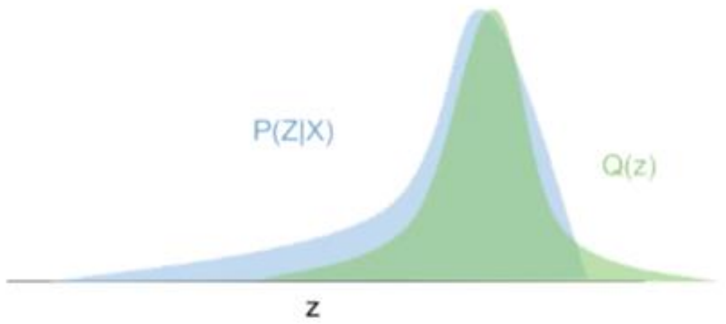
- We need two parameters to keep track and to enforce our VAE model to produce a normal distribution in the latent space:
  - Mean (should be zero)
  - Standard deviation (should be one)

VAE loss function

- Generative loss: This loss compares the model output with the model input.
- Latent loss: The loss we use here will be the KL divergence loss. This loss term penalizes the VAE if it starts to produce latent vectors that are not from the desired distribution.

```
generation_loss = mean(square(generated_image - real_image))
latent_loss = KL-Divergence(latent_variable, unit_gaussian)
loss = generation_loss + latent_loss
```

VARIATIONAL AUTOENCODERS (VAE)

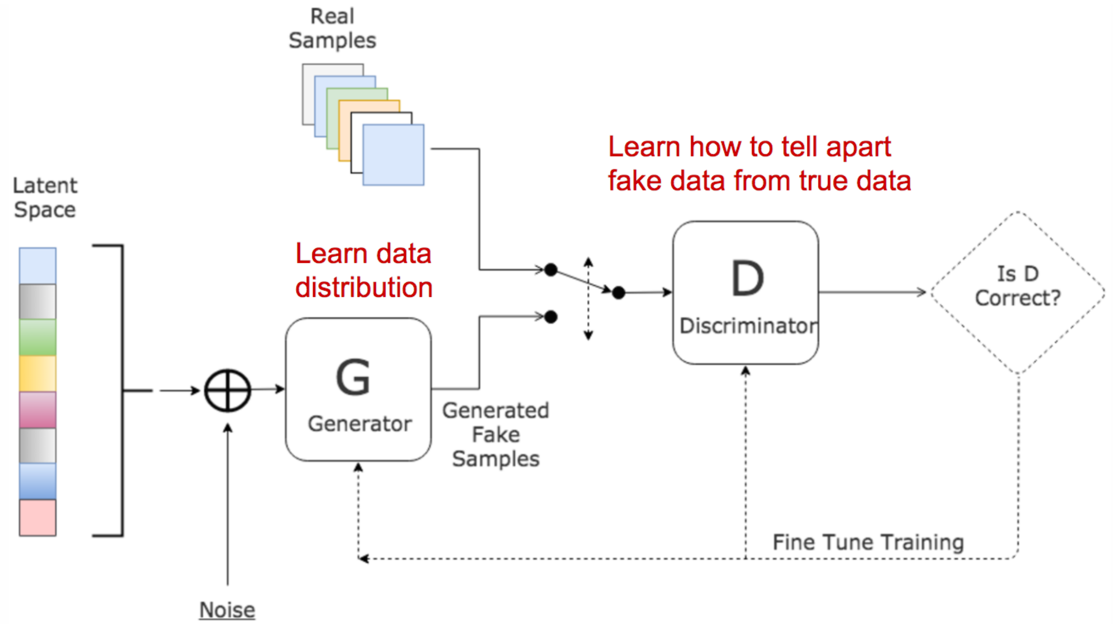


Kullback-Leibler divergence

- The blue distribution is trying to model the green distribution
- As the blue distribution comes closer and closer to the green one, the KL divergence loss will get closer to zero.

GENERATIVE ADVERSARIAL NETWORKS (GAN) (2014)

- **Generator:** Create images similar to the **real images** dataset using a size N, 1-D vector as input (Choice of N is up to us)
- **Discriminator:** Verify that the image given to it is real or a fake generated one



GENERATIVE ADVERSARIAL NETWORKS (GAN) (2014)

Practical usage of GANs

- Use the discriminator network weights as the initialization for a different task similar to what we can do with autoencoders
- Use the generator network to create new images, possibly to augment your dataset, like we can do with the trained decoder of the VAE
- Use the discriminator as a loss function (potentially, better than L1/L2 for images) and can also be used back in the VAE
- Semi-supervised learning by mixing generated data with labeled data

GENERATIVE ADVERSARIAL NETWORKS (GAN) (2014)

The Discriminator

- The discriminator takes as input a batch of 784 length vectors, which is our 28x28 MNIST images flattened.
- The output will be just a single number for each image
- We use Leaky ReLu as the activation function to prevent ReLu units from dying out.

```
def discriminator(x):
    with tf.variable_scope("discriminator"):
        fc1 = tf.layers.dense(inputs=x, units=256, activation=tf.nn.leaky_relu)
        fc2 = tf.layers.dense(inputs=fc1, units=256, activation=tf.nn.leaky_relu)
        logits = tf.layers.dense(inputs=fc2, units=1)
        return logits
```

Discriminator Loss

$$\frac{1}{m}\sum_{i=1}^{m} [log D(x^i) + log(1 - D(G(z^i)))]$$

It wants to output 1 for real image and 0 for generated images.

GENERATIVE ADVERSARIAL NETWORKS (GAN) (2014)

The Generator

- The job of the generator is to take a vector of random noise as input and from this, produce an output image
- We use the tanh activation on the output to restrict generated images to be in the range -1 to 1.

```
def generator(z):
    with tf.variable_scope("generator"):
        fc1 = tf.layers.dense(inputs=z, units=1024, activation=tf.nn.relu)
        fc2 = tf.layers.dense(inputs=fc1, units=1024, activation=tf.nn.relu)
        img = tf.layers.dense(inputs=fc2, units=784, activation=tf.nn.tanh)
        return img
```

Generator Loss

$$\frac{1}{m}\sum_{i=1}^{m} log(D(G(z^i)))$$

- **m:** Batch size
- **D:** Discriminator
- **G:** Generator
- **z:** Random noise vector

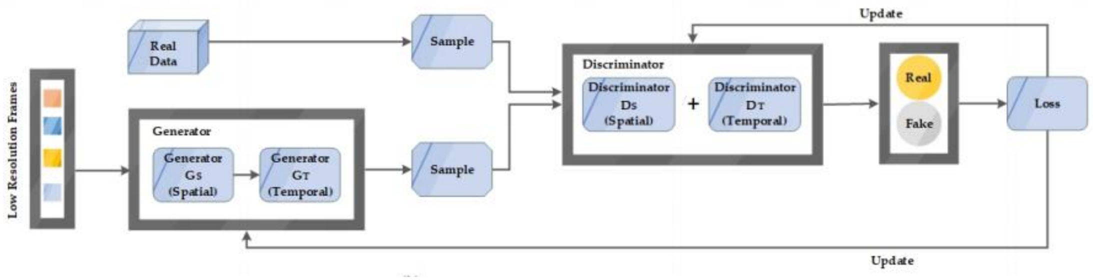
We want to maximize this loss function when training our GAN. When the loss is maximized, it means the generator is capable of generating images that can fool the discriminator, and the discriminator is outputting 1 for generated images.

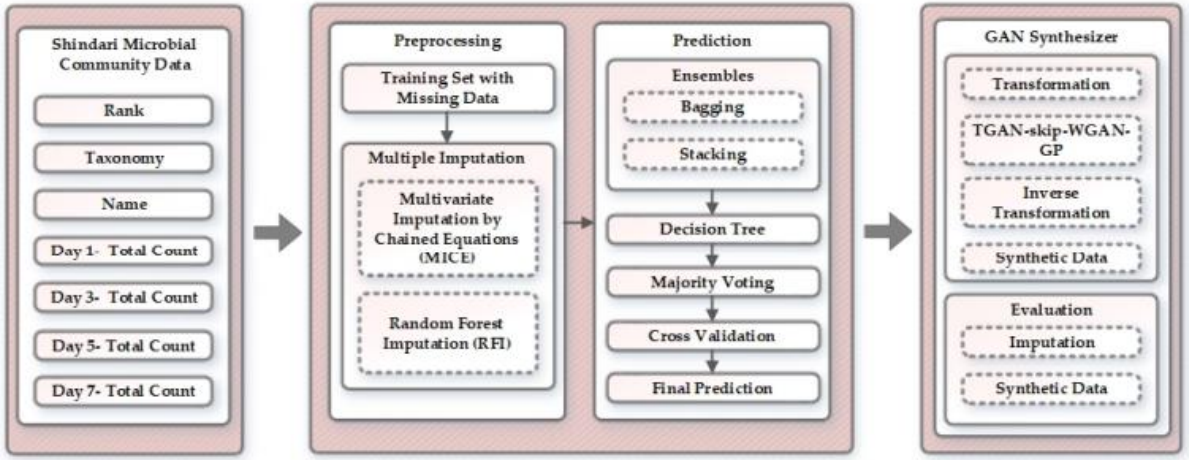
PROBLEMS WITH GAN

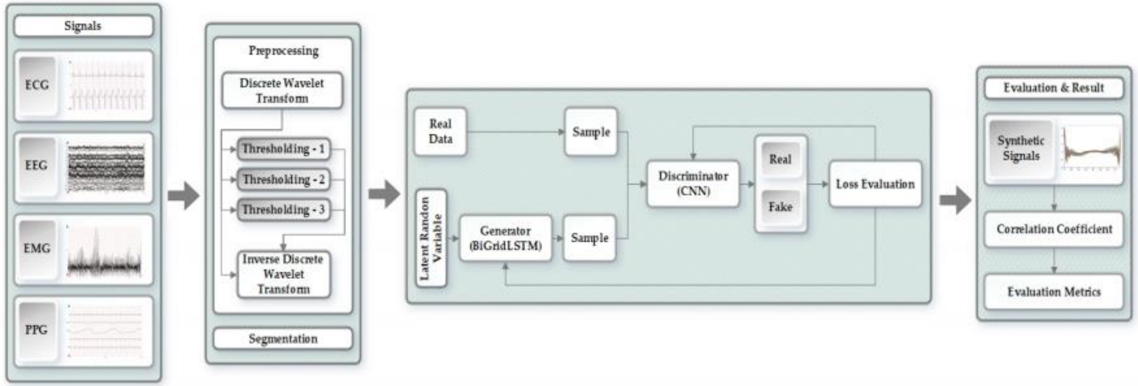
- **Non-convergence:** the model parameters oscillate, destabilize and never converge,
- **Mode collapse:** the generator collapses which produces limited varieties of samples,
- **Diminished gradient:** the discriminator gets too successful that the generator gradient vanishes and learns nothing,
- Unbalance between the generator and discriminator causing overfitting, &
- Highly sensitive to the hyperparameter selections.

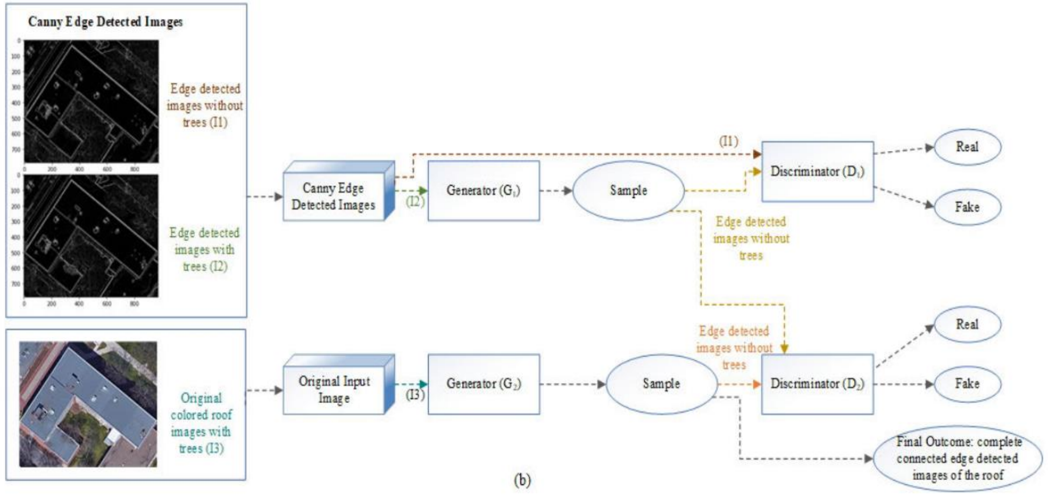
GENERATIVE ADVERSARIAL NETWORKS (GAN) (2014)

Our work









TRANSFER LEARNING

- The idea is to **transfer something learned from one task and apply it to another.**

When should we use transfer learning? Transfer learning can be applied in the following situations, depending on:

- The size of the new (target) dataset
- Similarity between the original and target datasets

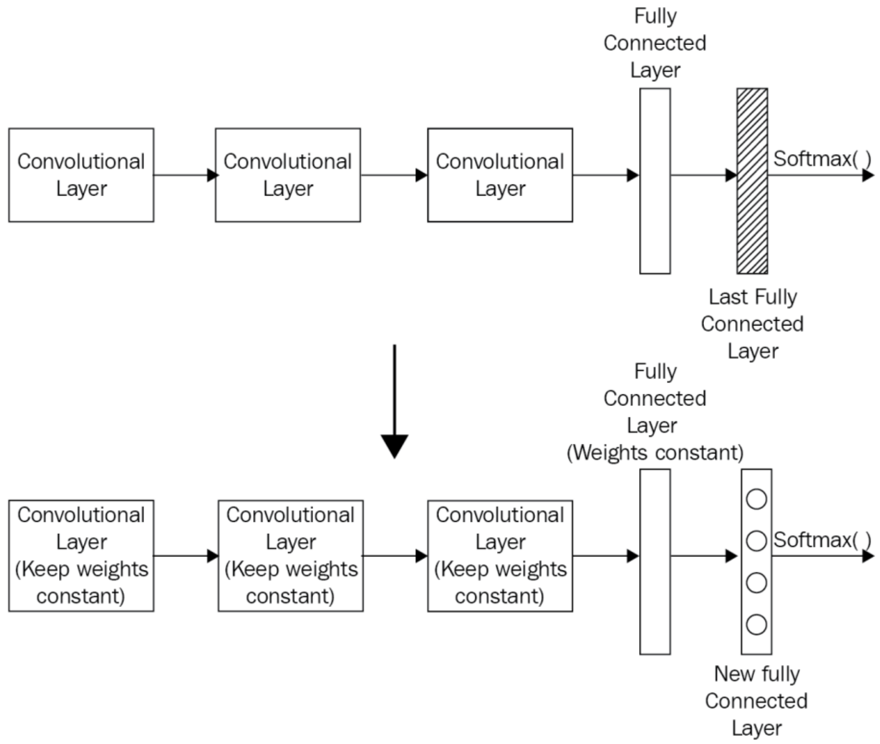
There are four main use cases:

- **Case 1:** New (target) dataset is small and is similar to the original training dataset
- **Case 2:** New (target) dataset is small but is different from the original training dataset
- **Case 3:** New (target) dataset is large and is similar to the original training dataset
- **Case 4:** New (target) dataset is large and is different from the original training dataset

TRANSFER LEARNING

Target dataset is small and is similar to the original training dataset

- In this case, replace the last fully connected layer with a **new fully connected layer that matches with the number of classes of the target dataset**
- Initialize old weights with **randomized weights**
- Train the network to update the weights of the new, fully connected layer:

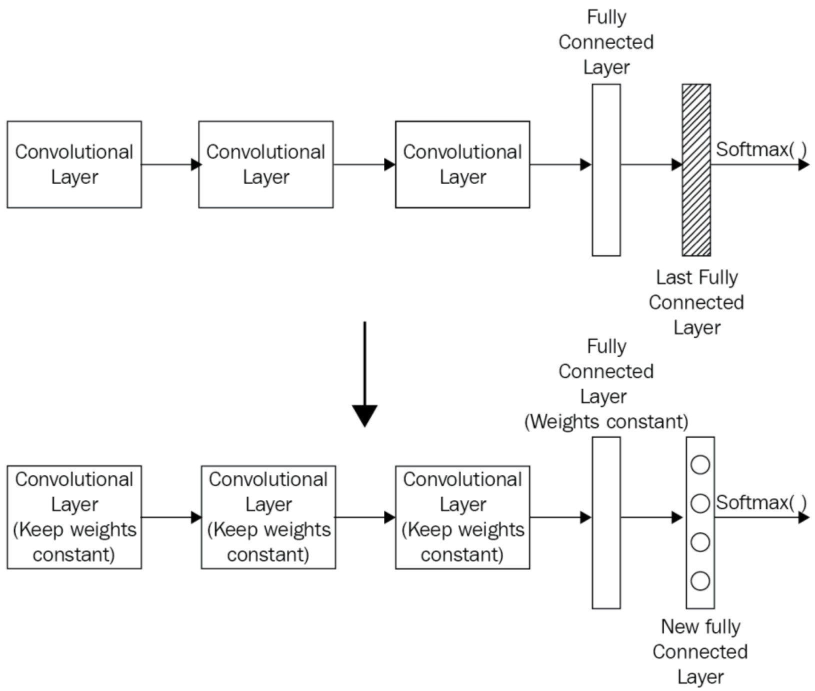


(Transfer Learning : Small data set and Similar data)

TRANSFER LEARNING

Target dataset is small but different from the original training dataset

- Slice most of the initial layers of the network
- Add to the remaining pre-trained layers a new fully connected layer that matches the number of classes of the target dataset
- Randomize the weights of the new fully connected layer and freeze all the weights from the pre-trained network
- Train the network to update the weights of the new fully connected layer

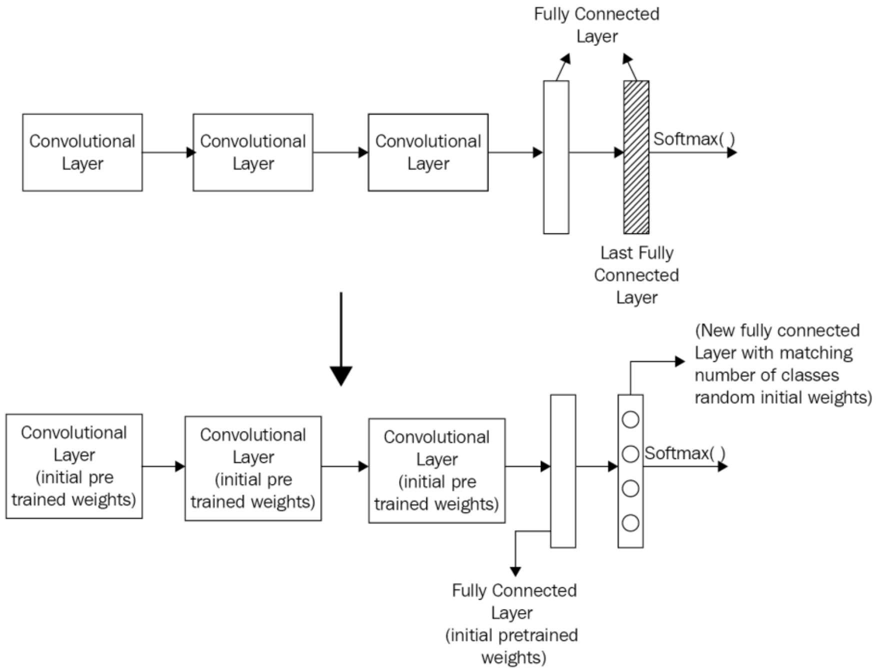


(Transfer Learning : Small data set and Similar data)

TRANSFER LEARNING

Target dataset is large and similar to the original training dataset

- Remove the last fully connected layer and replace it with a fully connected layer that matches the number of classes in the target dataset
- Randomly initialize the weights of this newly added, fully connected layer
- Initialize the rest of the weights with pre-trained weights
- Train the entire network:

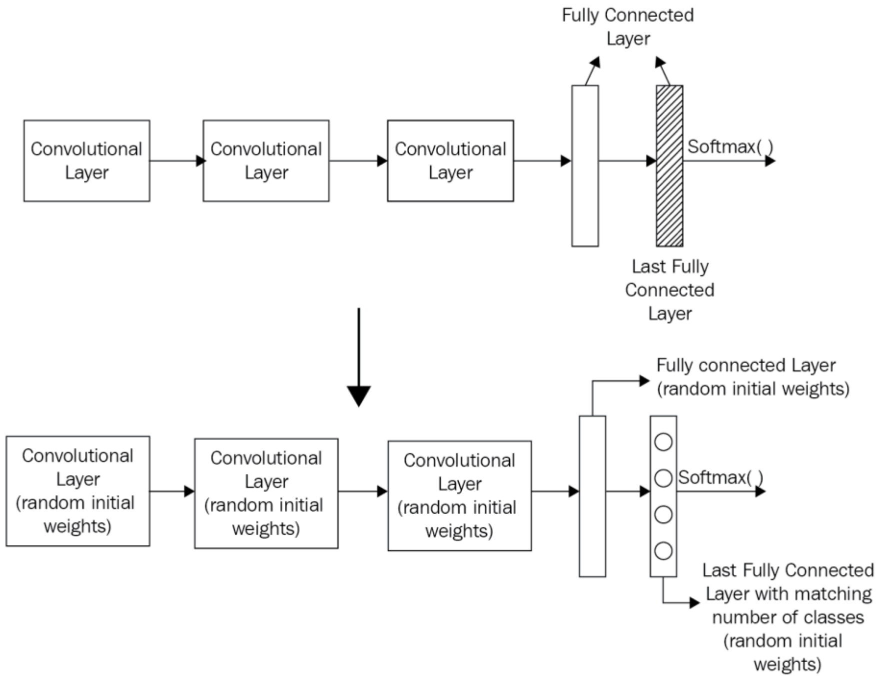


(Transfer Learning : Large data set and Similar data)

TRANSFER LEARNING

Target dataset is large and different from the original training dataset

- Remove the last fully connected layer and replace it with a fully connected layer that matches the number of classes in the target dataset
- Train the entire network from scratch with randomly initialized weights:



(Transfer Learning : Large data set and Different data)

Thank you !

ANY QUESTIONS ?