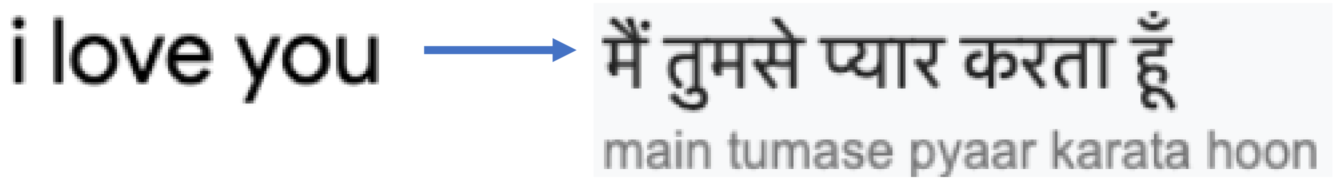
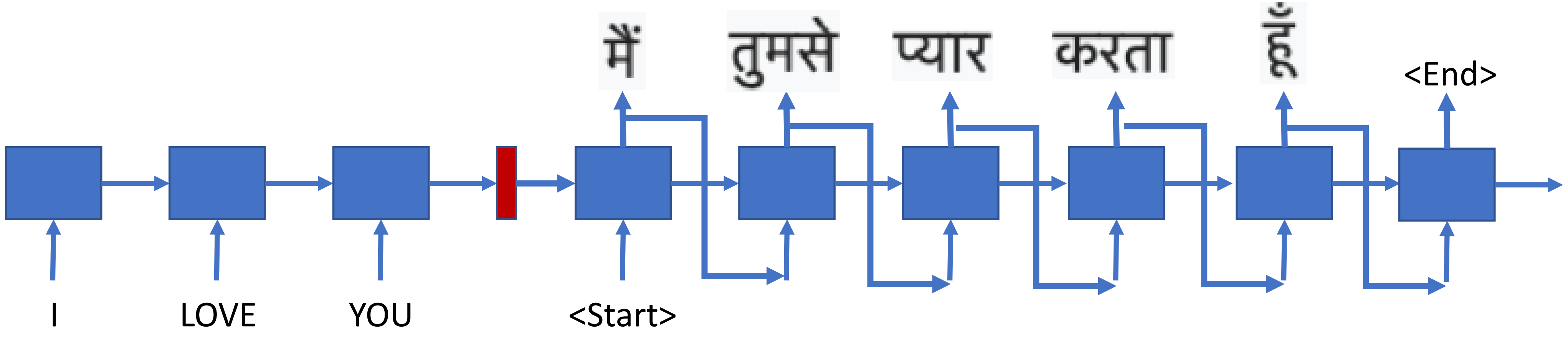
Attention

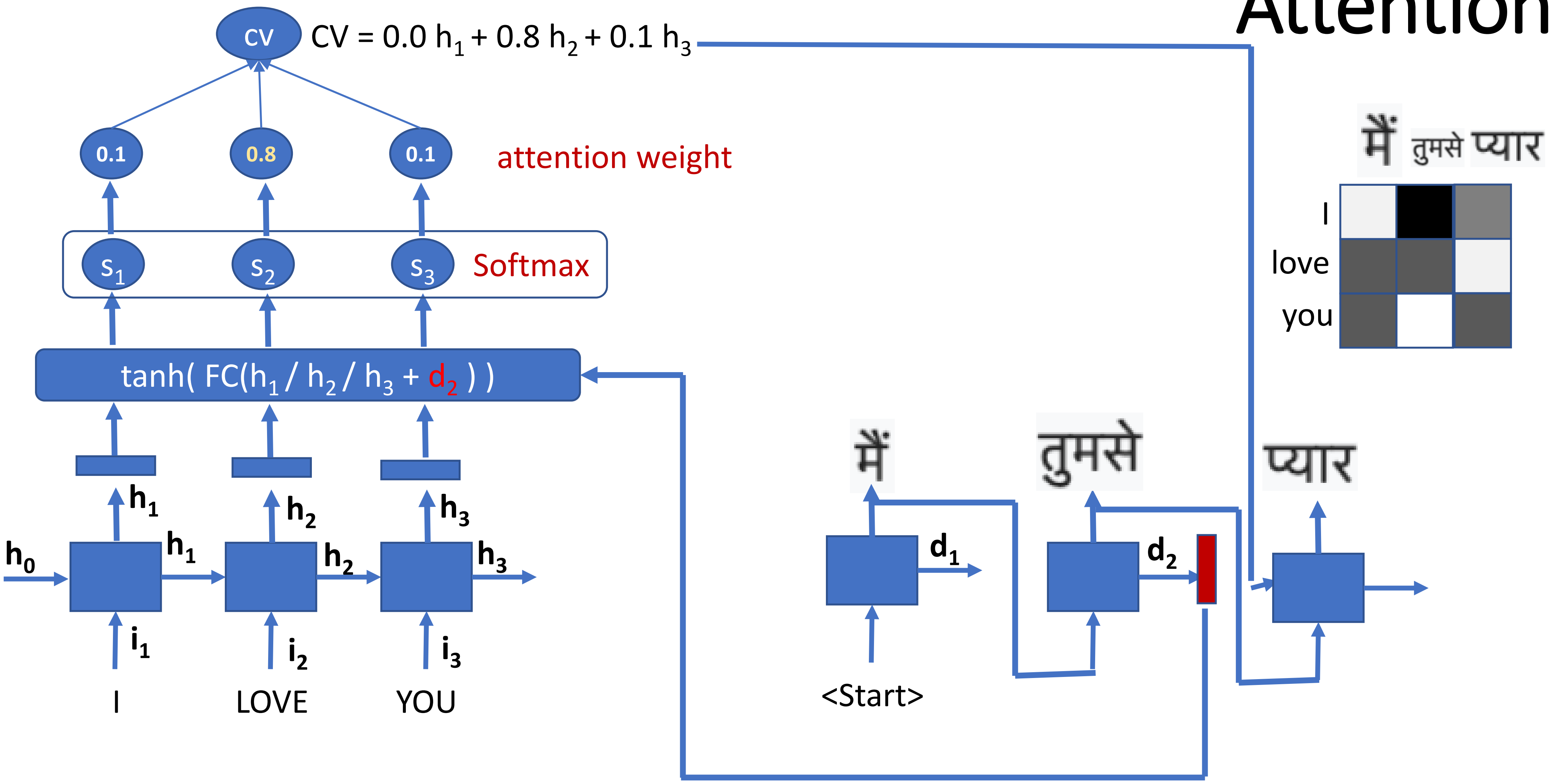
Attention on Sequence to Sequence Models

Sequence-to-Sequence (Seq2Seq)





Attention

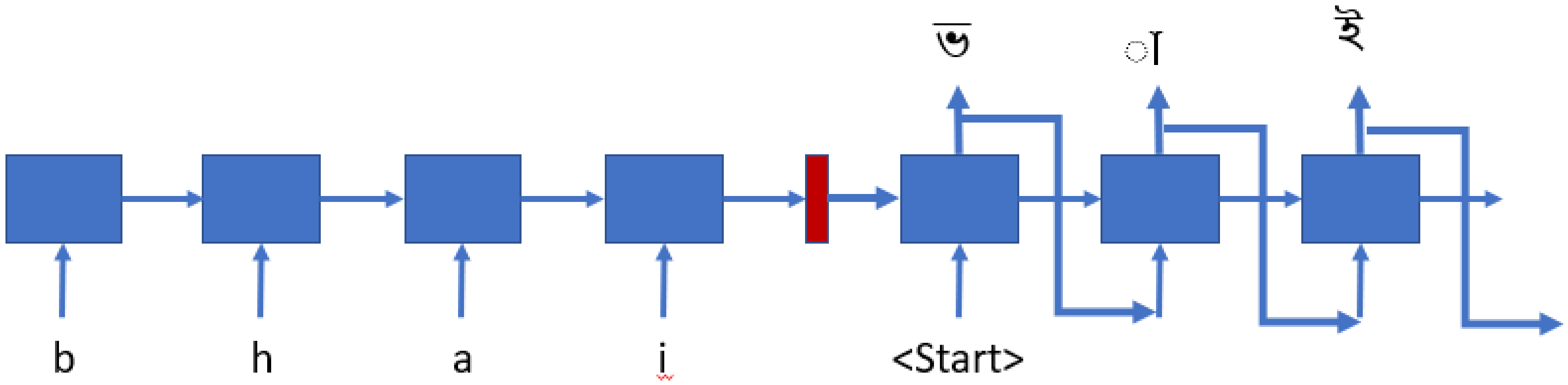


CV
CV = 0.0 h1 + 0.8 h2 + 0.1 h3
0.1
0.8
0.1
attention weight
s1
s2
s3
Softmax
tanh(FC(h1 / h2 / h3 + d2))
h1
h2
h3
h0
h1
h2
h3
i1
i2
i3
I
LOVE
YOU
मैं
तुमसे
प्यार
d1
d2
<Start>
मैं तुमसे प्यार
I
love
you

Machine Transliteration

Romanized Assamese: b h a i (brother)

Assamese in native script: ভ ◌া ই



Machine Transliteration

```
# Attention based Sequence to sequence model for Machine translation. The same model can be used for
# any sequence to sequence problem.
#
# In this example, we have applied form machine transliteration.
# Bidirectional LSTM units have been used for both encoder and decoder.
#
# Phonetically type assame word in roman script to native assamese script
#
# Ex.  "b h a i" -> "ভ া ই""
#
# This program is adopted by Hemanta, OSINT, CSE IITG from the following source
# https://github.com/Alireza-Akhavan/rnn-notebooks/blob/master/11_nmt-with-attention.ipynb


from __future__ import absolute_import, division, print_function
import tensorflow as tf
import matplotlib as mpl
import matplotlib.pyplot as plt
from sklearn.model_selection import train_test_split
import unicodedata
import re
import numpy as np
import os
import time

#tf.enable_eager_execution()
tf.executing_eagerly()
print(tf.__version__)
```

Machine Transliteration

b h a i	ভাই
e t a	এটা
d i y a s u n	দিয়াচোন
a	এ
a m k	আমাক
k o i	কৈ
a s e	আছে
t h i k	ঠিক
a n e k o i	এনেকৈ
t o p	টোপ
d i	দি
o d h i b e x o n t	অধিবেশনত
d a k h i l	দাখিল
h b o	হ'ব
n a k i	নেকি
k i	কি
l i k h i c h e	লিখিছে
k b o	ক'ব
n e	নে
o l o p	অলপ
n h o b o	নহ'ব
r e	ৰে
b h a a i	ভাই

Machine Transliteration

```
b h a i ভাই
e t a   এটা
d i y a s u n   দিয়াচোন
a       এ
a m k   আমাক
k o i   কৈ
a s e   আছে
t h i k ঠিক
a n e k o i     এনেকৈ
t o p   টোপ
d i     দি
o d h i b e x o n t     অধিবেশনত
d a k h i l     দাখিল
h b o   হ'ব
n a k i নেকি
k i     কি
l i k h i c h e লিখিছে
k b o   ক'ব
n e     নে
o l o p অলপ
n h o b o       নহ'ব
r e     ৰে
b h a a i       ভাই
```

```
['<start> b h a i <end>', '<start> ভ া ই <end>']
['<start> e t a <end>', '<start> এ ট া <end>']
['<start> d i y a s u n <end>', '<start> দ ি য় া চ ো ন <end>']
['<start> a <end>', '<start> এ <end>']
['<start> a m k <end>', '<start> আ ম া ক <end>']
['<start> k o i <end>', '<start> ক ৈ <end>']
['<start> a s e <end>', '<start> আ ছ ে <end>']
['<start> t h i k <end>', '<start> ঠ ি ক <end>']
['<start> a n e k o i <end>', '<start> এ ন ে ক ৈ <end>']
['<start> t o p <end>', '<start> ট ো প <end>']
['<start> d i <end>', '<start> দ ি <end>']
['<start> o d h i b e x o n t <end>', '<start> অ ধ ি ব ে শ ন ত <end>']
['<start> d a k h i l <end>', '<start> দ া খ ি ল <end>']
['<start> h b o <end>', "<start> হ ' ব <end>"]
['<start> n a k i <end>', '<start> ন ে ক ি <end>']
['<start> k i <end>', '<start> ক ি <end>']
['<start> l i k h i c h e <end>', '<start> ল ি খ ি ছ ে <end>']
['<start> k b o <end>', "<start> ক ' ব <end>"]
['<start> n e <end>', '<start> ন ে <end>']
```

Machine Transliteration

```
b h a i ভাই
e t a   এটা
d i y a s u n   দিয়াচোন
a       এ
a m k   আমাক
```

```
['<start> b h a i <end>', '<start> ভ া ই <end>']
['<start> e t a <end>', '<start> এ ট া <end>']
['<start> d i y a s u n <end>', '<start> দ ি য় া চ ো ন <end>']
['<start> a <end>', '<start> এ <end>']
['<start> a m k <end>', '<start> আ ম া ক <end>']
```

```python
# Path of the dataset
path_to_file = "data/train-en-as.txt"
```

```python
# Few of the text processing functions ate defined here

# Add <start> and <end> tokens at start and at the end
def preprocess_sentence(w):
    w = '<start> ' + w + ' <end>'
    return w

def create_dataset(path, num_examples):
    lines = open(path, encoding='UTF-8').read().strip().split('\n')
    word_pairs = [[preprocess_sentence(w) for w in l.split('\t')] for l in lines[:num_examples]]
    return word_pairs
```

Machine Transliteration

```
# This class creates a word -> index mapping (e.g,. "dad" -> 5) and vice-versa
# (e.g., 5 -> "dad") for each language,

# In our case the below class create (character -> index) and (index -> character) mapping
# for each language

class LanguageIndex():
    def __init__(self, lang):
        self.lang = lang
        self.word2idx = {}
        self.idx2word = {}
        self.vocab = set()
        self.create_index()

    def create_index(self):
        for phrase in self.lang:
            self.vocab.update(phrase.split(' '))
        self.vocab = sorted(self.vocab)
        self.word2idx['<pad>'] = 0
        for index, word in enumerate(self.vocab):
            self.word2idx[word] = index + 1

        for word, index in self.word2idx.items():
            self.idx2word[index] = word
```

Machine Transliteration

```python
# This class creates a word -> index mapping (e.g,. "dad" -> 5) and vice-versa
# (e.g., 5 -> "dad") for each langu

# In our case the below class create
# for each language

class LanguageIndex():
    def __init__(self, lang):
        self.lang = lang
        self.word2idx = {}
        self.idx2word = {}
        self.vocab = set()
        self.create_index()

    def create_index(self):
        for phrase in self.lang:
            self.vocab.update(phrase.split(' '))
        self.vocab = sorted(self.vocab)
        self.word2idx['<pad>'] = 0
        for index, word in enumerate(self.vocab):
            self.word2idx[word] = index + 1

        for word, index in self.word2idx.items():
            self.idx2word[index] = word
```

{'<pad>': 0, '<end>': 1, '<start>': 2, 'a': 3, 'b': 4, 'c': 5, 'd': 6, 'e': 7, 'f': 8, 'g': 9, 'h': 10, 'i': 11, 'j': 12, 'k': 13, 'l': 14, 'm': 15, 'n': 16, 'o': 17, 'p': 18, 'r': 19, 's': 20, 't': 21, 'u': 22, 'v': 23, 'w': 24, 'x': 25, 'y': 26, 'z': 27}
{'<pad>': 0, "'": 1, '<end>': 2, '<start>': 3, 'ঁ': 4, 'ং': 5, 'অ': 6, 'আ': 7, 'ই': 8, 'উ': 9, 'এ': 10, 'ঐ': 11, 'ও': 12, 'ক': 13, 'খ': 14, 'গ': 15, 'ঘ': 16, 'ঙ': 17, 'চ': 18, 'ছ': 19, 'জ': 20, 'ট': 21, 'ঠ': 22, 'ড': 23, 'ণ': 24, 'ত': 25, 'থ': 26, 'দ': 27, 'ধ': 28, 'ন': 29, 'প': 30, 'ফ': 31, 'ব': 32, 'ভ': 33, 'ম': 34, 'য': 35, 'ল': 36, 'শ': 37, 'ষ': 38, 'স': 39, 'হ': 40, 'া': 41, 'ি': 42, 'ী': 43, 'ু': 44, 'ূ': 45, 'ে': 46, 'ৈ': 47, 'ো': 48, 'ৌ': 49, '্': 50, 'ড়': 51, 'য়': 52, 'ৰ': 53, 'ৱ': 54}

{0: '<pad>', 1: '<end>', 2: '<start>', 3: 'a', 4: 'b', 5: 'c', 6: 'd', 7: 'e', 8: 'f', 9: 'g', 10: 'h', 11: 'i', 12: 'j', 13: 'k', 14: 'l', 15: 'm', 16: 'n', 17: 'o', 18: 'p', 19: 'r', 20: 's', 21: 't', 22: 'u', 23: 'v', 24: 'w', 25: 'x', 26: 'y', 27: 'z'}
{0: '<pad>', 1: "'", 2: '<end>', 3: '<start>', 4: 'ঁ', 5: 'ং', 6: 'অ', 7: 'আ', 8: 'ই', 9: 'উ', 10: 'এ', 11: 'ঐ', 12: 'ও', 13: 'ক', 14: 'খ', 15: 'গ', 16: 'ঘ', 17: 'ঙ', 18: 'চ', 19: 'ছ', 20: 'জ', 21: 'ট', 22: 'ঠ', 23: 'ড', 24: 'ণ', 25: 'ত', 26: 'থ', 27: 'দ', 28: 'ধ', 29: 'ন', 30: 'প', 31: 'ফ', 32: 'ব', 33: 'ভ', 34: 'ম', 35: 'য', 36: 'ল', 37: 'শ', 38: 'ষ', 39: 'স', 40: 'হ', 41: 'া', 42: 'ি', 43: 'ী', 44: 'ু', 45: 'ূ', 46: 'ে', 47: 'ৈ', 48: 'ো', 49: 'ৌ', 50: '্', 51: 'ড়', 52: 'য়', 53: 'ৰ', 54: 'ৱ'}

Machine Transliteration

```
def max_length(tensor):
    return max(len(t) for t in tensor)


def load_dataset(path, num_examples):
    # creating cleaned input, output pairs
    pairs = create_dataset(path, num_examples)

    # index language using the class defined above
    inp_lang = LanguageIndex(en for en, ass in pairs)
    targ_lang = LanguageIndex(ass for en, ass in pairs)

    # English source tensor (Phonetically typed Assamese)
    input_tensor = [[inp_lang.word2idx[s] for s in en.split(' ')] for en, ass in pairs]
    #print(input_tensor)

    # Assamese target tensor
    target_tensor = [[targ_lang.word2idx[s] for s in ass.split(' ')] for en, ass in pairs]

    max_length_inp, max_length_tar = max_length(input_tensor), max_length(target_tensor)

    # Padding the input and output tensor to the maximum length
    input_tensor = tf.keras.preprocessing.sequence.pad_sequences(input_tensor,
                                                                 maxlen=max_length_inp,
                                                                 padding='post')

    #print(input_tensor)
    target_tensor = tf.keras.preprocessing.sequence.pad_sequences(target_tensor,
                                                                  maxlen=max_length_tar,
                                                                  padding='post')

    return input_tensor, target_tensor, inp_lang, targ_lang, max_length_inp, max_length_tar
```

<start> b h a i<end>

[1, 4, 10, 3, 11, 2]

<start> ভ া ই <end>

[1, 33, 41, 8, 2]

Machine Transliteration

```
# Try experimenting with the size of that dataset
num_examples = 1000
input_tensor, target_tensor, inp_lang, targ_lang, max_length_inp, max_length_targ = load_dataset(path_to_file, num_examples)


# Creating training and validation sets using an 80-20 split
input_tensor_train, input_tensor_val, target_tensor_train, target_tensor_val = train_test_split(input_tensor,
                                                                                                target_tensor, test_size=0.2)

# Show length
print(len(input_tensor_train))
print(len(target_tensor_train))
print(len(input_tensor_val))
print(len(target_tensor_val))

BUFFER_SIZE = len(input_tensor_train)
BATCH_SIZE = 64
N_BATCH = BUFFER_SIZE//BATCH_SIZE
embedding_dim = 256
units = 256
vocab_inp_size = len(inp_lang.word2idx)
vocab_tar_size = len(targ_lang.word2idx)

dataset = tf.data.Dataset.from_tensor_slices((input_tensor_train, target_tensor_train)).shuffle(BUFFER_SIZE)
dataset = dataset.batch(BATCH_SIZE, drop_remainder=True)
```

```
800
800
200
200
```

Machine Transliteration

```
# Forward LSTM
def lstm(units):
    return tf.keras.layers.LSTM(units, return_sequences=True, return_state=True, recurrent_initializer='glorot_uniform')

# Backward LSTM
def lstm_bw(units):
    return tf.keras.layers.LSTM(units, return_sequences=True, return_state=True, go_backwards=True,
                                recurrent_initializer='glorot_uniform')

class Encoder(tf.keras.Model):
    def __init__(self, vocab_size, embedding_dim, enc_units, batch_sz):
        super(Encoder, self).__init__()
        self.batch_sz = batch_sz
        self.enc_units = enc_units
        self.embedding = tf.keras.layers.Embedding(vocab_size, embedding_dim)
        self.lstm = lstm(self.enc_units)
        self.lstm_bw = lstm_bw(self.enc_units)

    def call(self, x, hidden):
        x = self.embedding(x)
        output_fw, state_fw_h, state_fw_c = self.lstm(x, initial_state=hidden)
        output_bw, state_bw_h, state_bw_c = self.lstm_bw(x, initial_state=hidden)
        output_bw = tf.reverse(output_bw, [-2])
        output  = tf.concat((output_fw, output_bw), -1)
        state_h = tf.concat((state_fw_h, state_bw_h), -1)
        state_c = tf.concat((state_fw_c, state_bw_c), -1)
        state = [state_h, state_c]
        return output, state

    def initialize_hidden_state(self):
        return tf.zeros((self.batch_sz, self.enc_units))###
```

Machine Transliteration

```
class Decoder(tf.keras.Model):
    def __init__(self, vocab_size, embedding_dim, dec_units, batch_sz):
        super(Decoder, self).__init__()
        self.batch_sz = batch_sz
        self.dec_units = dec_units * 2
        self.embedding = tf.keras.layers.Embedding(vocab_size, embedding_dim)
        self.lstm = lstm(self.dec_units)
        self.fc = tf.keras.layers.Dense(vocab_size)

        # used for attention
        self.W1 = tf.keras.layers.Dense(self.dec_units)
        self.W2 = tf.keras.layers.Dense(self.dec_units)
        self.V = tf.keras.layers.Dense(1)

    def call(self, x, hidden, enc_output):

        # Attention
        hidden_with_time_axis = tf.expand_dims(hidden[0], 1)
        score = self.V(tf.nn.tanh(self.W1(enc_output) + self.W2(hidden_with_time_axis)))
        attention_weights = tf.nn.softmax(score, axis=1)

        # context_vector shape after sum == (batch_size, hidden_size)
        context_vector = attention_weights * enc_output
        context_vector = tf.reduce_sum(context_vector, axis=1)

        x = self.embedding(x)
        x = tf.concat([tf.expand_dims(context_vector, 1), x], axis=-1)

        output, state_h, state_c = self.lstm(x)
        state = [state_h, state_c]
        output = tf.reshape(output, (-1, output.shape[2]))
        x = self.fc(output)

        return x, state, attention_weights

    def initialize_hidden_state(self):
        return tf.zeros([self.dec_units, self.dec_units])
```

Machine Transliteration

```
# Initialize Encoder and Decoder

encoder = Encoder(vocab_inp_size, embedding_dim, units, BATCH_SIZE)
decoder = Decoder(vocab_tar_size, embedding_dim, units, BATCH_SIZE)
```

```
# Define the optimizer: Adam

optimizer = tf.optimizers.Adam()
```

```
# Define the Loss function

def loss_function(real, pred):
    mask = 1 - np.equal(real, 0)
    loss_ = tf.nn.sparse_softmax_cross_entropy_with_logits(labels=real, logits=pred) * mask
    return tf.reduce_mean(loss_)
```

```
# Create checkpoint directory to save the training checkpoints

if not os.path.exists('training_checkpoints'):
        os.makedirs('training_checkpoints')

checkpoint_dir = "training_checkpoints/"
checkpoint_prefix = os.path.join(checkpoint_dir, "ckpt")
checkpoint = tf.train.Checkpoint(optimizer=optimizer, encoder=encoder, decoder=decoder)
```

Machine Transliteration

```
# Training the model
EPOCHS = 20

for epoch in range(EPOCHS):
    start = time.time()
    hidden = encoder.initialize_hidden_state()
    total_loss = 0

    for (batch, (inp, targ)) in enumerate(dataset):
        loss = 0
        with tf.GradientTape() as tape:
            enc_output, enc_hidden = encoder(inp, [hidden, hidden])###
            dec_hidden = enc_hidden
            dec_input = tf.expand_dims([targ_lang.word2idx['<start>']] * BATCH_SIZE, 1)

            # feeding the target as the next input
            for t in range(1, targ.shape[1]):
                # passing enc_output to the decoder
                predictions, dec_hidden, _ = decoder(dec_input, dec_hidden, enc_output)
                loss += loss_function(targ[:, t], predictions)
                dec_input = tf.expand_dims(targ[:, t], 1)

        batch_loss = (loss / int(targ.shape[1]))
        total_loss += batch_loss
        variables = encoder.variables + decoder.variables
        gradients = tape.gradient(loss, variables)
        optimizer.apply_gradients(zip(gradients, variables))

        if batch % 100 == 0:
            print('Epoch {} Batch {} Loss {:.4f}'.format(epoch + 1, batch, batch_loss.numpy()))
    # saving (checkpoint) the model every 2 epochs
    if (epoch + 1) % 2 == 0:
        checkpoint.save(file_prefix=checkpoint_prefix)

    print('Epoch {} Loss {:.4f}'.format(epoch + 1, total_loss / N_BATCH))
    print('Time taken for 1 epoch {} sec\n'.format(time.time() - start))
```

Machine Transliteration

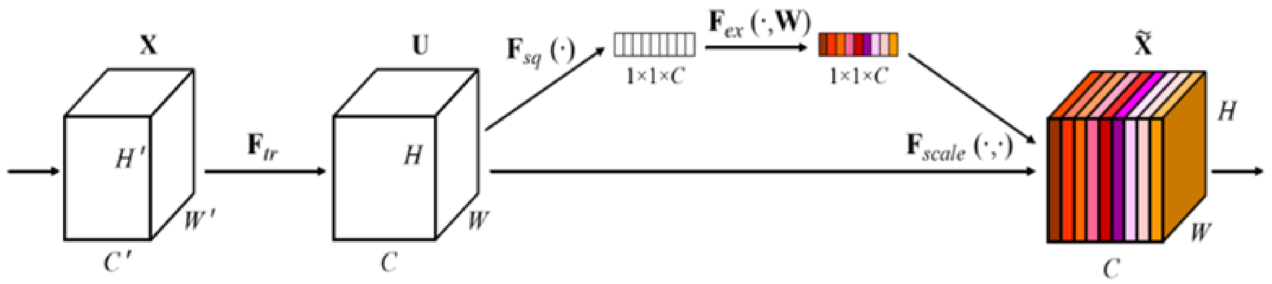
```
# Model evaluation

def evaluate(sentence, encoder, decoder, inp_lang, targ_lang, max_length_inp, max_length_targ):
    attention_plot = np.zeros((max_length_targ, max_length_inp))

    sentence = preprocess_sentence(sentence)

    inputs = [inp_lang.word2idx[i] for i in sentence.split(' ')]
    inputs = tf.keras.preprocessing.sequence.pad_sequences([inputs], maxlen=max_length_inp, padding='post')
    inputs = tf.convert_to_tensor(inputs)
    result = ''
    hidden = tf.zeros((1, units))###
    enc_out, enc_hidden = encoder(inputs, [hidden, hidden])###
    dec_hidden = enc_hidden
    dec_input = tf.expand_dims([targ_lang.word2idx['<start>']], 0)

    for t in range(max_length_targ):
        predictions, dec_hidden, attention_weights = decoder(dec_input, dec_hidden, enc_out)

        # storing the attention weigths to plot later on
        attention_weights = tf.reshape(attention_weights, (-1,))

        attention_plot[t] = attention_weights.numpy()

        predicted_id = tf.argmax(predictions[0]).numpy()

        result += targ_lang.idx2word[predicted_id] + ' '

        if targ_lang.idx2word[predicted_id] == '<end>':
            return result, sentence, attention_plot

        # the predicted ID is fed back into the model
        dec_input = tf.expand_dims([predicted_id], 0)

    return result, sentence, attention_plot
```

Machine Transliteration

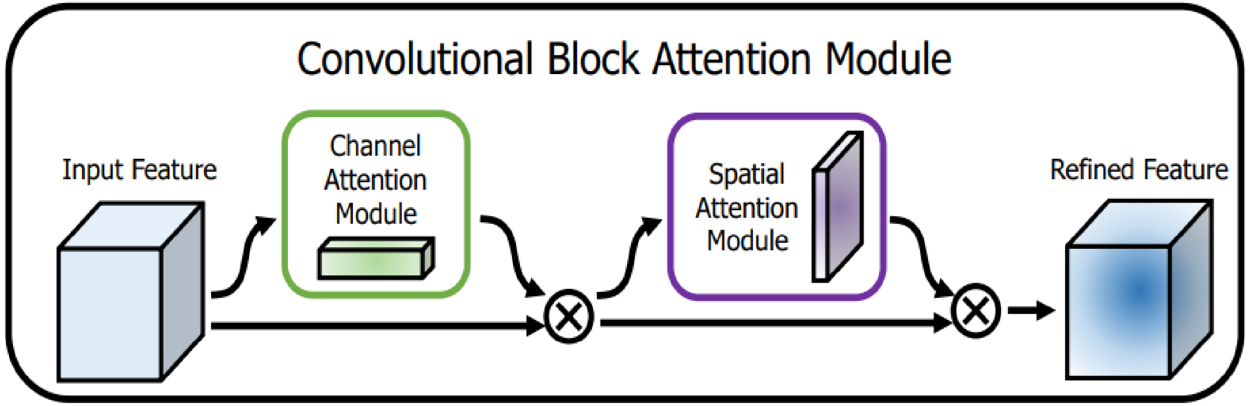
```python
# Translate test sentences

def translate(sentence, encoder, decoder, inp_lang, targ_lang, max_length_inp, max_length_targ, num):
    result, sentence, attention_plot = evaluate(sentence, encoder, decoder, inp_lang, targ_lang, max_length_inp,
                                                max_length_targ)
    r = sentence + "\t" + result
    print(r)
    with open("predicted-sentences-ass.txt", 'a', encoding='utf-8') as f:
       f.write(sentence + "\t" + result +'\n')
    attention_plot = attention_plot[:len(result.split(' ')), :len(sentence.split(' '))]
    plot_attention(attention_plot, sentence.split(' '), result.split(' '), num)

checkpoint.restore(tf.train.latest_checkpoint(checkpoint_dir))

with open('data/test-en.txt', 'r', encoding='utf-8') as f:
    lines = f.read().split('\n')
    n = len(lines)-1
    for line in lines[:300]:

        translate(line, encoder, decoder, inp_lang, targ_lang, max_length_inp, max_length_targ, n)
        n = n-1
```

```
<start> n y a y <end>   ন া য় ে <end>
<start> s r i <end>     ছ ৰ ি <end>
<start> k r i s h n a <end>     খ ি ছ া ং <end>
<start> n i d e a <end> ন ি দ ে <end>
<start> d o y a <end>   দ ি য় <end>
```

Attention on CNN



Squeeze and Excitation Model



Convolutional Block Attention Module

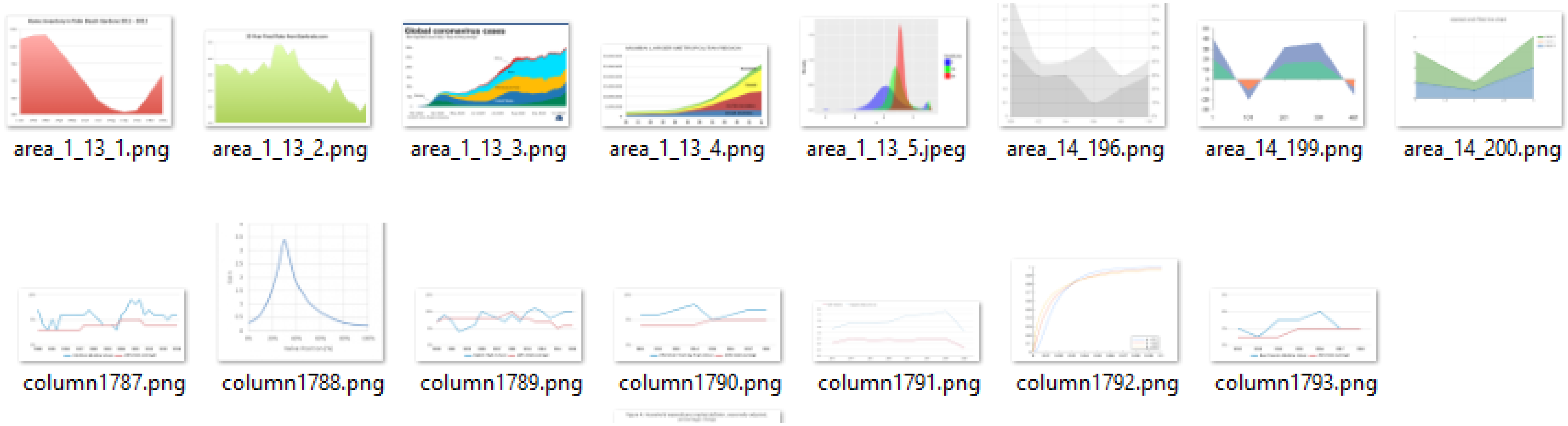
Attention on CNN

```
# This program is developed by Jennil, OSINT, CSE IITG
# CNN with Squeeze and CBAM Attentions

#from keras.preprocessing.image import ImageDataGenerator, img_to_array, load_img
from keras.models import Sequential, Model
from keras.layers import Conv2D, MaxPooling2D, Concatenate, Reshape, Input, Multiply
from keras.layers import Activation, Dropout, Flatten, Dense, GlobalAveragePooling2D, GlobalMaxPooling2D
from keras import backend as K
import tensorflow as tf
#from tensorflow.keras.layers import GlobalAveragePooling2D, Reshape, Dense, Input
import matplotlib.pyplot as plt
from tensorflow.keras.preprocessing.image import ImageDataGenerator, img_to_array, load_img
import numpy as np
```

Attention on CNN

```python
# Load the training and testing datasets
img_width, img_height = 224, 224

trdata = ImageDataGenerator()
traindata = trdata.flow_from_directory(directory="./train",target_size=(224,224))
tsdata = ImageDataGenerator()
testdata = tsdata.flow_from_directory(directory="./test", target_size=(224,224))
```

Name	Date modified	Type	Size
area	12/13/2022 11:49 AM	File folder	
line	12/13/2022 11:50 AM	File folder	

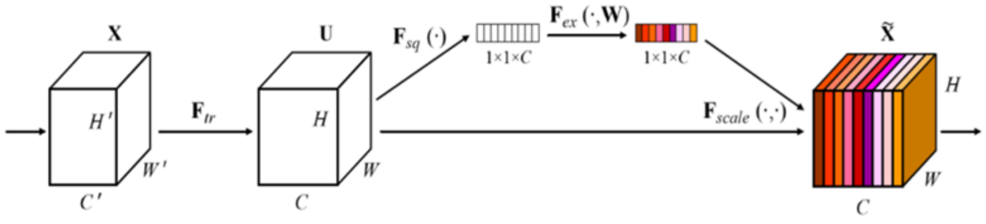


Attention on CNN

```
# Set the data shape
if K.image_data_format() == 'channels_first':
    input_shape = (3, img_width, img_height)
else:
    input_shape = (img_width, img_height, 3)
print(input_shape)
```

```
(224, 224, 3)
```

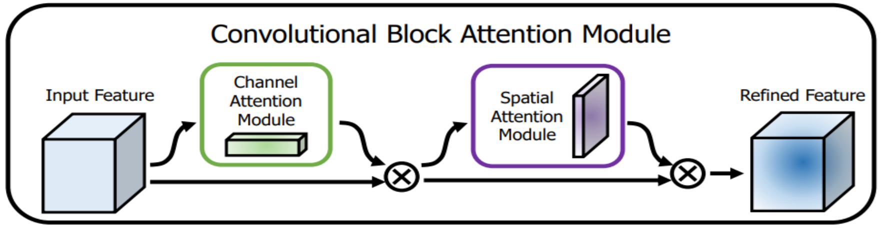
```
def SE(x):
    # ------------- Estimate the Attention weight ---------------
    # Add a squeeze and excitation block
    x = GlobalAveragePooling2D()(x)
    x = Dense(units=8, activation='relu')(x)
    x = Dense(units=8, activation='sigmoid')(x)

    # ------------- Apply Attention weight to feature map---------------
    x = Multiply()([x, original_x])
    return x
```



Attention on CNN

```
def channel_attention_module(x, ratio=8):
    batch, h, w, channel = x.shape

    ## Shared layers
    l1 = Dense(channel//ratio, activation="relu", use_bias=False)
    l2 = Dense(channel, use_bias=False)

    ## Global Average Pooling
    x1 = GlobalAveragePooling2D()(x)
    x1 = l1(x1)
    x1 = l2(x1)

    ## Global Max Pooling
    x2 = GlobalMaxPooling2D()(x)
    x2 = l1(x2)
    x2 = l2(x2)

    ## Add both the features and pass through sigmoid
    feats = x1 + x2
    feats = Activation("sigmoid")(feats)
    feats = Multiply()([x, feats])

    return feats
```

```
def spatial_attention_module(x):
    ## Average Pooling
    x1 = tf.reduce_mean(x, axis=-1)
    x1 = tf.expand_dims(x1, axis=-1)

    ## Max Pooling
    x2 = tf.reduce_max(x, axis=-1)
    x2 = tf.expand_dims(x2, axis=-1)

    ## Concatenat both the features
    feats = Concatenate()([x1, x2])
    ## Conv layer
    feats = Conv2D(1, kernel_size=7, padding="same", activation="sigmoid")(feats)
    feats = Multiply()([x, feats])

    return feats

def CBAM(x):
    x = channel_attention_module(x)
    x = spatial_attention_module(x)
    return x
```



Attention on CNN

```
# Define the CNN

# Set the input shape and number of classes
#input_shape = (224, 224, 3)  # 128x128 pixels, 3 channels (RGB)

# Create the input layer
inputs = Input(shape=input_shape)

# Add a series of convolutional and max pooling layers
x = Conv2D(filters=8, kernel_size=(3, 3), activation='relu')(inputs)
x = MaxPooling2D(pool_size=(2, 2))(x)
x = Conv2D(filters=8, kernel_size=(3, 3), activation='relu')(x)
x = MaxPooling2D(pool_size=(2, 2))(x)
original_x=x

# ------ Apply attention-----
#x = SE(x)
x = CBAM(x)

# Remaining CNN layer
f1=Flatten()(x)
d1=Dense(8, activation='relu')(f1)
outputs=Dense(2, activation='softmax')(d1)

# Create the model
model = Model(inputs=inputs, outputs=outputs)
```

```
model.compile(loss = "categorical_crossentropy",  optimizer = "adam", metrics=["accuracy"])
```

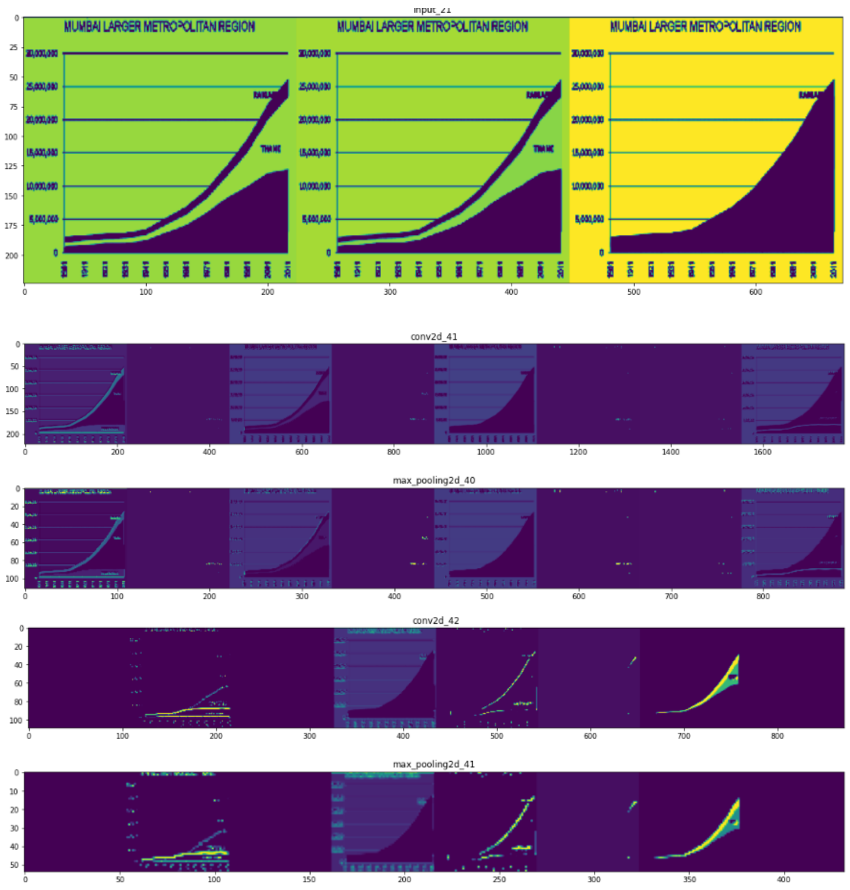
```
model.fit_generator(generator= traindata, steps_per_epoch= 2, epochs= 30, validation_data= testdata, validation_steps=1)
```

Attention on CNN

```
# Visulaization of Feature maps of an input image from successive layers
successive_outputs = [layer.output for layer in model.layers]

# Caputure output from each layer
visualization_model = Model(inputs = model.input, outputs = successive_outputs)

# Load a test image for visualization
img = load_img('./out/area_1_13_4.png', target_size=(224, 224))
x   = img_to_array(img)
x   = x.reshape((1,) + x.shape) # convent to array of images
x /= 255.0                      # convert to [0, 1]

# Get the output of the given image from each layer
successive_feature_maps = visualization_model.predict(x)
```

Attention on CNN

```python
# Display for each layer
layer_names = [layer.name for layer in model.layers]
for layer_name, feature_map in zip(layer_names, successive_feature_maps):

    if len(feature_map.shape) == 4:  # Print only for Convolution and Pooling layer (consider 4 to ignore dense layers)

        n_features = feature_map.shape[-1]   # number of channel or filters
        size       = feature_map.shape[ 1]   # size of the feature maps
        #print(size)

        # CReation of grid to display feature maps of all the fiters
        display_grid = np.zeros((size, size * n_features))

        #  Prepare the grid, and also change the pixel values to see the feature map visually
        for i in range(n_features):
            x  = feature_map[0, :, :, i]
            x -= x.mean()
            x /= x.std ()
            x *=  64
            x += 12
            x  = np.clip(x, 0, 255).astype('uint8')
            # Tile each filter into a horizontal grid
            display_grid[:, i * size : (i + 1) * size] = x
        # Display the grid
        scale = 20. / n_features
        plt.figure( figsize=(scale * n_features, scale) )
        plt.title ( layer_name )
        plt.grid  ( False )
        plt.imshow( display_grid, aspect='auto', cmap='viridis' )
```



Attention on CNN

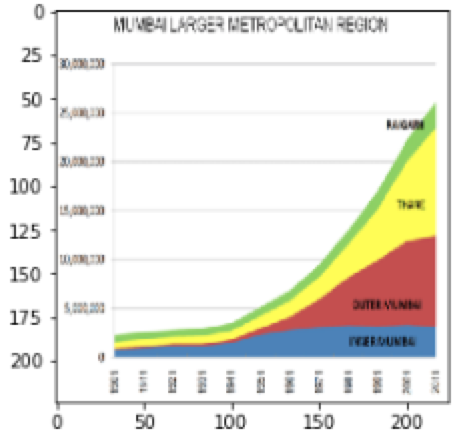
```
model.save('CNN.h5')
```

```
model = keras.models.load_model('CNN.h5')
```

```
img = load_img('out/area_1_13_4.png', target_size=(224, 224))
img = np.asarray(img)
img = np.expand_dims(img, axis=0)
predict = model.predict(img)
classes=predict=np.argmax(predict,axis=1)
```

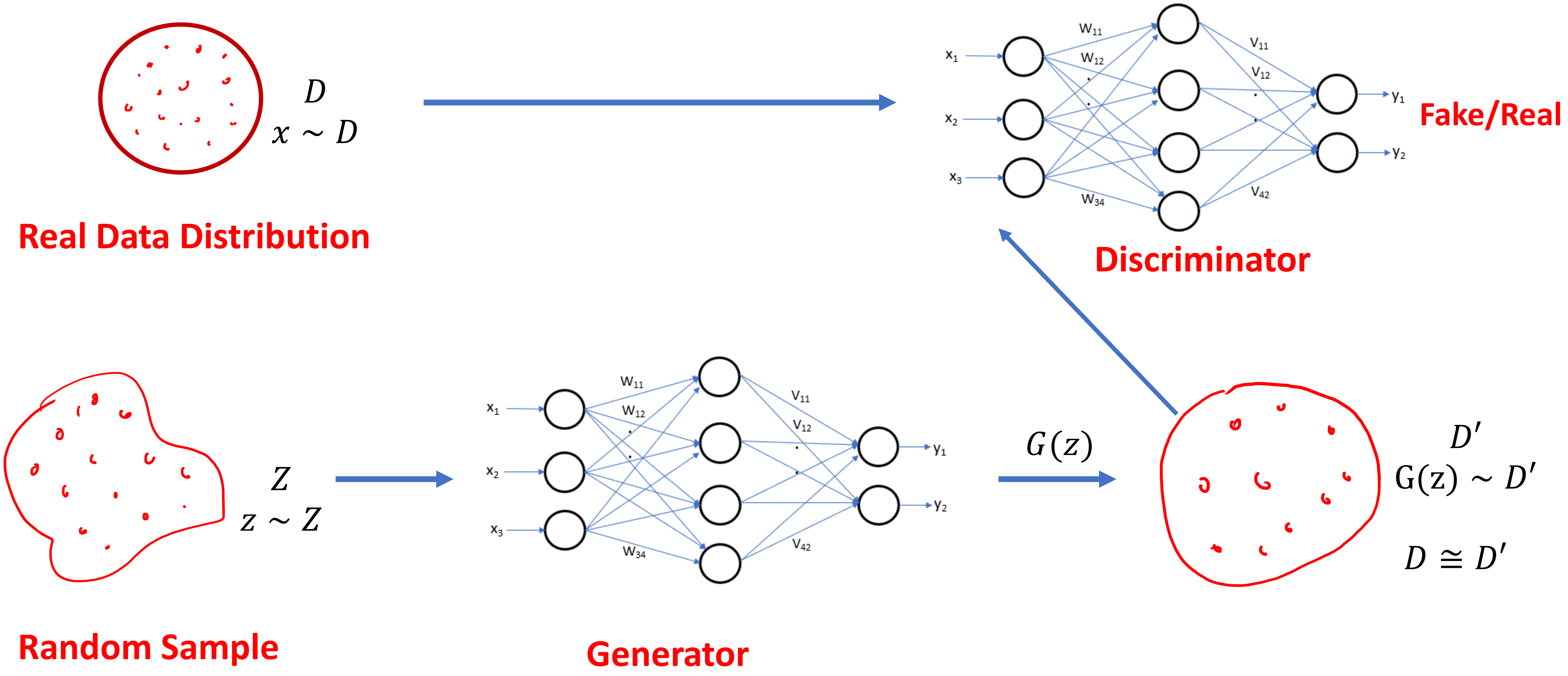
```
img = load_img('out/area_1_13_4.png', target_size=(224, 224))
plt.imshow(img)
if classes==0:
    print('The Class is AREA')
elif classes==1:
    print('The Class is LINE')
```

```
The Class is AREA
```



Generative Adversarial Network (GAN)

GAN



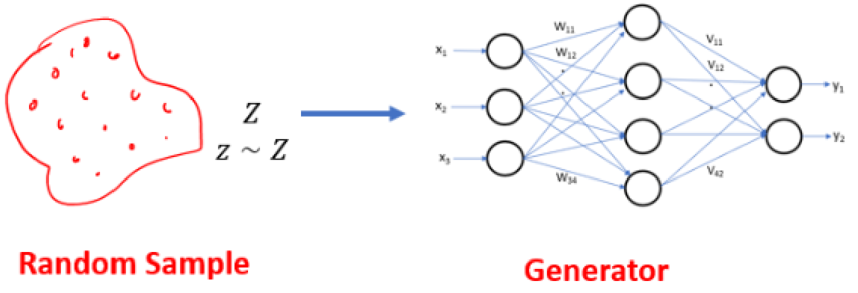
D
$x \sim D$
Real Data Distribution
x_1
x_2
x_3
W_{11}
W_{12}
W_{34}
V_{11}
V_{12}
V_{42}
y_1
y_2
Fake/Real
Discriminator
Z
$z \sim Z$
Random Sample
x_1
x_2
x_3
W_{11}
W_{12}
W_{34}
V_{11}
V_{12}
V_{42}
y_1
y_2
Generator
$G(z)$
D'
$G(z) \sim D'$
$D \cong D'$

Generative Adversarial Network (GAN)

```
from keras.datasets import mnist
from keras.layers import Input, Dense, Reshape, Flatten
from keras.layers import BatchNormalization
#from keras.layers.advanced_activations import LeakyReLU
from keras.layers import LeakyReLU
from keras.models import Sequential, Model
from tensorflow.keras.optimizers import Adam
import matplotlib.pyplot as plt
import numpy as np
```

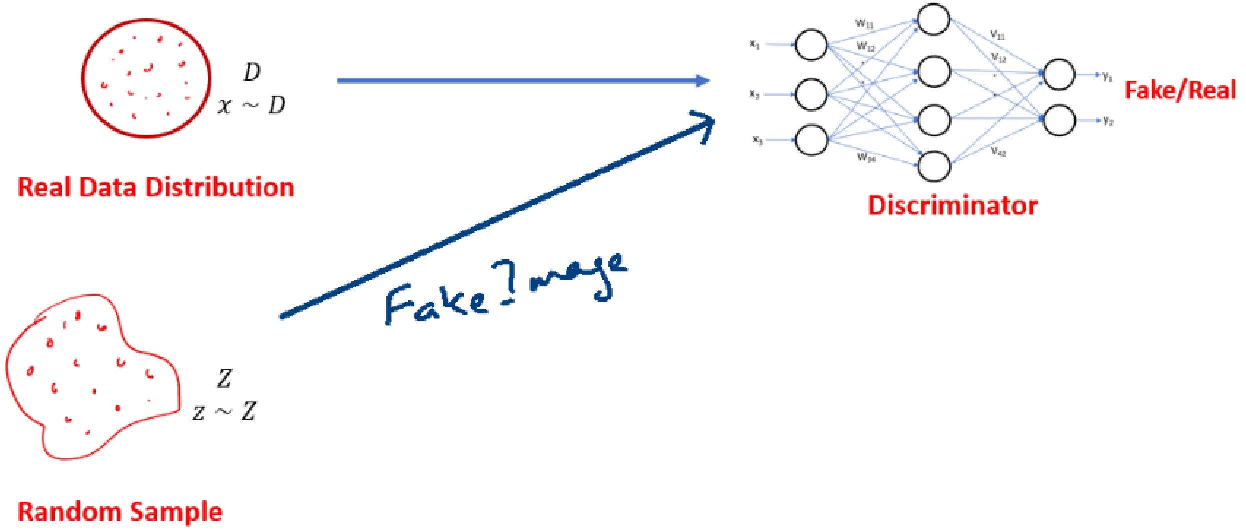
Generative Adversarial Network (GAN)

```
# Define the generator networ.
# This example comsider a MLP. Depending on the nature of the problem,
# it can be made more complex, and also different models such as VGG can also be used

def Generator(latentShape,imgShape):
    # Define the MLP model with one hidden layer. You can add more layers
    model = Sequential()
    model.add(Dense(128, input_shape=latentShape))
    model.add(LeakyReLU(alpha=0.2))
    model.add(BatchNormalization(momentum=0.8))
    model.add(Dense(np.prod(imgShape), activation='tanh'))
    model.add(Reshape(imgShape))
    model.summary()

    opt = Adam(0.0002, 0.5)  #Learning rate and momentum.
    model.compile(loss='binary_crossentropy', optimizer=opt, metrics=['accuracy'])

    return model
```



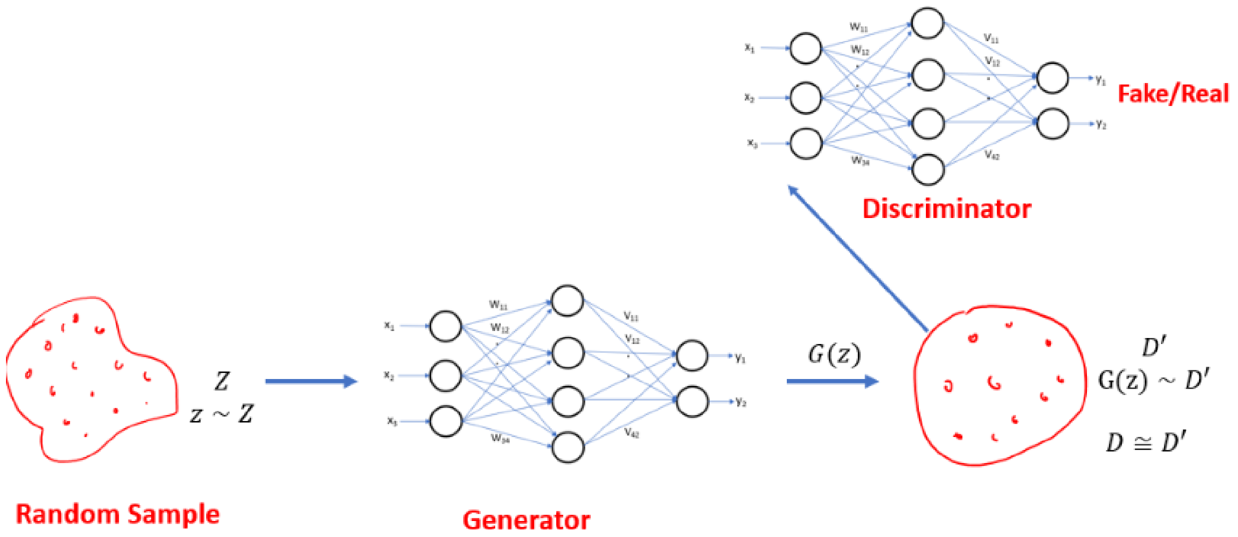
Generative Adversarial Network (GAN)

```
def Discriminator(imgShape):

    model = Sequential()
    model.add(Flatten(input_shape=imgShape))
    model.add(Dense(256))
    model.add(LeakyReLU(alpha=0.2))
    model.add(Dense(1, activation='sigmoid'))
    model.summary()
    opt = Adam(0.0002, 0.5)  #Learning rate and momentum.
    model.compile(loss='binary_crossentropy', optimizer=opt, metrics=['accuracy'])

    return model
```



Generative Adversarial Network (GAN)

```
# Combine the generator and discriminator models into one pipeline
def GAN(g, d):
    # Set the discriminator parameters to false
    d.trainable = False

    # Create a GAN model = generator + discriminator
    model = Sequential()
    model.add(g)
    model.add(d)    # discriminator takes output of generator as input

    # Set compile parameters
    opt = Adam(0.0002, 0.5)  #Learning rate and momentum.
    model.compile(loss='binary_crossentropy', optimizer=opt, metrics=['accuracy'])

    return model
```



Generative Adversarial Network (GAN)

```
# load the MNIST dataset and change its shape to 28x28x1
def get_real_samples():
    # load MNIST dataset without its class labels
    (mnistX, _), (_, _) = mnist.load_data()

    # scale the original values [0,255] to [-1,1]
    X = (mnistX.astype('float32') - 127.5) / 127.5
    X = np.expand_dims(X, axis=3) # expand the shape to 28x28x1
    return X
```

```
# While training the discriminator, only a random batch are considered
def get_random_real_samples(dataset, n):
    # choose random n samples
    X = dataset[np.random.randint(0, dataset.shape[0], n)]

    # Generate Class labels of real samples as 1
    y = np.ones((n, 1))
    return X, y
```

Generative Adversarial Network (GAN)

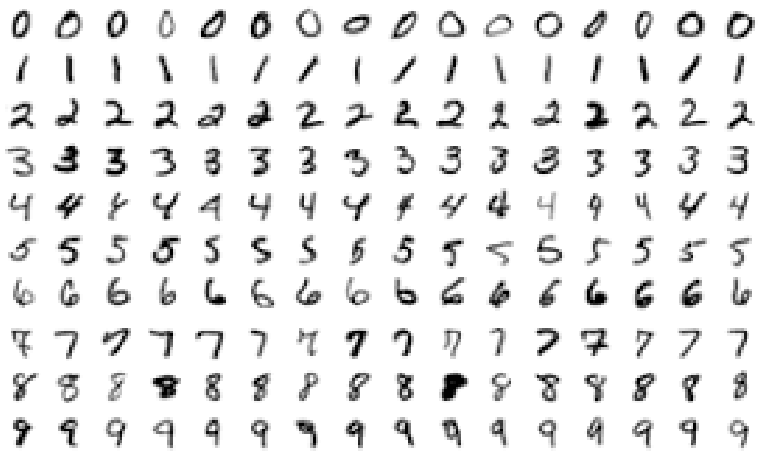
```python
# load the MNIST dataset and change its shape to 28x28x1
def get_real_samples():
    # load MNIST dataset without its class labels
    (mnistX, _), (_, _) = mnist.load_data()

    # scale the original values [0,255] to [-1,1]
    X = (mnistX.astype('float32') - 127.5) / 127.5
    X = np.expand_dims(X, axis=3) # expand the shape to 28x28x1
    return X
```

```python
# While training the discriminator, only a random batch are considered
def get_random_real_samples(dataset, n):
    # choose random n samples
    X = dataset[np.random.randint(0, dataset.shape[0], n)]

    # Generate Class labels of real samples as 1
    y = np.ones((n, 1))
    return X, y
```

MNIST database

From Wikipedia, the free encyclopedia

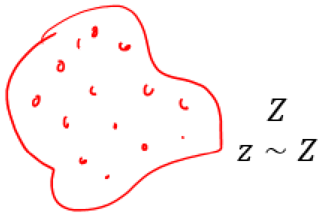
The **MNIST database** (*Modified National Institute of Standards and Technology database*[1]) is a large database of handwritten digits that is commonly used for training various image processing systems.[2][3] The database is also widely used for training and testing in the field of machine learning.[4][5] It was created by "re-mixing" the samples from NIST's original datasets.[6] The creators felt that since NIST's training dataset was taken from American Census Bureau employees, while the testing dataset was taken from American high school students, it was not well-suited for machine learning experiments.[7] Furthermore, the black and white images from NIST were normalized to fit into a 28x28 pixel bounding box and anti-aliased, which introduced grayscale levels.[7]



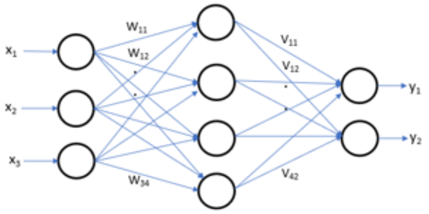
Generative Adversarial Network (GAN)

```python
# Generate n number of noise samples  for the generator
def generate_noise_samples(noise_dim, n):
    ld=np.prod(noise_dim) # convert noise_dim shape (28,28,1) to 28x28x1

    # generate n number of random noise samples of dim ld
    x_input = np.random.normal(0,1, (n, ld))
    # reshape into a batch of inputs for the network
    #x_input = x_input.reshape(n_samples, ld)
    return x_input
```

```python
# Generate FAKE samles using Generator from the noise samples
def generate_fake_samples(g_model, noise_dim, n):

    x_input = generate_noise_samples(noise_dim, n)

    # Generate Fake samples and its class label as 0
    X = g_model.predict(x_input)
    y = np.zeros((n, 1))

    return X, y
```



Random Sample



Generator

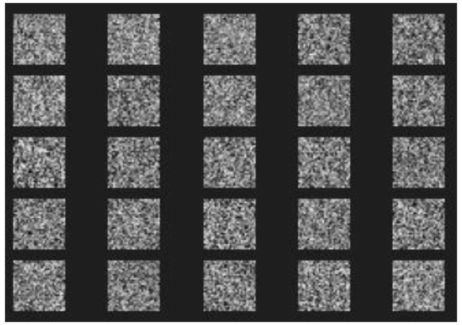
Generative Adversarial Network (GAN)

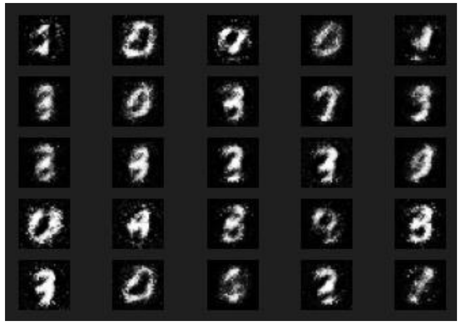
```
# ----------------
# For each epoch, randomely select a batch of real and fake images and train the discriminator and GAN
# ---------------
def train(g, d, gan, dataset, noise_dim, epochs, batch):

    #epo = int(dataset.shape[0] / batch)

    for i in range(epochs):


            # Train discriminator with real and random samples
            X_real, y_real = get_random_real_samples(dataset, batch)
            r_error, _ = d.train_on_batch(X_real, y_real)
            X_fake, y_fake = generate_fake_samples(g, noise_dim, batch)
            f_error, _ = d.train_on_batch(X_fake, y_fake)


            # Train GAN
            X_gan = generate_noise_samples(noise_dim, batch)
            y_gan = np.ones((batch, 1))
            g_error, _ = gan.train_on_batch(X_gan, y_gan)

            # Batch training loss
            print('Epoch %d -> Discriminator loss %f GAN loss %d' % (i, (r_error+f_error)/2, g_error))

            # Save the generated images every 1000 epoch
            if (i % 499 == 0):
                save_plot(g,i)
```

Generative Adversarial Network (GAN)

```
# create and save a plot of generated images
def save_plot(g,epoch):

    r, c = 5, 5
    noise = np.random.normal(0, 1, (r * c, 100))
    gen_imgs = g.predict(noise)

    # Rescale images 0 - 1
    gen_imgs = 0.5 * gen_imgs + 0.5

    fig, axs = plt.subplots(r, c)
    cnt = 0
    for i in range(r):
        for j in range(c):
            axs[i,j].imshow(gen_imgs[cnt, :,:,0], cmap='gray')
            axs[i,j].axis('off')
            cnt += 1
    fig.savefig("images/mnist_%d.png" % epoch)
    plt.close()
```

```
# size of the noise input
noiseShape = (100,)

# shape of the input image
imgRow = 28
imgCol = 28
imgChannels = 1
imgShape = (imgRow, imgCol, imgChannels)

# create the discriminator
d = Discriminator(imgShape)
# create the generator
g = Generator(noiseShape, imgShape)
# create the gan
gan = GAN(g, d)
# ret the real image data
dataset = get_real_samples()
# train model
train(g, d, gan, dataset, noiseShape, 5000, 256)
```





Generative Adversarial Network (GAN)

```
# Define the generator networ.
# This example comsider a CNN. Depending on the nature of the problem,
# it can be made more complex, and also different models such as VGG can also be used

def Generator(latentShape,imgShape):
    model = Sequential()
    # foundation for 4x4 image
    n_nodes = 256 * 4 * 4
    model.add(Dense(n_nodes, input_shape=latentShape))
    model.add(LeakyReLU(alpha=0.2))
    model.add(Reshape((4, 4, 256)))
    # upsample to 8x8
    model.add(Conv2DTranspose(128, (4,4), strides=(2,2), padding='same'))
    model.add(LeakyReLU(alpha=0.2))
    # upsample to 16x16
    model.add(Conv2DTranspose(128, (4,4), strides=(2,2), padding='same'))
    model.add(LeakyReLU(alpha=0.2))
    # upsample to 32x32
    model.add(Conv2DTranspose(128, (4,4), strides=(2,2), padding='same'))
    model.add(LeakyReLU(alpha=0.2))
    # output layer
    model.add(Conv2D(3, (3,3), activation='tanh', padding='same'))
    model.summary()

    opt = Adam(0.0002, 0.5)  #Learning rate and momentum.
    model.compile(loss='binary_crossentropy', optimizer=opt, metrics=['accuracy'])

    return model
```

```
# load the MNIST dataset and change its shape to 28x28x1
def get_real_samples():
    # load MNIST dataset without its class labels
    (cifar10X, _), (_, _) = cifar10.load_data()

    # scale the original values [0,255] to [-1,1]
    X = (cifar10X.astype('float32') - 127.5) / 127.5

    return X
```

```
# size of the noise input
noiseShape = (100,)
imgShape = (32,32,3)

# create the discriminator
d = Discriminator(imgShape)
# create the generator
g = Generator(noiseShape, imgShape)
# create the gan
gan = GAN(g, d)
# load image data
dataset = get_real_samples()
# train model
train(g, d, gan, dataset, noiseShape, 1000, 512)
```

Generative Adversarial Network (GAN)

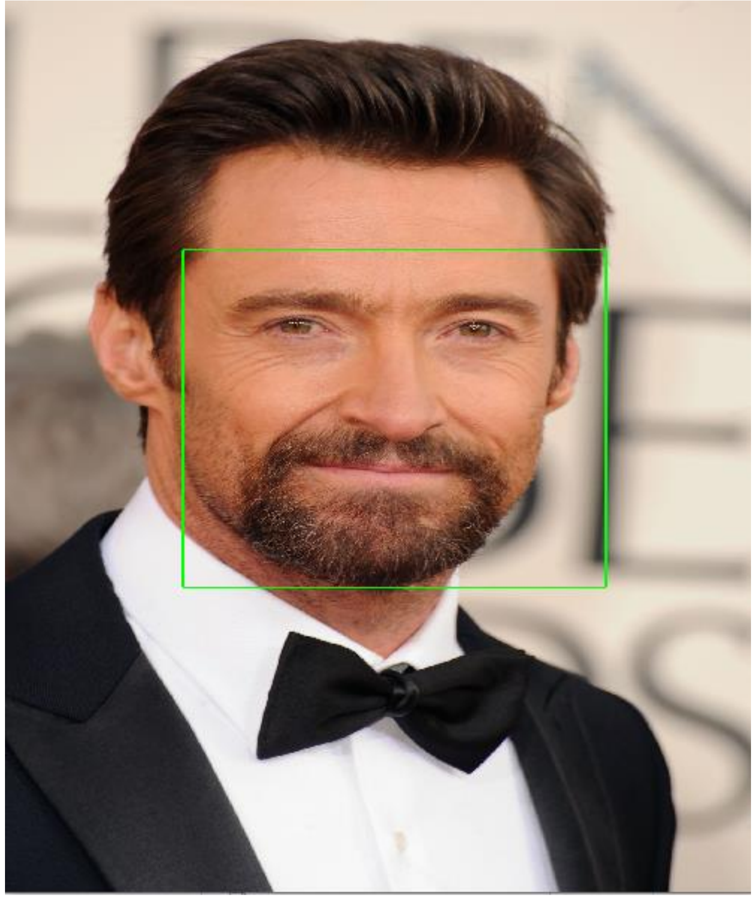
```python
def Discriminator(imgShape):

    model = Sequential()
    # normal
    model.add(Conv2D(64, (3,3), padding='same', input_shape=imgShape))
    model.add(LeakyReLU(alpha=0.2))
    # downsample
    model.add(Conv2D(128, (3,3), strides=(2,2), padding='same'))
    model.add(LeakyReLU(alpha=0.2))
    # downsample
    model.add(Conv2D(128, (3,3), strides=(2,2), padding='same'))
    model.add(LeakyReLU(alpha=0.2))
    # downsample
    model.add(Conv2D(256, (3,3), strides=(2,2), padding='same'))
    model.add(LeakyReLU(alpha=0.2))
    # classifier
    model.add(Flatten())
    model.add(Dropout(0.4))
    model.add(Dense(1, activation='sigmoid'))
    model.summary()
    # compile model
    opt = Adam(lr=0.0002, beta_1=0.5)
    model.compile(loss='binary_crossentropy', optimizer=opt, metrics=['accuracy'])

    return model
```

Some Fun Applications with dlib

```python
import cv2
import dlib

# show the image
img = cv2.imread("image1.jpg")

cv2.namedWindow("Resized", cv2.WINDOW_NORMAL)
cv2.resizeWindow("Resized", 500, 600)
cv2.imshow(winname="Resized", mat=img)
cv2.waitKey(delay=0)
cv2.destroyAllWindows()
```

Some Fun Applications with dlib

```
# read the image
img = cv2.imread("image1.jpg")


# Load the detector
detector = dlib.get_frontal_face_detector()

# Convert image into grayscale
gray = cv2.cvtColor(src=img, code=cv2.COLOR_BGR2GRAY)

# Use detector to find landmarks
faces = detector(gray)

for face in faces:
    x1 = face.left() # left point
    y1 = face.top() # top point
    x2 = face.right() # right point
    y2 = face.bottom() # bottom point
    # Draw a rectangle
    cv2.rectangle(img=img, pt1=(x1, y1), pt2=(x2, y2), color=(0, 255, 0), thickness=4)

cv2.namedWindow("Resized", cv2.WINDOW_NORMAL)
cv2.resizeWindow("Resized", 500, 600)
cv2.imshow(winname="Resized", mat=img)
cv2.waitKey(delay=0)
cv2.destroyAllWindows()
```



Some Fun Applications with dlib

```
import cv2
import numpy as np
import dlib

# Load the detector
detector = dlib.get_frontal_face_detector()

# Load the predictor
predictor = dlib.shape_predictor("shape_predictor_68_face_landmarks.dat")

# read the image
img = cv2.imread("image1.jpg")

# Convert image into grayscale
gray = cv2.cvtColor(src=img, code=cv2.COLOR_BGR2GRAY)

# Use detector to find landmarks
faces = detector(gray)
for face in faces:
    x1 = face.left() # left point
    y1 = face.top() # top point
    x2 = face.right() # right point
    y2 = face.bottom() # bottom point

    # Create landmark object
    landmarks = predictor(image=gray, box=face)

    # Loop through all the points
    for n in range(0, 68):
        x = landmarks.part(n).x
        y = landmarks.part(n).y

        # Draw a circle
        cv2.circle(img=img, center=(x, y), radius=3, color=(0, 255, 0), thickness=2)

cv2.namedWindow("Resized", cv2.WINDOW_NORMAL)
cv2.resizeWindow("Resized", 500, 600)
cv2.imshow(winname="Resized", mat=img)
cv2.waitKey(delay=0)
cv2.destroyAllWindows()
```

Some Fun Applications with dlib

```
import cv2
import dlib

# Load the detector
detector = dlib.get_frontal_face_detector()

# Load the predictor
predictor = dlib.shape_predictor("shape_predictor_68_face_landmarks.dat")

# read the image
cap = cv2.VideoCapture(0)

frame_width = int(cap.get(3))
frame_height = int(cap.get(4))
size = (frame_width, frame_height)
result = cv2.VideoWriter('filename.avi',
                         cv2.VideoWriter_fourcc(*'MJPG'),
                         10, size)
```

```
while True:
    _, frame = cap.read()
    # Convert image into grayscale
    gray = cv2.cvtColor(src=frame, code=cv2.COLOR_BGR2GRAY)

    # Use detector to find landmarks
    faces = detector(gray)
    for face in faces:
        x1 = face.left()  # left point
        y1 = face.top()  # top point
        x2 = face.right()  # right point
        y2 = face.bottom()  # bottom point
        # Create landmark object
        landmarks = predictor(image=gray, box=face)

        # Loop through all the points
        for n in range(0, 68):
            x = landmarks.part(n).x
            y = landmarks.part(n).y

            # Draw a circle
            cv2.circle(img=frame, center=(x, y), radius=3, color=(0, 255, 0), thickness=-1)
        result.write(frame)

    # show the image
    cv2.imshow(winname="Face", mat=frame)

    # Exit when escape is pressed
    if cv2.waitKey(delay=1) == 27:
        break

cap.release()
result.release()
cv2.destroyAllWindows()
```

Some Fun Applications with dlib

```
import cv2
import numpy as np
import dlib

# Load the detector
detector = dlib.get_frontal_face_detector()

# Load the predictor
predictor = dlib.shape_predictor("shape_predictor_68_face_landmarks.dat")

# read the image
img = cv2.imread("image1.jpg")

# Convert image into grayscale
gray = cv2.cvtColor(src=img, code=cv2.COLOR_BGR2GRAY)

# Use detector to find landmarks
faces = detector(gray)
for face in faces:
    x1 = face.left() # left point
    y1 = face.top() # top point
    x2 = face.right() # right point
    y2 = face.bottom() # bottom point

    # Create landmark object
    landmarks = predictor(image=gray, box=face)

    # Loop through all the points
    for n in range(0, 68):
        x = landmarks.part(n).x
        y = landmarks.part(n).y

        # Draw a circle
        cv2.circle(img=img, center=(x, y), radius=3, color=(0, 255, 0), thickness=2)

cv2.namedWindow("Resized", cv2.WINDOW_NORMAL)
cv2.resizeWindow("Resized", 500, 600)
cv2.imshow(winname="Resized", mat=img)
cv2.waitKey(delay=0)
cv2.destroyAllWindows()
```

Face Swap with dlib

```
# Adopted from https://www.kaggle.com/code/hamedetezadi/face-swap-using-dlib

import cv2
import numpy as np
import dlib
import time
import matplotlib.pyplot as plt
```

```
def extract_index_nparray(nparray):
    index = None
    for num in nparray[0]:
        index = num
        break
    return index
```

```
# Load the two images and convert it to gray image
img1 = cv2.imread("image3.jpg")
img1 = cv2.resize(img1, (250, 300))
img1_gray = cv2.cvtColor(img1, cv2.COLOR_BGR2GRAY)
mask = np.zeros_like(img1_gray)
cv2.namedWindow("Resized", cv2.WINDOW_NORMAL)
cv2.resizeWindow("Resized", 500, 600)
cv2.imshow(winname="Resized", mat=img1)
cv2.waitKey(delay=0)
cv2.destroyAllWindows()
```

```
# Load the two images and convert it to gray image
img2 = cv2.imread("images5.jpg")
img2 = cv2.resize(img2, (250, 300))
img2_gray = cv2.cvtColor(img2, cv2.COLOR_BGR2GRAY)
mask2 = np.zeros_like(img2_gray)
cv2.namedWindow("Resized", cv2.WINDOW_NORMAL)
cv2.resizeWindow("Resized", 500, 600)
cv2.imshow(winname="Resized", mat=img2)
cv2.waitKey(delay=0)
cv2.destroyAllWindows()
```

Face Swap with dlib

```
# Define face area detector using dlib
detector = dlib.get_frontal_face_detector()

#load the 68 face landmarks from dlib
predictor = dlib.shape_predictor("shape_predictor_68_face_landmarks.dat")

height, width, channels = img2.shape
img2_new_face = np.zeros((height, width, channels), np.uint8)
```

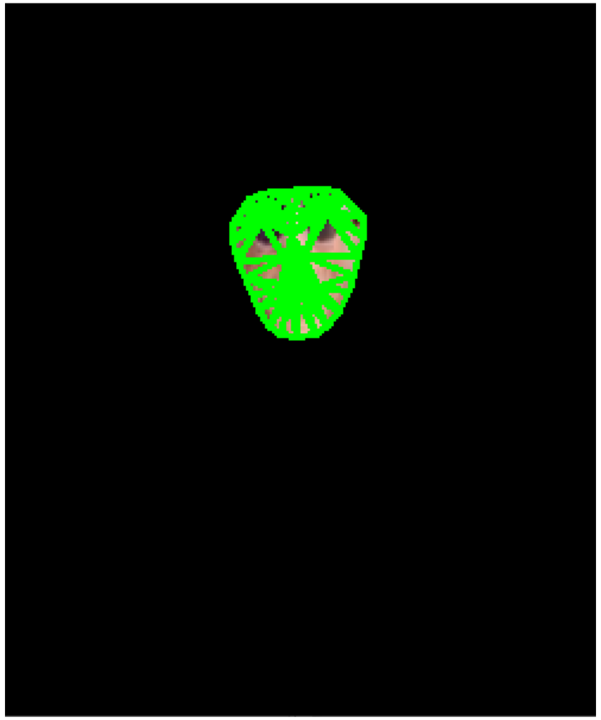
```
# Face 1
faces = detector(img1_gray)
for face in faces:
    x1 = face.left() # left point
    y1 = face.top() # top point
    x2 = face.right() # right point
    y2 = face.bottom() # bottom point
    # Draw a rectangle
    cv2.rectangle(img=img1_gray, pt1=(x1, y1), pt2=(x2, y2), color=(0, 255, 0), thickness=4)

cv2.namedWindow("Resized", cv2.WINDOW_NORMAL)
cv2.resizeWindow("Resized", 500, 600)
cv2.imshow(winname="Resized", mat=img1_gray)
cv2.waitKey(delay=0)
cv2.destroyAllWindows()
```

Face Swap with dlib

```
for face in faces:
    landmarks = predictor(img1_gray, face)
    landmarks_points = []
    for n in range(0, 68):
        x = landmarks.part(n).x
        y = landmarks.part(n).y
        landmarks_points.append((x, y))
        # Draw a circle
        cv2.circle(img=img1_gray, center=(x, y), radius=3, color=(0, 255, 0), thickness=2)


    cv2.namedWindow("Resized", cv2.WINDOW_NORMAL)
    cv2.resizeWindow("Resized", 500, 600)
    cv2.imshow(winname="Resized", mat=img1_gray)
    cv2.waitKey(delay=0)
    cv2.destroyAllWindows()


    points = np.array(landmarks_points, np.int32)
    convexhull = cv2.convexHull(points)
    # cv2.polylines(img, [convexhull], True, (255, 0, 0), 3)
    cv2.fillConvexPoly(mask, convexhull, 255)

    face_image_1 = cv2.bitwise_and(img1, img1, mask=mask)

    cv2.namedWindow("Resized", cv2.WINDOW_NORMAL)
    cv2.resizeWindow("Resized", 500, 600)
    cv2.imshow(winname="Resized", mat=face_image_1)
    cv2.waitKey(delay=0)
    cv2.destroyAllWindows()
```

Face Swap with dlib

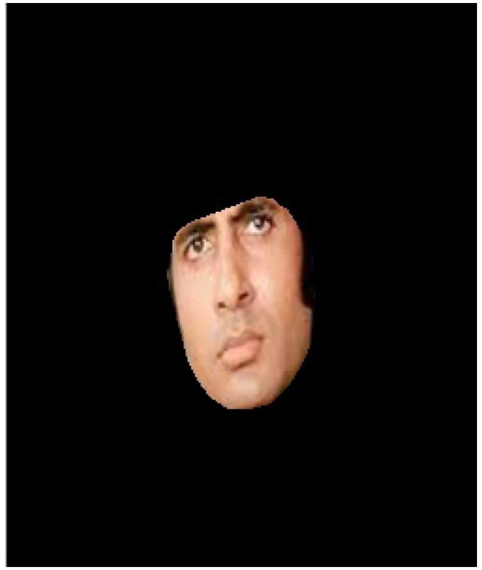
```python
# Delaunay triangulation
rect = cv2.boundingRect(convexhull)
subdiv = cv2.Subdiv2D(rect)
subdiv.insert(landmarks_points)
triangles = subdiv.getTriangleList()
triangles = np.array(triangles, dtype=np.int32)

indexes_triangles = []
for t in triangles:
    pt1 = (t[0], t[1])
    pt2 = (t[2], t[3])
    pt3 = (t[4], t[5])


    index_pt1 = np.where((points == pt1).all(axis=1))
    index_pt1 = extract_index_nparray(index_pt1)

    index_pt2 = np.where((points == pt2).all(axis=1))
    index_pt2 = extract_index_nparray(index_pt2)

    index_pt3 = np.where((points == pt3).all(axis=1))
    index_pt3 = extract_index_nparray(index_pt3)
    t_f1= face_image_1
    if index_pt1 is not None and index_pt2 is not None and index_pt3 is not None:
        triangle = [index_pt1, index_pt2, index_pt3]
        indexes_triangles.append(triangle)
        cv2.line(t_f1, pt1, pt2, (0, 255, 0), 2)
        cv2.line(t_f1, pt3, pt2, (0, 255, 0), 2)
        cv2.line(t_f1, pt1, pt3, (0, 255, 0), 2)

cv2.namedWindow("Resized", cv2.WINDOW_NORMAL)
cv2.resizeWindow("Resized", 500, 600)
cv2.imshow(winname="Resized", mat=t_f1)
cv2.waitKey(delay=0)
cv2.destroyAllWindows()
```



Face Swap with dlib

```python
# Face 2
faces2 = detector(img2_gray)
for face in faces2:
    landmarks = predictor(img2_gray, face)
    landmarks_points2 = []
    for n in range(0, 68):
        x = landmarks.part(n).x
        y = landmarks.part(n).y
        landmarks_points2.append((x, y))
        cv2.circle(img=img2_gray, center=(x, y), radius=3, color=(0, 255, 0), thickness=2)

    cv2.namedWindow("Resized", cv2.WINDOW_NORMAL)
    cv2.resizeWindow("Resized", 500, 600)
    cv2.imshow(winname="Resized", mat=img2_gray)
    cv2.waitKey(delay=0)
    cv2.destroyAllWindows()
```

```python
    points2 = np.array(landmarks_points2, np.int32)
    convexhull2 = cv2.convexHull(points2)

    # Display The Convexhull region
    cv2.fillConvexPoly(mask2, convexhull2, 255)
    c_img2_t=img2
    c_img2 = cv2.bitwise_and(c_img2_t, img2, mask=mask2)
    cv2.namedWindow("Resized", cv2.WINDOW_NORMAL)
    cv2.resizeWindow("Resized", 500, 600)
    cv2.imshow(winname="Resized", mat=c_img2)
    cv2.waitKey(delay=0)
    cv2.destroyAllWindows()
```



Face Swap with dlib

```
lines_space_mask = np.zeros_like(img1_gray)
lines_space_new_face = np.zeros_like(img2)


# Triangulation of both faces
for triangle_index in indexes_triangles:
    # Triangulation of the first face
    tr1_pt1 = landmarks_points[triangle_index[0]]
    tr1_pt2 = landmarks_points[triangle_index[1]]
    tr1_pt3 = landmarks_points[triangle_index[2]]
    triangle1 = np.array([tr1_pt1, tr1_pt2, tr1_pt3], np.int32)


    rect1 = cv2.boundingRect(triangle1)
    (x, y, w, h) = rect1
    cropped_triangle = img1[y: y + h, x: x + w]
    cropped_tr1_mask = np.zeros((h, w), np.uint8)



    points = np.array([[tr1_pt1[0] - x, tr1_pt1[1] - y],
                       [tr1_pt2[0] - x, tr1_pt2[1] - y],
                       [tr1_pt3[0] - x, tr1_pt3[1] - y]], np.int32)

    cv2.fillConvexPoly(cropped_tr1_mask, points, 255)
    #cv2.namedWindow("Resized", cv2.WINDOW_NORMAL)
    #cv2.resizeWindow("Resized", 500, 600)
    #cv2.imshow("Resized", cropped_tr1_mask)
    #cv2.waitKey(delay=0)
    #cv2.destroyAllWindows()
```

Face Swap with dlib

```
# Lines space
cv2.line(lines_space_mask, tr1_pt1, tr1_pt2, 255)
cv2.line(lines_space_mask, tr1_pt2, tr1_pt3, 255)
cv2.line(lines_space_mask, tr1_pt1, tr1_pt3, 255)
lines_space = cv2.bitwise_and(img1, img1, mask=lines_space_mask)
lines_space_t = lines_space
#cv2.namedWindow("Resized", cv2.WINDOW_NORMAL)
#cv2.resizeWindow("Resized", 500, 600)
#cv2.imshow("Resized", lines_space)
#cv2.waitKey(delay=0)
#cv2.destroyAllWindows()

# Triangulation of second face
tr2_pt1 = landmarks_points2[triangle_index[0]]
tr2_pt2 = landmarks_points2[triangle_index[1]]
tr2_pt3 = landmarks_points2[triangle_index[2]]
triangle2 = np.array([tr2_pt1, tr2_pt2, tr2_pt3], np.int32)


rect2 = cv2.boundingRect(triangle2)
(x, y, w, h) = rect2

cropped_tr2_mask = np.zeros((h, w), np.uint8)

points2 = np.array([[tr2_pt1[0] - x, tr2_pt1[1] - y],
                    [tr2_pt2[0] - x, tr2_pt2[1] - y],
                    [tr2_pt3[0] - x, tr2_pt3[1] - y]], np.int32)

cv2.fillConvexPoly(cropped_tr2_mask, points2, 255)
#cv2.namedWindow("Resized", cv2.WINDOW_NORMAL)
#cv2.resizeWindow("Resized", 500, 600)
#cv2.imshow("Resized", cropped_tr2_mask)
#cv2.waitKey(delay=0)
#cv2.destroyAllWindows()
```

Face Swap with dlib

```
    # Warp triangles
    points = np.float32(points)
    points2 = np.float32(points2)
    M = cv2.getAffineTransform(points, points2)
    warped_triangle = cv2.warpAffine(cropped_triangle, M, (w, h))
    warped_triangle = cv2.bitwise_and(warped_triangle, warped_triangle, mask=cropped_tr2_mask)
    #cv2.namedWindow("Resized", cv2.WINDOW_NORMAL)
    #cv2.resizeWindow("Resized", 500, 600)
    #cv2.imshow("Resized", warped_triangle)
    #cv2.waitKey(delay=0)
    #cv2.destroyAllWindows()

    # Reconstructing destination face
    img2_new_face_rect_area = img2_new_face[y: y + h, x: x + w]
    img2_new_face_rect_area_gray = cv2.cvtColor(img2_new_face_rect_area, cv2.COLOR_BGR2GRAY)
    _, mask_triangles_designed = cv2.threshold(img2_new_face_rect_area_gray, 1, 255, cv2.THRESH_BINARY_INV)
    warped_triangle = cv2.bitwise_and(warped_triangle, warped_triangle, mask=mask_triangles_designed)

    img2_new_face_rect_area = cv2.add(img2_new_face_rect_area, warped_triangle)
    img2_new_face[y: y + h, x: x + w] = img2_new_face_rect_area

cv2.namedWindow("Resized", cv2.WINDOW_NORMAL)
cv2.resizeWindow("Resized", 500, 600)
cv2.imshow("Resized", img2_new_face)
cv2.waitKey(delay=0)
cv2.destroyAllWindows()
```

Face Swap with dlib

```
# Face swapped (putting 1st face into 2nd face)
img2_face_mask = np.zeros_like(img2_gray)
img2_head_mask = cv2.fillConvexPoly(img2_face_mask, convexhull2, 255)
img2_face_mask = cv2.bitwise_not(img2_head_mask)


img2_head_noface = cv2.bitwise_and(img2, img2, mask=img2_face_mask)
result = cv2.add(img2_head_noface, img2_new_face)

(x, y, w, h) = cv2.boundingRect(convexhull2)
center_face2 = (int((x + x + w) / 2), int((y + y + h) / 2))

seamlessclone = cv2.seamlessClone(result, img2, img2_head_mask, center_face2, cv2.NORMAL_CLONE)

cv2.imshow("seamlessclone", seamlessclone)
cv2.waitKey(0)



cv2.destroyAllWindows()
```