

CSPE31 IMAGE PROCESSING

Mini Project

on

Image Recognition



DEPARTMENT OF COMPUTER SCIENCE AND
ENGINEERING NATIONAL INSTITUTE OF TECHNOLOGY
TIRUCHIRAPPALLI – 620 015

March 2020

submitted by

N. D. Jagan 106116058

G.Sai Charan 106116076

1. Problem Statement	3
2. Models Introduction	3
2.1 Vgg19	
2.2 Resnet	
2.3 Self-created model	
3. Datasets	9
3.1 Imagenet	
3.2 Cifar100	
4. Implementation.....	10
4.1 vgg19 and resnet50	
4.2 self-built model	
5. Results.....	11

Problem statement

Image recognition is the process of identifying and detecting an object or a feature in a digital image or video. This concept is used in many applications like systems for factory automation, toll booth monitoring, and security surveillance.

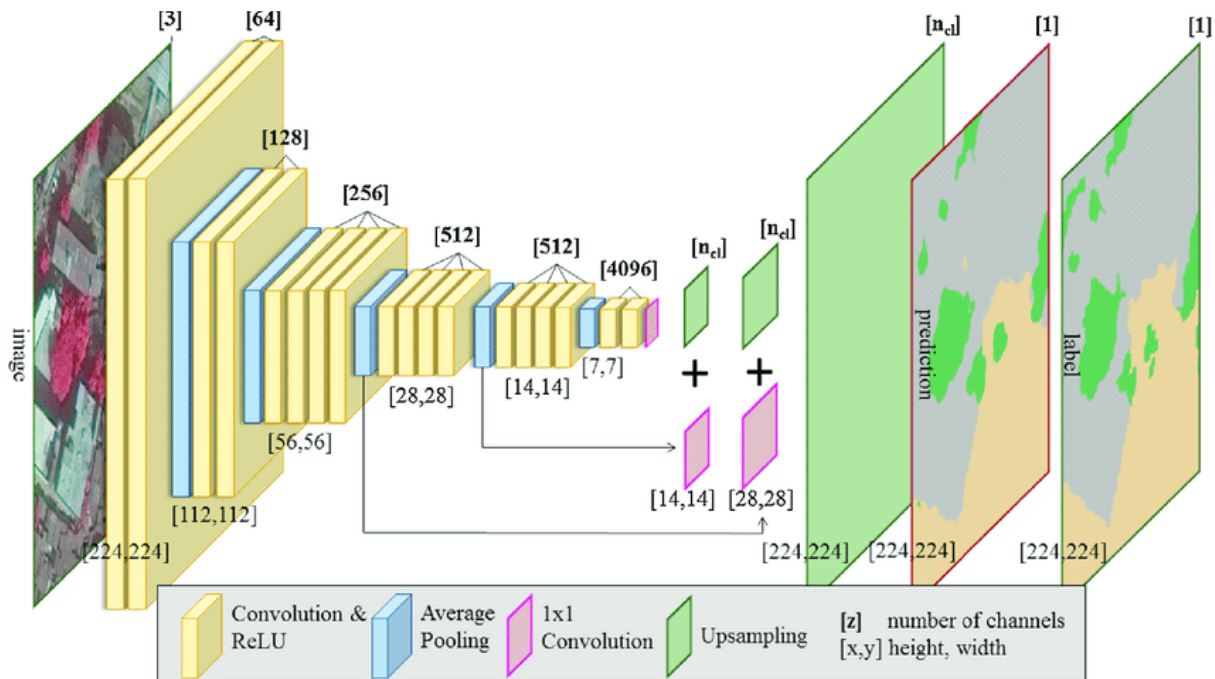
We have implemented it using 3 different models

- 1) Vgg19 model
- 2) Resnet model
- 3) We have created our own architecture and trained it using CIFAR100 dataset.

VGG19 model

VGG19 is a variant of VGG model which in short consists of **19** layers (16 convolution layers, 3 Fully connected layer, 5 Max Pool layers and 1 SoftMax layer).

Architecture



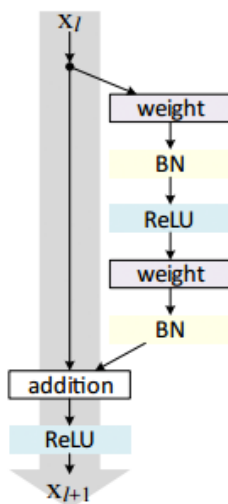
- A fixed size of $(224 * 224)$ RGB image was given as input to this network which means that the matrix was of shape $(224, 224, 3)$.
- The only pre-processing that was done is that they subtracted the mean RGB value from each pixel, computed over the whole training set.
- Used kernels of $(3 * 3)$ size with a stride size of 1 pixel, this enabled them to cover the whole notion of the image.
- Spatial padding was used to preserve the spatial resolution of the image.
- Max pooling was performed over a $2 * 2$ Pixel windows with stride of 2.
- This was followed by Rectified linear unit (ReLU) to introduce non-linearity to make the model classify better and to improve computational time as the previous models used tanh or sigmoid functions this proved much better than those.
- Implemented three fully connected layers from which first two were of size 4096 and after that a layer with 1000 channels for 1000-way *ILSVRC* classification and the final layer is a soft-max function.

RESNET50 Model

Residual Networks Unlike traditional sequential network architectures such as AlexNet, OverFeat, and VGG, ResNet is instead a form of “exotic architecture” that relies on micro-architecture modules (also called “network-in-network architectures”).

The term **micro-architecture** refers to the set of “building blocks” used to construct the network. A collection of micro-architecture building blocks (along with your standard CONV, POOL, etc. layers) leads to the *macro-architecture* (i.e the end network itself).

First introduced by He et al. in their 2015 paper, Deep Residual Learning for Image Recognition, the ResNet architecture has become a seminal work, demonstrating that extremely deep networks can be trained using standard SGD (and a reasonable initialization function) through the use of residual modules:



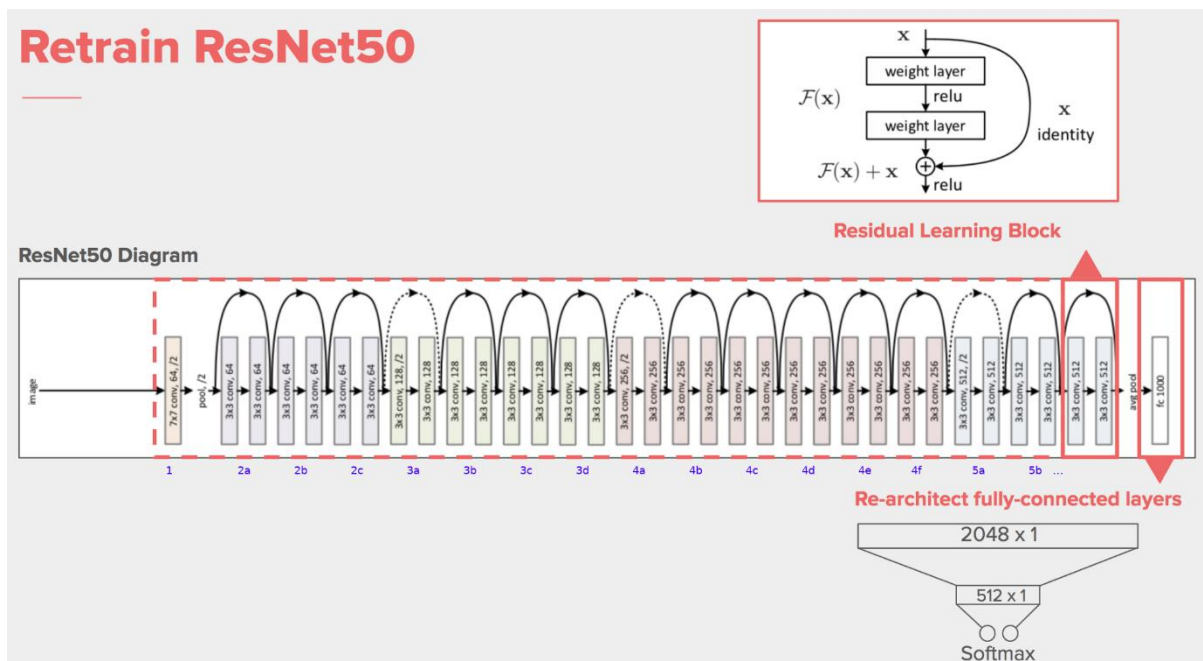
The residual module in ResNet as originally proposed by He et al. in 2015.

Even though ResNet is *much* deeper than VGG16 and VGG19, the model size is actually smaller due to the usage of global average pooling rather than fully-connected layers this reduces the model size down to 102MB for ResNet50.

Residual neural networks do this by utilizing skip connections, or shortcuts to jump over some layers. Typical ResNet models are implemented with double- or triple- layer skips that contain nonlinearities (ReLU) and batch normalization in between.

Architecture:

Retrain ResNet50



Self-built model architecture

Convolutional neural networks (CNNs) are the current state-of-the-art model architecture for image classification tasks. CNNs apply a series of filters to the raw pixel data of an image to extract and learn higher-level features, which the model can then use for classification. CNNs contains different components:

Convolutional layers, which apply a specified number of convolution filters to the image. For each subregion, the layer performs a set of mathematical operations to produce a single value in the output feature map. Convolutional layers then typically apply a ReLU activation function) to the output to introduce nonlinearities into the model.

Batch Normalization layers, Batch Normalization is a technical trick to make training faster.

Dropout layers, Dropout is a regularization method, where the layer randomly replaces a proportion of its weights to zero for each training sample. This forces the net to learn features in a distributed way, not relying too much on a weight, and therefore improves generalization.

Activation layers, Activation functions are important for any Neural Network to learn and make sense of something complicated and Non-linear complex functional mappings between the inputs and response variable. They introduce non-linear properties to our Network.

Pooling layers, which down sample the image data extracted by the convolutional layers to reduce the dimensionality of the feature map in order to decrease processing time. A commonly used pooling algorithm is max pooling, which extracts subregions of the feature map (e.g., 2x2-pixel tiles), keeps their maximum value, and discards all other values. it gives rotation invariant feature extraction ability to model.

Dense (fully connected) layers, which perform classification on the features extracted by the convolutional layers and down sampled by the pooling layers. In a dense layer, every node in the layer is connected to every node in the preceding layer.

Typically, a CNN is composed of a stack of convolutional modules that perform feature extraction. Each module consists of a convolutional layer followed by a pooling layer. The last convolutional module is followed by one or more dense layers that perform classification. The final dense layer in a CNN contains a single node for each target class in the model (all the possible classes the model may predict), with a soft max activation function to generate a value between 0–1 for each node (the sum of all these soft max values is equal to 1). We can interpret the soft max values for a given image as relative measurements of how likely it is that the image falls into each target class.

We are going to use below architecture model to classify the images with **CIFAR-100 dataset**

Model: "sequential_1"

Layer (type)	Output Shape	Param #
=====		
conv2d_1 (Conv2D)	(None, 32, 32, 32)	896
activation_1 (Activation)	(None, 32, 32, 32)	0
dropout_1 (Dropout)	(None, 32, 32, 32)	0
batch_normalization_1 (Batch Normalization)	(None, 32, 32, 32)	128
conv2d_2 (Conv2D)	(None, 32, 32, 64)	18496

activation_2 (Activation) (None, 32, 32, 64) 0

max_pooling2d_1 (MaxPooling2 (None, 16, 16, 64) 0

dropout_2 (Dropout) (None, 16, 16, 64) 0

batch_normalization_2 (Batch (None, 16, 16, 64) 256

conv2d_3 (Conv2D) (None, 16, 16, 64) 36928

activation_3 (Activation) (None, 16, 16, 64) 0

max_pooling2d_2 (MaxPooling2 (None, 8, 8, 64) 0

dropout_3 (Dropout) (None, 8, 8, 64) 0

batch_normalization_3 (Batch (None, 8, 8, 64) 256

conv2d_4 (Conv2D) (None, 8, 8, 128) 73856

activation_4 (Activation) (None, 8, 8, 128) 0

dropout_4 (Dropout) (None, 8, 8, 128) 0

batch_normalization_4 (Batch (None, 8, 8, 128) 512

flatten_1 (Flatten)	(None, 8192)	0
dropout_5 (Dropout)	(None, 8192)	0
dense_1 (Dense)	(None, 256)	2097408
activation_5 (Activation)	(None, 256)	0
dropout_6 (Dropout)	(None, 256)	0
batch_normalization_5 (Batch Normalization)	(None, 256)	1024
dense_2 (Dense)	(None, 128)	32896
activation_6 (Activation)	(None, 128)	0
dropout_7 (Dropout)	(None, 128)	0
batch_normalization_6 (Batch Normalization)	(None, 128)	512
dense_3 (Dense)	(None, 100)	12900
activation_7 (Activation)	(None, 100)	0
=====		

Total params: 2,276,068

Trainable params: 2,274,724

Non-trainable params: 1,344

ImageNet Dataset

ImageNet is formally a project aimed at (manually) labelling and categorizing images into almost 22,000 separate object categories for the purpose of computer vision research.

However, when we hear the term “ImageNet” in the context of deep learning and Convolutional Neural Networks, we are likely referring to the ILSVRC .

The goal of this image classification challenge is to train a model that can correctly classify an input image into 1,000 separate object categories.

Models are trained on ~1.2 million training images with another 50,000 images for validation and 100,000 images for testing.

These 1,000 image categories represent object classes that we encounter in our day-to-day lives, such as species of dogs, cats, various household objects, vehicle types, and much more. You can find the full list of object categories in the ILSVRC challenge [here](#).

CIFAR100

This dataset is just like the CIFAR-10, except it has 100 classes containing 600 images each. There are 500 training images and 100 testing images per class. The 100 classes in CIFAR-100 are grouped into 20 super classes. Each image comes with a fine label (the class to which it belongs to) and a coarse label (the superclass to which it belongs to)

Implementation using vgg19 and resnet50

Requirements:

Python3, keras, numpy, cv2

```
import keras
import numpy as np
import cv2
from google.colab.patches import cv2_imshow
```

```

from keras.applications import VGG19
from keras.applications import ResNet50
from keras.applications.vgg19 import preprocess_input
from keras.applications.resnet50 import preprocess_input
from keras.preprocessing.image import img_to_array, load_img
from keras.applications import imagenet_utils

```

The below function is used to return feature vectors of an image using vgg19 model and imagenet dataset. Here Image pre-processing is done using preprocess_input library from keras

```

def get_image_vector_vgg19(img_path):
    vgg19_model = VGG19(weights='imagenet')
    img = load_img(img_path, target_size=(224, 224))
    img = img_to_array(img)
    img = np.expand_dims(img, axis=0)
    img = preprocess_input(img)
    img_vec= vgg19_model.predict(img)
    return img_vec

```

The below function is used to return feature vectors of an image using Resnet model and weights from imagenet

```

def get_image_vector_resnet50(img_path):
    resnet50_model = ResNet50(weights='imagenet')
    img = load_img(img_path, target_size=(224, 224))
    img = img_to_array(img)
    img = np.expand_dims(img, axis=0)
    img = preprocess_input(img)
    img_vec= resnet50_model.predict(img)
    return img_vec

```

This function is used to decode feature vectors and prints top 5 probable predictions

```

def decode_display_predictions(img_path,img_vec):
    predictions = imagenet_utils.decode_predictions(img_vec)
    img = cv2.imread(img_path, cv2.IMREAD_UNCHANGED)
    cv2_imshow(img)
    for (i, (imagenetID, label, prob)) in enumerate(predictions[0]):
        print("{}: {}: {:.2f}%".format(i + 1, label, prob * 100))




```

Call the above functions to predict and recognise object in image

```
decode_display_predictions('Mountain.jpg',get_image_vector_vgg19('Mountain.jpg'))
```

Results

Each image's vgg19 results following by resnet50 results are compared side by side

 <p>1. sports_car: 24.80% 2. racer: 14.15% 3. ambulance: 10.67% 4. cab: 6.15% 5. fire_engine: 5.33%</p>	 <p>1. sports_car: 62.51% 2. racer: 24.21% 3. cab: 7.47% 4. convertible: 2.74% 5. ambulance: 1.01%</p>
 <p>1. desktop_computer: 99.91% 2. monitor: 0.03% 3. screen: 0.02% 4. notebook: 0.01% 5. mouse: 0.01%</p>	 <p>1. desktop_computer: 97.19% 2. loudspeaker: 1.03% 3. monitor: 0.38% 4. screen: 0.37% 5. radio: 0.29%</p>



```
1. lab_coat: 92.09%
2. stethoscope: 6.65%
3. notebook: 0.14%
4. trench_coat: 0.08%
5. web_site: 0.07%
```



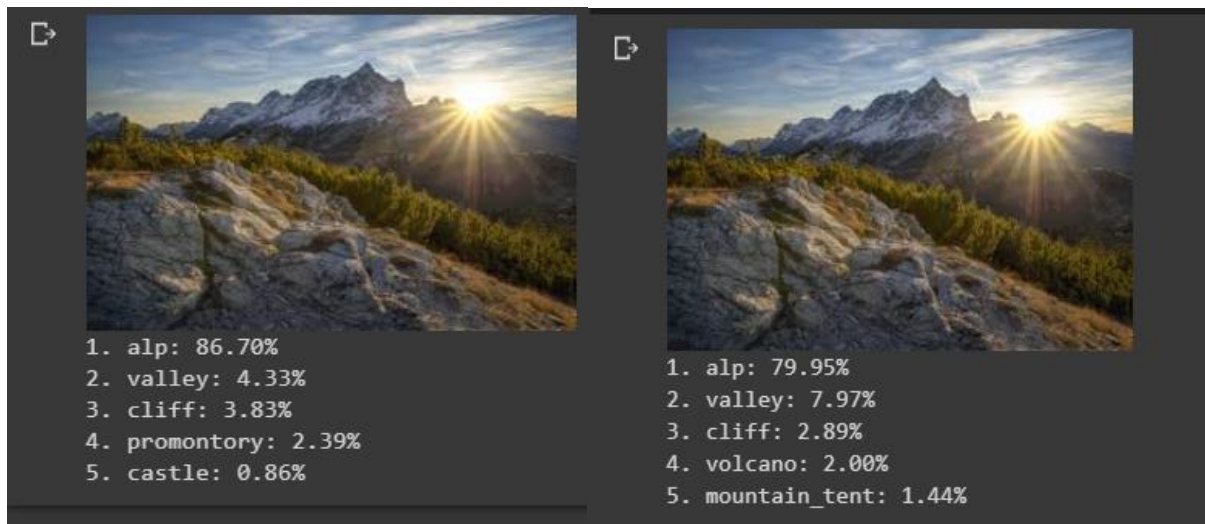
```
1. lab_coat: 99.17%
2. stethoscope: 0.77%
3. pajama: 0.01%
4. hand-held_computer: 0.00%
5. Windsor_tie: 0.00%
```



```
1. Dandie_Dinmont: 65.70%
2. Tibetan_terrier: 13.76%
3. Norfolk_terrier: 6.54%
4. otterhound: 4.49%
5. soft-coated_wheaten_terrier: 1.97%
```



```
1. Dandie_Dinmont: 43.95%
2. otterhound: 19.17%
3. Tibetan_terrier: 15.55%
4. Border_terrier: 7.11%
5. Norfolk_terrier: 6.21%
```



Implementation using self-built model

```
import numpy
from keras.models import Sequential
from keras.layers import Dense, Dropout, Flatten, BatchNormalization, Activation
from keras.layers.convolutional import Conv2D, MaxPooling2D
from keras.constraints import maxnorm
from keras.utils import np_utils
import cv2
from google.colab.patches import cv2_imshow
from keras.datasets import cifar100
from keras.preprocessing.image import img_to_array, load_img
```

Now let's load cifar100 dataset followed by pre-processing it. If the values of the input data are in too wide a range it can negatively impact how the network performs. In this case, the input values are the pixels in the image, which have a value between 0 to 255.

In order to normalize the data we can simply divide the image values by 255. To do this we first need to make the data a float type, since they are currently integers.

We are effectively doing binary classification here because an image either belongs to one class or it does not, it can't fall somewhere in-between. The Numpy command `to_categorical()` is used to one-hot encode. This is why we imported the `np_utils` function from Keras, as it contains `to_categorical()`

```
(X_train, y_train), (X_test, y_test) = cifar100.load_data()
X_train = X_train.astype('float32')
X_test = X_test.astype('float32')
X_train = X_train / 255.0
X_test = X_test / 255.0
```

```
y_train = np_utils.to_categorical(y_train)
y_test = np_utils.to_categorical(y_test)
class_num = y_test.shape[1]
```

Sequential model is designed here, the optimizer is what will tune the weights in your network to approach the point of lowest loss. The Adam algorithm is one of the most used optimizers. Finally, the softmax activation function selects the neuron with the highest probability as its output, voting that the image belongs to that class:

```
model = Sequential()
model.add(Conv2D(32, (3, 3), input_shape=X_train.shape[1:], padding='same'))
model.add(Activation('relu'))
model.add(Dropout(0.2))
model.add(BatchNormalization())

model.add(Conv2D(64, (3, 3), padding='same'))
model.add(Activation('relu'))
model.add(MaxPooling2D(pool_size=(2, 2)))
model.add(Dropout(0.2))
model.add(BatchNormalization())

model.add(Conv2D(64, (3, 3), padding='same'))
model.add(Activation('relu'))
model.add(MaxPooling2D(pool_size=(2, 2)))
model.add(Dropout(0.2))
model.add(BatchNormalization())

model.add(Conv2D(128, (3, 3), padding='same'))

model.add(Activation('relu'))
model.add(Dropout(0.2))
model.add(BatchNormalization())
model.add(Flatten())
model.add(Dropout(0.2))

model.add(Dense(256, kernel_constraint=maxnorm(3)))
model.add(Activation('relu'))
model.add(Dropout(0.2))
model.add(BatchNormalization())
model.add(Dense(128, kernel_constraint=maxnorm(3)))
model.add(Activation('relu'))
model.add(Dropout(0.2))
model.add(BatchNormalization())
model.add(Dense(class_num))
model.add(Activation('softmax'))
```



```
epochs = 25
optimizer = 'adam'
model.compile(loss='categorical_crossentropy', optimizer=optimizer, metrics=['accuracy'])
print(model.summary())
```

We have used 25 epochs and took seed value as 21

Train on 50000 samples, validate on 10000 samples

Running this piece of code will yield:

```
model.fit(X_train, y_train, validation_data=(X_test, y_test), epochs=10, batch_size=64)
```

Epoch 1/25

50000/50000 [=====] - 414s 8ms/step - loss: 1.5080 - accuracy: 0.4645 - val_loss: 1.1620 - val_accuracy: 0.5785

Epoch 2/25

50000/50000 [=====] - 413s 8ms/step - loss: 1.0310 - accuracy: 0.6340 - val_loss: 0.8667 - val_accuracy: 0.6936

Epoch 3/25

50000/50000 [=====] - 412s 8ms/step - loss: 0.8531 - accuracy: 0.6988 - val_loss: 0.7937 - val_accuracy: 0.7210

Goes on till epoch 25

Epoch 24/25

50000/50000 [=====] - 408s 8ms/step - loss: 0.4199 - accuracy: 0.8531 - val_loss: 0.5387 - val_accuracy: 0.8159

Epoch 25/25

50000/50000 [=====] - 407s 8ms/step - loss: 0.4085 - accuracy: 0.8553 - val_loss: 0.4995 - val_accuracy: 0.8307

<keras.callbacks.callbacks.History at 0x7f44376c8908>

```
scores = model.evaluate(X_test, y_test, verbose=0)
print("Accuracy: %.2f%%" % (scores[1]*100))
```

Accuracy: 87.16%

```
def get_img_vec(img_path):  
    img = load_img(img_path, target_size=(32, 32))  
    img = img_to_array(img)  
    img = np.expand_dims(img, axis=0)  
    img_vec= model.predict(img)  
    return img_vec
```

```
def predictions_display(img_path,img_vec):  
    img = cv2.imread(img_path, cv2.IMREAD_UNCHANGED)  
    cv2.imshow(img)  
    count=-1  
    for i in img_vec[0]:  
        count=count+1  
        if i > 0.0:  
            print('i guess it is '+load_labels[count])
```

```
▶ predictions_display('Computer.jpg',get_img_vec('Computer.jpg'))
```



```
i guess it is Computer Desktop  
i guess it is Oak
```



```
[120] predictions_display('Mountain.jpg',get_img_vec('Mountain.jpg'))
```



```
i guess it is Mountain  
i guess it is Castle  
i guess it is Kangaroo  
i guess it is Porcupine  
i guess it is Rabbit
```