ARTIFICIAL INTELLIGENCE (CO-304)

CNN AND ITS APPLICATIONS IN THE FIELD OF AI

Submitted To:

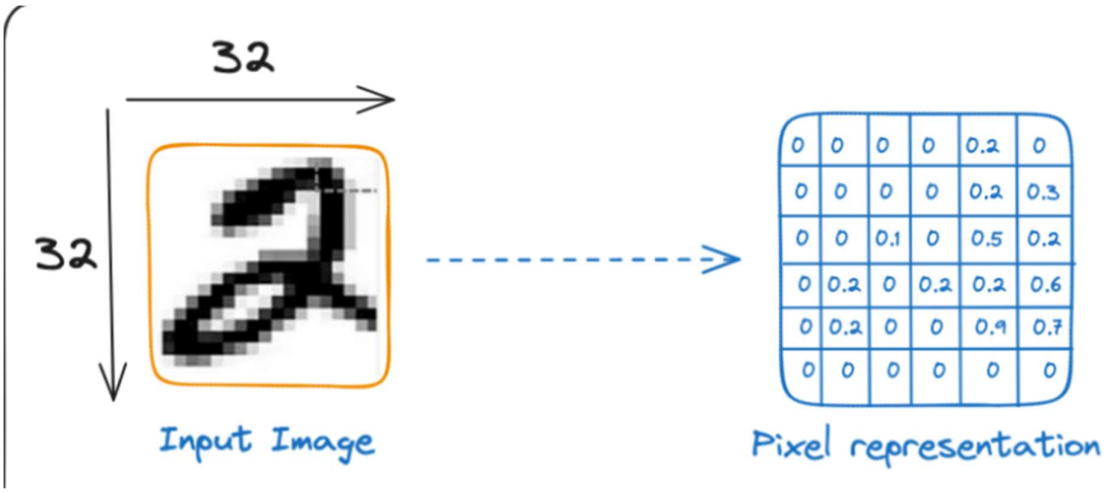
Mr. Himanshu Nandanwar

Submitted By:

Anish Sinha

2K21/MC/22

Introduction to Convolutional Neural Networks (CNNs)



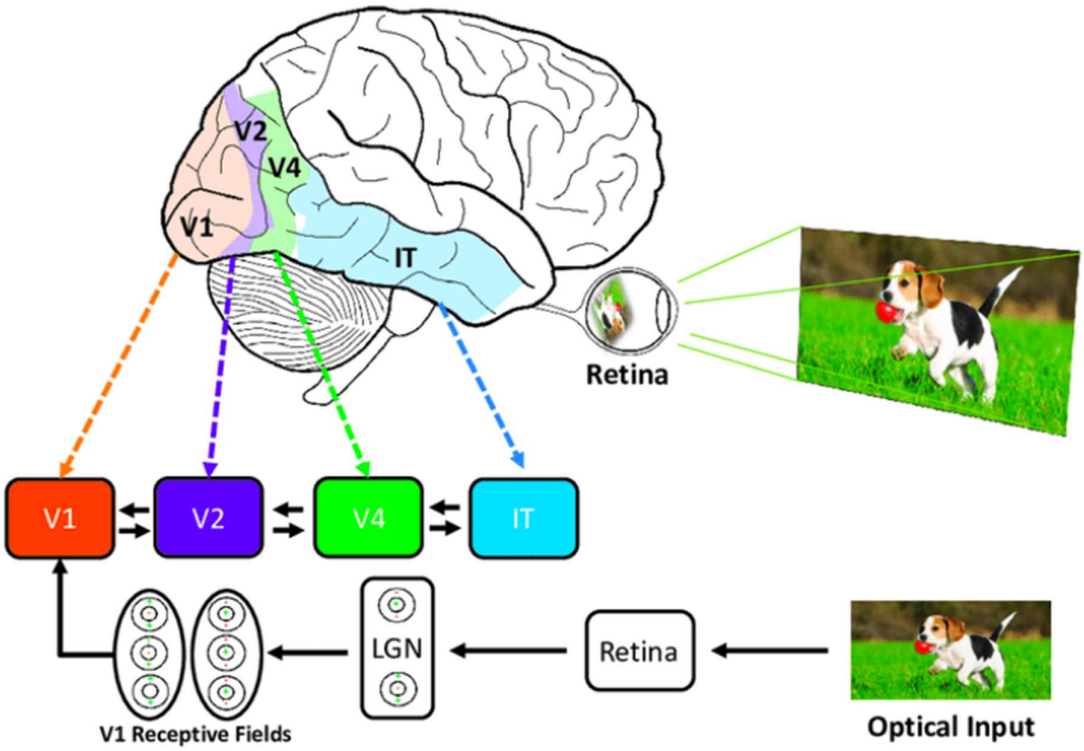
Convolutional Neural Networks (CNNs), also known as ConvNets, are a class of deep learning algorithms specifically designed for processing structured grid data, such as images. CNNs have emerged as a cornerstone in the field of artificial intelligence, particularly in tasks involving image recognition, classification, and object detection. The significance of CNNs lies in their ability to autonomously extract and learn hierarchical patterns and features directly from raw pixel data, bypassing the need for manual feature engineering.

Importance of CNNs

CNNs play a pivotal role in various real-world applications, ranging from autonomous vehicles and medical imaging to security surveillance systems and facial recognition technology. Their importance stems from several key factors:

1. **Automated Feature Extraction:** Unlike traditional machine learning algorithms that rely on handcrafted features, CNNs can automatically learn and extract features from raw data, leading to more efficient and accurate models.
2. **Translation-Invariant Properties:** Inspired by the human visual system, CNNs exhibit translation-invariant properties, enabling them to recognize and extract features from images regardless of variations in position, orientation, or scale.
3. **Performance and Scalability:** CNNs have demonstrated state-of-the-art performance in various benchmark datasets and can scale effectively to handle large-scale image datasets with millions of samples.
4. **Versatility:** While initially developed for image-related tasks, CNNs have found applications in other domains such as natural language processing, time series analysis, and even audio processing, showcasing their versatility and adaptability.

Inspiration from Human Visual System

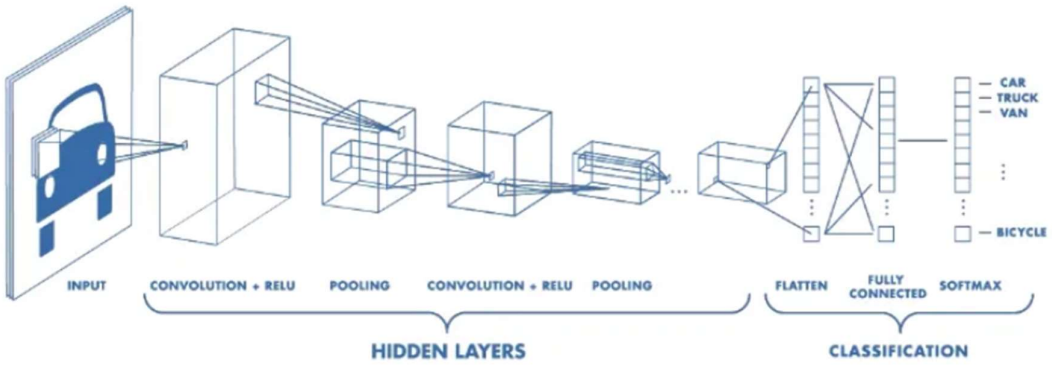


CNNs draw inspiration from the hierarchical architecture and processing mechanisms of the human visual system, particularly the visual cortex. Key parallels between CNNs and the human visual system include:

1. **Hierarchical Architecture:** Both CNNs and the visual cortex consist of multiple layers of interconnected neurons, with each layer processing increasingly complex features.
2. **Local Connectivity:** Neurons in both CNNs and the visual cortex exhibit local connectivity, where they are selectively connected to nearby neurons, enabling efficient information processing.
3. **Translation Invariance:** CNNs, like the visual cortex, demonstrate translation-invariant properties, allowing them to recognize and extract features from images irrespective of their spatial location.

By mimicking the underlying principles of the human visual system, CNNs achieve remarkable efficiency and effectiveness in processing visual data, making them indispensable in modern AI systems.

Key Components of Convolutional Neural Networks (CNNs)



CNNs consist of several interconnected layers, each serving a specific function in the overall architecture. The key components of CNNs include:

1. **Convolutional Layers**:

Convolutional layers are the fundamental building blocks of CNNs responsible for feature extraction. These layers apply learnable filters (also known as kernels) to input images, sliding them across the image and computing dot products to produce feature maps. Each filter learns to detect specific patterns or features, such as edges, textures, or shapes, within the input data.

2. **Activation Functions**:

Activation functions introduce non-linearity into the network, enabling it to learn complex relationships between features. ReLU (Rectified Linear Unit) is one of the most commonly used activation functions in CNNs, as it efficiently addresses the vanishing gradient problem and accelerates convergence during training.

3. **Pooling Layers**:

Pooling layers are used to reduce the spatial dimensions of the feature maps generated by convolutional layers. They aggregate information from local regions of the feature maps, such as taking the maximum or average value, to create downsampled representations. Pooling layers help in reducing computational complexity, extracting invariant features, and enhancing the model's robustness to variations in input data.

4. **Fully Connected Layers**:

Fully connected layers, also known as dense layers, are typically found at the end of CNN architectures. These layers connect every neuron from the previous layer to every neuron in the subsequent layer, allowing the network to learn complex mappings between extracted features and output labels. Fully connected layers are commonly used for classification tasks, where they produce the final predictions based on the learned features.

Convolution Operation:

The convolution operation performed in convolutional layers involves sliding a filter (kernel) over the input image and computing the dot product between the filter weights and the

corresponding pixel values. This process generates feature maps that represent the presence of specific patterns or features in different spatial locations of the input image. The convolution operation is the core mechanism through which CNNs extract meaningful information from raw pixel data.

Pooling Operations:

Pooling operations, such as max pooling or average pooling, are applied to feature maps generated by convolutional layers. These operations reduce the spatial dimensions of the feature maps by downsampling them, thereby decreasing the computational load and extracting invariant features. Max pooling, for example, selects the maximum value from a local region of the feature map, emphasizing the most significant features while discarding irrelevant details.

Training Convolutional Neural Networks (CNNs)

Training a CNN involves optimizing its parameters (weights and biases) to minimize a chosen loss function, thereby improving its ability to make accurate predictions on unseen data. The training process typically consists of the following steps:

1. **Forward Propagation**:

During forward propagation, input data is passed through the network, layer by layer, to generate predictions. Each layer performs computations on the input data using its parameters (weights and biases) and activation functions, producing output activations that serve as input to the subsequent layer. The final layer's output is compared to the ground truth labels using a chosen loss function to quantify the prediction error.

2. **Loss Computation**:

The loss function measures the discrepancy between the predicted outputs and the ground truth labels. Common loss functions used in classification tasks include cross-entropy loss and mean squared error. The goal of training is to minimize this loss function by adjusting the network's parameters through optimization algorithms.

3. **Backpropagation**:

Backpropagation is the core algorithm for training neural networks, including CNNs. It involves computing the gradients of the loss function with respect to the network's parameters using the chain rule of calculus. These gradients indicate the direction and magnitude of parameter updates required to reduce the loss. The gradients are propagated backward through the network, layer by layer, allowing each layer to adjust its parameters based on the error signals received from subsequent layers.

4. **Parameter Update**:

Once the gradients have been computed through backpropagation, the network's parameters are updated using optimization algorithms such as stochastic gradient descent (SGD), Adam,

or RMSprop. These algorithms adjust the parameters in the direction that minimizes the loss function, with the learning rate controlling the size of the parameter updates.

5. **Iteration**:

The forward propagation, loss computation, backpropagation, and parameter update steps are repeated iteratively over multiple epochs (passes through the entire training dataset). Each iteration refines the network's parameters, gradually improving its performance on the training data.

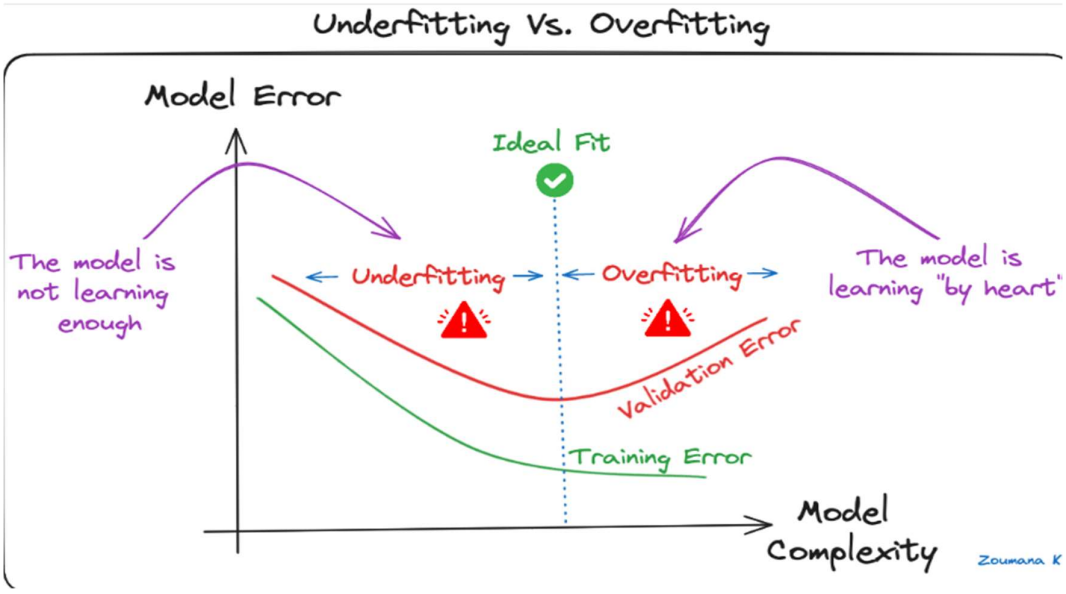
6. **Validation and Testing**:

Throughout the training process, the performance of the CNN is evaluated on validation data, which is separate from the training data. This allows monitoring for overfitting and selecting the best-performing model based on validation performance. Finally, the trained model is evaluated on a separate test dataset to assess its generalization performance.

Overfitting and Regularization in CNNs

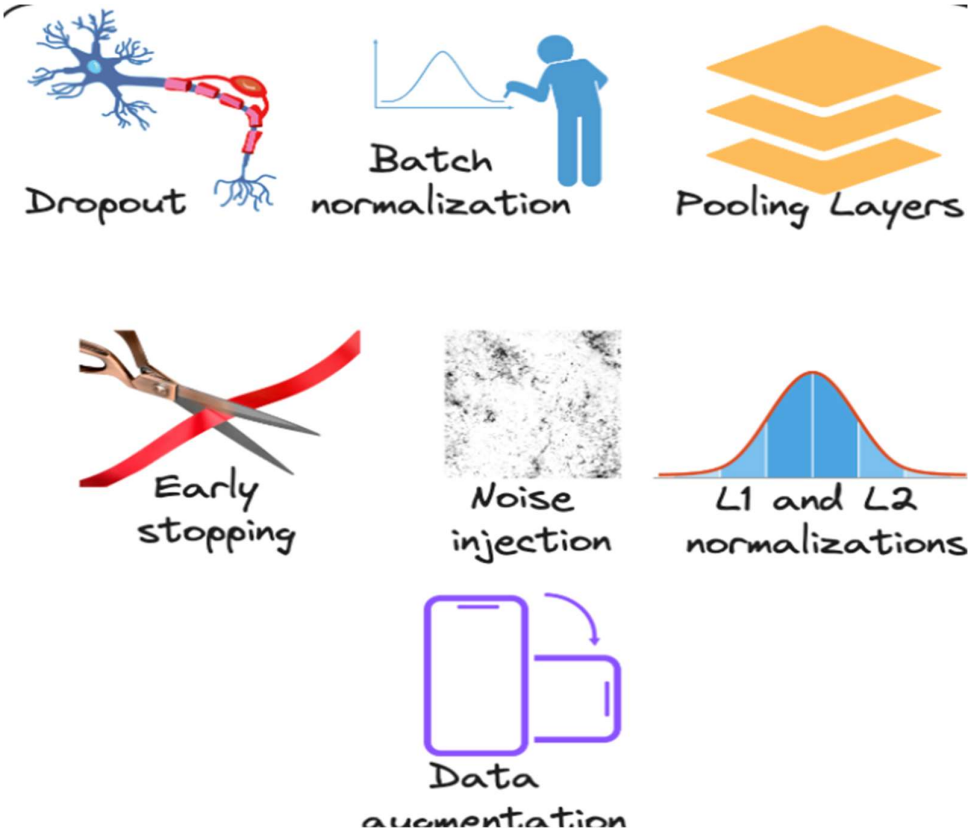
Overfitting is a common challenge in machine learning models and CNN deep learning projects. It happens when the model learns the training data too well ("learning by heart"), including its noise and outliers. Such a learning leads to a model that performs well on the training data but badly on new, unseen data.

This can be observed when the performance on training data is too low compared to the performance on validation or testing data, and a graphical illustration is given below:



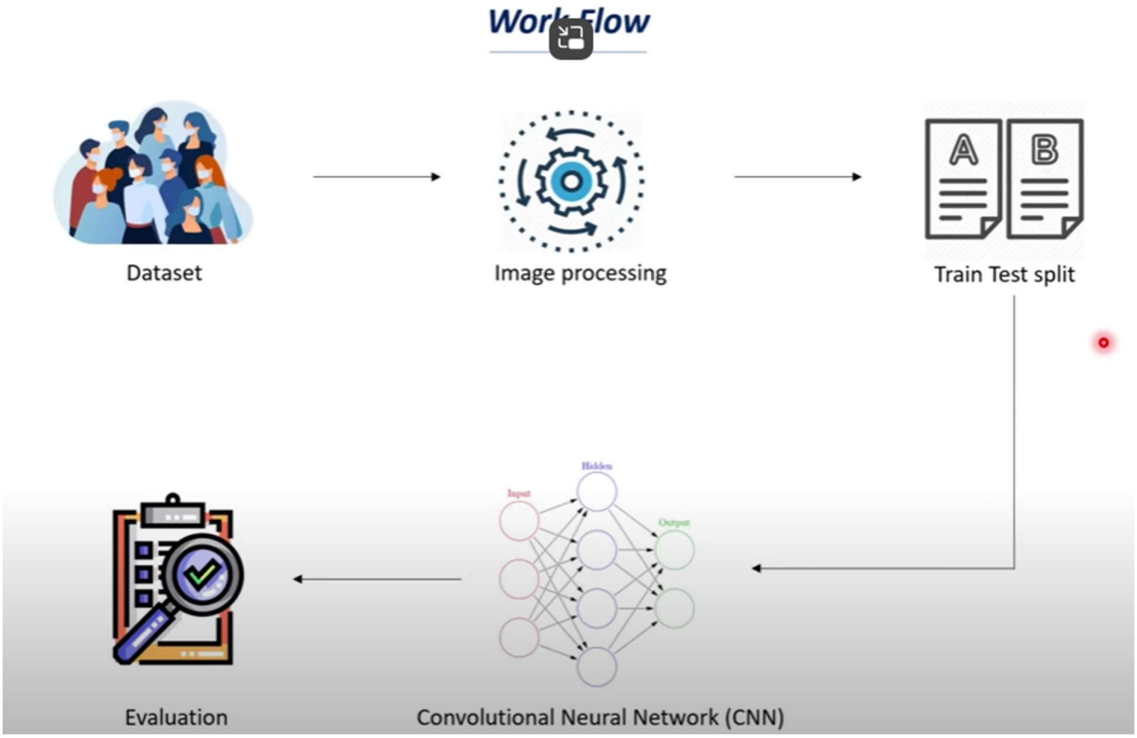
Deep learning models, especially Convolutional Neural Networks (CNNs), are particularly susceptible to overfitting due to their capacity for high complexity and their ability to learn detailed patterns in large-scale data.

Several regularization techniques can be applied to mitigate overfitting in CNNs, and some are illustrated below:



- **Dropout:** This consists of randomly dropping some neurons during the training process, which forces the remaining neurons to learn new features from the input data.
- **Batch normalization:** The overfitting is reduced at some extent by normalizing the input layer by adjusting and scaling the activations. This approach is also used to speed up and stabilize the training process.
- **Pooling Layers:** This can be used to reduce the spatial dimensions of the input image to provide the model with an abstracted form of representation, hence reducing the chance of overfitting.
- **Early stopping:** This consists of consistently monitoring the model's performance on validation data during the training process and stopping the training whenever the validation error does not improve anymore.
- **Noise injection:** This process consists of adding noise to the inputs or the outputs of hidden layers during the training to make the model more robust and prevent it from a weak generalization.
- **L1 and L2 normalizations:** Both L1 and L2 are used to add a penalty to the loss function based on the size of weights. More specifically, L1 encourages the weights to be spare, leading to better feature selection. On the other hand, L2 (also called weight decay) encourages the weights to be small, preventing them from having too much influence on the predictions.
- **Data augmentation:** This is the process of artificially increasing the size and diversity of the training dataset by applying random transformations like rotation, scaling, flipping, or cropping to the input images.

WORKFLOW OF FACE MASK DETECTION PROJECT:



Building a face mask detection system using Convolutional Neural Networks (CNNs) involves several steps. Here's a simplified guide to creating one:

1. Data Collection:

- Gather a dataset containing images of people with and without masks.
- Ensure the dataset is diverse, balanced, and annotated with labels indicating whether each person is wearing a mask.

2. Data Preprocessing:

- Resize all images to a uniform size, typically square dimensions like 128x128 pixels.
- Convert images to grayscale or RGB format depending on your model requirements.
- Normalize pixel values to the range [0, 1].

3. Data Augmentation (Optional):

- Augment your dataset by applying transformations like rotation, zoom, and horizontal flips to increase its size and diversity.
- Data augmentation helps prevent overfitting and improves model generalization.

4. Model Architecture:

- Define a CNN architecture suitable for image classification tasks.

- Typical architectures include a series of convolutional layers followed by pooling layers, and then fully connected layers.
- Add dropout layers to prevent overfitting.

5. Model Compilation:

- Compile your model with an appropriate optimizer (e.g., Adam), a loss function (e.g., binary cross-entropy for binary classification), and evaluation metrics (e.g., accuracy).

6. Model Training:

- Split your dataset into training and validation sets.
- Train your model on the training data using the **fit()** function.
- Monitor training progress by observing loss and accuracy metrics on both training and validation sets.
- Tweak hyperparameters as needed based on training performance.

7. Model Evaluation:

- Evaluate your model's performance on a separate test dataset using the **evaluate()** function.
- Check metrics like accuracy, precision, recall, and F1-score to assess model performance.

8. Model Deployment:

- Deploy your trained model to a production environment where it can make real-time predictions.
- Integrate the model into an application or system that captures images and processes them for mask detection.
- Implement a user interface to display the results of mask detection.

9. Continuous Improvement:

- Monitor the model's performance in the production environment.
- Collect feedback and data to iteratively improve the model.
- Consider retraining the model with new data periodically to keep it up-to-date.

THE PROJECT:

Importing Necessary Libraries and Downloading Dataset

- The code starts by installing the Kaggle API and configuring the path for the **kaggle.json** file.
- It then downloads a dataset called "Face Mask Dataset" from Kaggle.

```
!pip install kaggle

# configuring the path of Kaggle.json file
!mkdir -p ~/.kaggle
!cp kaggle.json ~/.kaggle/
!chmod 600 ~/.kaggle/kaggle.json

"""### Importing Face Mask DataSet"""

# API to fetch the dataset from Kaggle
!kaggle datasets download -d omkargurav/face-mask-dataset

# extracting the compessed Dataset
from zipfile import ZipFile
dataset = '/content/face-mask-dataset.zip'

with ZipFile(dataset,'r') as zip:
  zip.extractall()
  print('The dataset is extracted')

!ls
```

Importing and Preprocessing Images

- The code imports required libraries and extracts the downloaded dataset.
- It creates lists of file names for images with and without masks.
- It checks the number of images in each class to ensure balance.
- Labels are created for the classes: 1 for images with masks and 0 for images without masks.
- The code processes images by resizing them to a uniform size of 128x128 pixels and converting them to numpy arrays.

```
"""### Importing the Basic Dependencies"""

import os
import numpy as np
import matplotlib.pyplot as plt
import matplotlib.image as mpimg
```

```
import cv2
from google.colab.patches import cv2_imshow
from PIL import Image
from sklearn.model_selection import train_test_split

with_mask_files = os.listdir('/content/data/with_mask')
print(with_mask_files[0:5])
# Creates a list of filenames in the given directory

without_mask_files = os.listdir('/content/data/without_mask')
print(without_mask_files[0:5])

print('Number of with mask images: ', len(with_mask_files))
print('Number of without mask images: ', len(without_mask_files))

"""Almost equal so no Imbalance

### Creating labels for both the Classes
- with mask --> 1
- without mask --> 0
"""

with_mask_labels = [1] * 3725
without_mask_labels = [0] * 3828

print(len(with_mask_labels))
print(len(without_mask_labels))

print(with_mask_labels[0:5])
print(without_mask_labels[0:5])

labels = with_mask_labels + without_mask_labels

print(len(labels))

"""**Displaying the Images**"""

# with mask image
img = mpimg.imread('/content/data/with_mask/with_mask_1753.jpg')
imgplot = plt.imshow(img)
plt.show()
# here we are representing the image as a np array then plotting it

# without mask image
img = mpimg.imread('/content/data/without_mask/without_mask_890.jpg')
imgplot = plt.imshow(img)
plt.show()
```

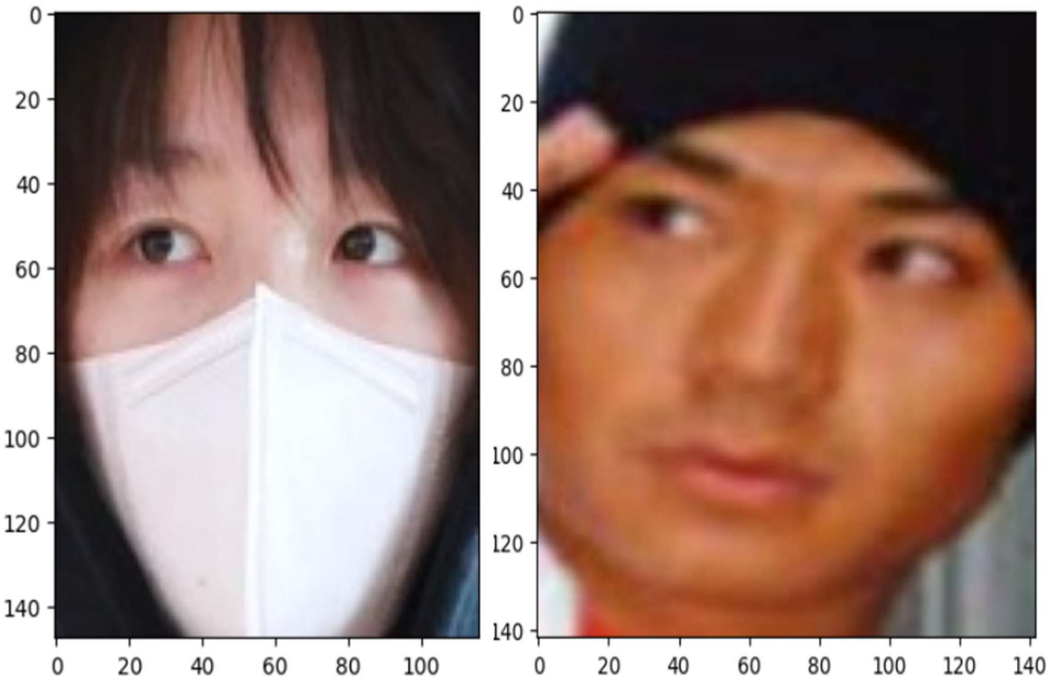
```
"""Here we dont know if the images are of the same size so we have to keep in mind during preprocessing

### Image Processing
- Resize
- Convert the images to nparray
"""

# Resizing, dealing with frrey scale images, convert to np.array

with_mask_path = '/content/data/with_mask/'
# put that slash at the end
data = []

for img_file in with_mask_files:
  image = Image.open(with_mask_path + img_file) # to read the image and convert it to pillow object
  image = image.resize((128,128))

  image = image.convert("RGB") # TO DEAL WITH GREY SCALE IMAGE
  image = np.array(image)
  data.append(image)

without_mask_path = '/content/data/without_mask/'

for img_file in without_mask_files:

  image = Image.open(without_mask_path + img_file)
  image = image.resize((128,128))

  image = image.convert("RGB")
  image = np.array(image)
  data.append(image)

print(len(data))
print(type(data))

print(data[0])

print(data[0].shape)

# converting data and labels to np array
X = np.array(data)
Y = np.array(labels)
```

Output:

```
['with_mask_1753.jpg', 'with_mask_986.jpg', 'with_mask_3366.jpg', 'with_mask_3571.jpg', 'with_mask_793.jpg']
```

```
['without_mask_890.jpg', 'without_mask_1992.jpg',
'without_mask_1869.jpg', 'without_mask_3123.jpg',
'without_mask_1642.jpg']
```

```
Number of with mask images:  3725
Number of without mask images:  3828
```



Train-Test Split

- The dataset is split into training and testing sets with an 80-20 ratio.
- Pixel values of images are scaled to the range [0, 1].

```
"""### Train Test Split"""

X_train, X_test, Y_train, Y_test = train_test_split(X, Y, test_size =
0.2, random_state=2, stratify = Y)

print(X_train.shape, X_test.shape)

# Scaling the data
X_train_scaled = X_train/ 255.0
X_test_scaled = X_test/255.0

# basically done min_max scaling
# white -->1
# black -->0
```

Building the Convolutional Neural Network (CNN)

- A CNN model is defined using Keras Sequential API.

- The model consists of convolutional layers, max-pooling layers, and dense layers.
- The final layer has a sigmoid activation function since it's a binary classification problem.

```
"""### Building a Convolutional Neural Network"""

import tensorflow as tf
from tensorflow import keras
from keras.models import Sequential
from keras.layers import Activation,Conv2D,MaxPooling2D,Flatten,
Dropout, Dense

num_of_classes = 2

model = Sequential([
    Conv2D(32, kernel_size = (3,3), activation='relu',
input_shape=(128,128,3)),
    MaxPooling2D(pool_size=(2,2)),

    Conv2D(64, kernel_size = (3,3), activation='relu'),
    MaxPooling2D(pool_size=(2,2)),

    Flatten(),

    Dense(128, activation = 'relu'),
    Dropout(0.3),

    Dense(64, activation = 'relu'),
    Dropout(0.3),

    Dense(num_of_classes, activation = 'sigmoid')

])
```

Model Training

- The model is compiled with an Adam optimizer, sparse categorical cross-entropy loss function, and accuracy as the metric.
- Training is performed using the training data with validation split for monitoring.

```
# comlpile the neural network

model.compile(
    optimizer = 'adam',
    loss = 'sparse_categorical_crossentropy',
    metrics = ['acc']
)
```

```
# training the NN
history = model.fit(X_train_scaled, Y_train, validation_split = 0.1, epochs=5)
```

Output:

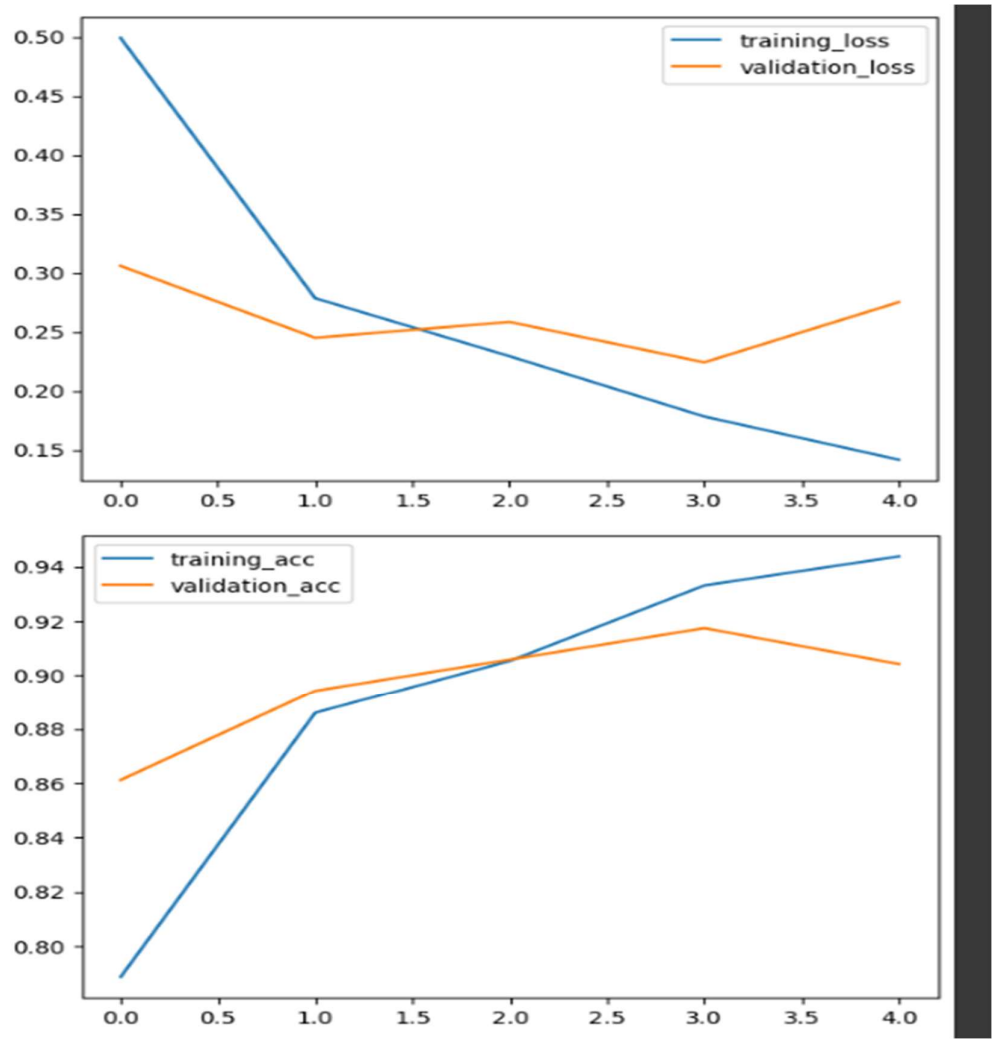
```
Epoch 1/5
170/170 [==============================] - 11s 24ms/step - loss: 0.4991 - acc: 0.7887 - val_loss: 0.3058 - val_acc: 0.8612
Epoch 2/5
170/170 [==============================] - 3s 18ms/step - loss: 0.2785 - acc: 0.8860 - val_loss: 0.2449 - val_acc: 0.8942
Epoch 3/5
170/170 [==============================] - 3s 18ms/step - loss: 0.2292 - acc: 0.9053 - val_loss: 0.2584 - val_acc: 0.9058
Epoch 4/5
170/170 [==============================] - 3s 20ms/step - loss: 0.1782 - acc: 0.9331 - val_loss: 0.2241 - val_acc: 0.9174
Epoch 5/5
170/170 [==============================] - 3s 17ms/step - loss: 0.1416 - acc: 0.9437 - val_loss: 0.2751 - val_acc: 0.9041
```

Model Evaluation

- The model's performance is evaluated on the test data, and accuracy is printed.
- Training and validation loss and accuracy are plotted to visualize the model's learning curve.

```

"""### Model Evaluation"""

loss, accuracy = model.evaluate(X_test_scaled, Y_test)
print('Test Accuracy = ', accuracy)

h = history

# plot the loss value
plt.plot(h.history['loss'], label = 'training_loss')
plt.plot(h.history['val_loss'], label = 'validation_loss')
plt.legend()
plt.show()

# plot the accuracy value
plt.plot(h.history['acc'], label = 'training_acc')
plt.plot(h.history['val_acc'], label = 'validation_acc')
plt.legend()
plt.show()
```

Output:



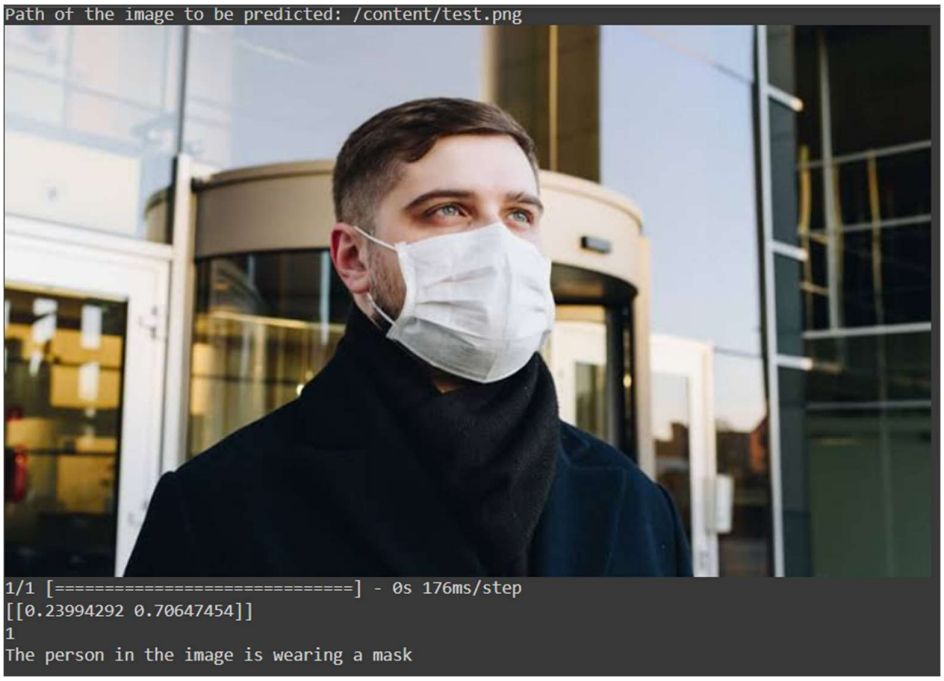
Predictive System

- A function **pred_system()** is defined to predict whether a person in an image is wearing a mask or not.
- It takes the path of an image as input, reads the image, resizes it, scales the pixel values, and reshapes it for prediction.
- The trained model predicts the class probabilities, and the predicted label is determined.
- Based on the predicted label, it prints whether the person in the image is wearing a mask or not.

```
"""**Predictive System**"""

def pred_system():
  input_image_path = input('Path of the image: ')

  input_image = cv2.imread(input_image_path)

```

```
    cv2_imshow(input_image)

    input_image_resized = cv2.resize(input_image, (128,128))

    cv2_imshow(input_image_resized)

    input_image_scaled = input_image_resized/255

    input_image_reshaped = np.reshape(input_image_scaled, [1, 128, 128,
  3])

    input_prediction = model.predict(input_image_reshaped)

    print(input_prediction)

    input_pred_label = np.argmax(input_prediction)

    print(input_pred_label)

    if input_pred_label == 1:
      print('The person in the image is wearing a mask')

    else:
      print('The person in the image is not wearing a mask')
```

Output:



```
Path of the image to be predicted: /content/test.jpg
```


```
1/1 [==============================] - 0s 21ms/step
[[0.49811754 0.47740024]]
0
The person in the image is not wearing a mask
```

Hence our Prediction System performs pretty well on unseen data.