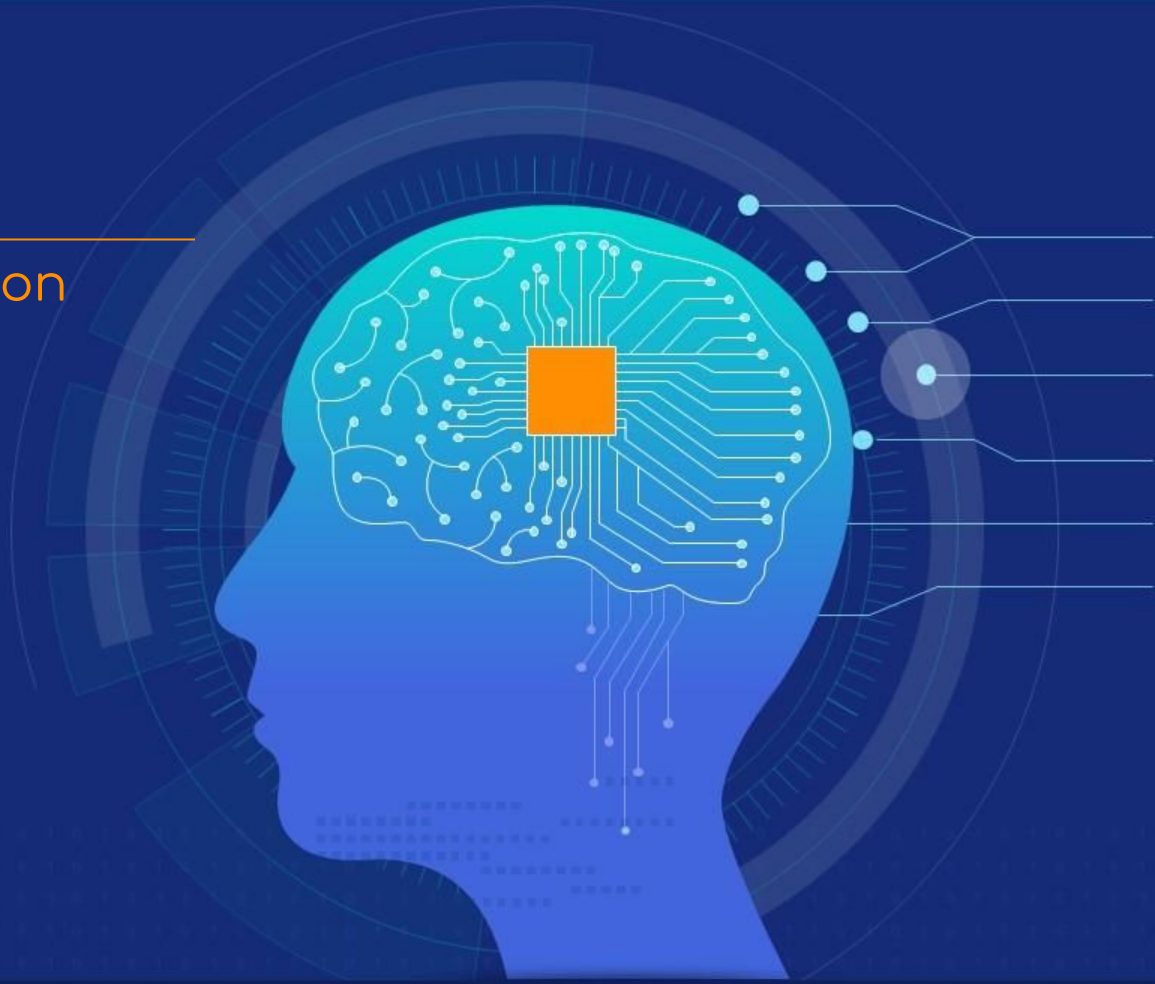
Project 4

Human Emotion Recognition



- Parvez Alam
- Kaushal Patel
- Wilfred Andoh
- Frank McKenzie-Stripp

Introduction

- Emotion recognition is the field of identifying distinct human emotions - usually through facial expressions.
- Humans are innately skilled at this, but even we struggle sometimes.
- Goal: Create a machine learning model capable of classifying a person's emotion based on their facial expression.

Overview

01 Data Collection

02 ETL Process

03 Train the Model

04 Test the Model

Dataset

- Consists of 27,000 48×48 grayscale images of faces, categorized by facial expression into seven categories:
  - Anger, disgust, fear, happiness, neutrality, sadness, and surprise.

Sources:
https://www.kaggle.com/datasets/sudarshanvaidya/random-images-for-face-emotion-recognition
https://www.kaggle.com/datasets/jonathanoheix/face-expression-recognition-dataset

Human Face Recognition

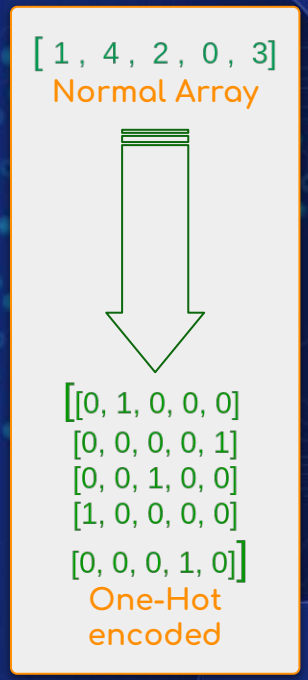
- An additional model for determining if a face is human or not.
  - Dataset is made up of images divided into “human_face” and “others”.






Source:
https://www.kaggle.com/datasets/prasunroy/natural-images

Preprocessing



To prepare the dataset for use:

- Image files were programmatically renamed.
  - E.g. Anger images renamed as: “an_0”, “an_1”, “an_2”, etc.
- Resize and convert images to grayscale.
- Created a function for converting images to NumPy arrays.
- Split the datasets into two, for training and testing.
- Scaled the data using MinMaxScaler()
- One-Hot encoded data using the Keras to_categorical() function.

CNN

What is a CNN?

- A class of deep learning neural networks.
- In a CNN, every image is represented in the form of an array of pixel values.
- CNN can take in an input image, assign importance (learnable weights and biases) to various aspects/objects in the image, and be able to differentiate one from the other.

How it works:

- The input layer which is a grayscale image.
- The output layer which is a binary or multi-class labels.
- Hidden layers consisting of convolution layers with ReLU (rectified linear unit)/ swish activation, batch normalization layer, pooling layers, and a fully connected Neural Network.

CNN

A CNN architecture primarily involves three types of layers:

1. Convolutional layer

```
model.add(Conv2D(32, (3, 3), activation='relu', input_shape=(48, 48, 1)))
```

The model uses 32 filters stacked one after the other and each filter has the dimension 3x3. Thus, the model will learn 3x3x32 i.e total of 288 parameters.

2. Pooling Layer

```
model.add(MaxPooling2D((2, 2)))
```

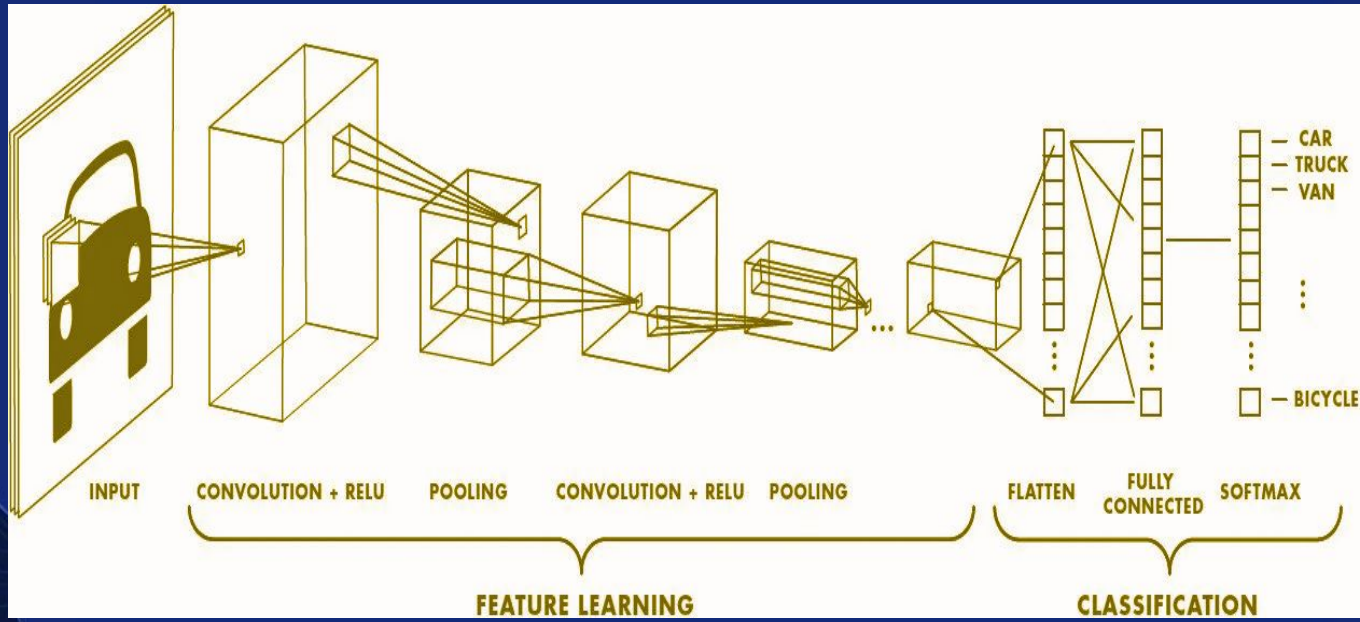
Here the layer reduces the input image by half in width and height.

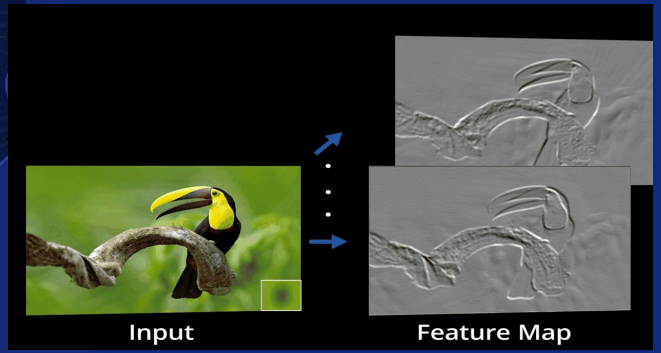
3. Fully connected layer

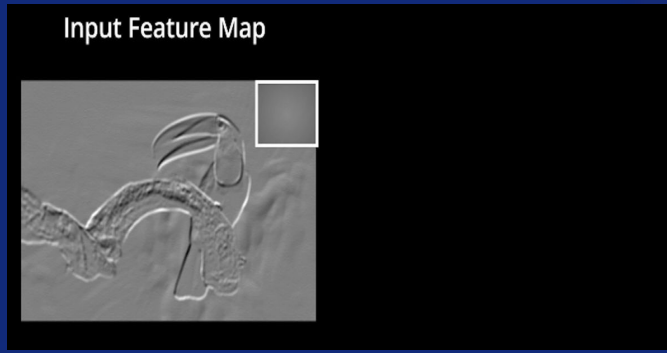
```
model.add(Dense(7, activation='softmax'))
```

Our model will have the above layer at the end, corresponding to 7 categories.

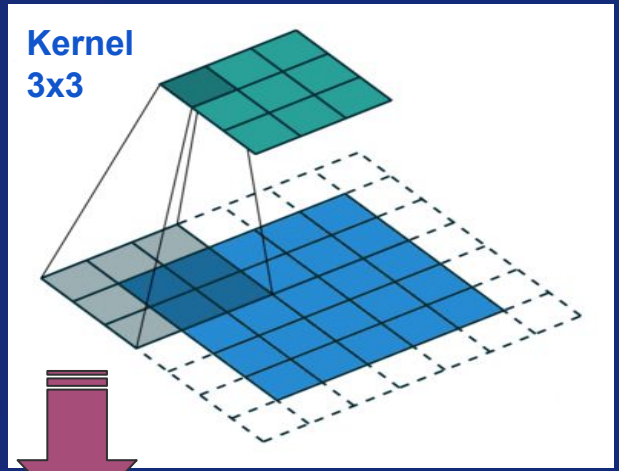
CNN

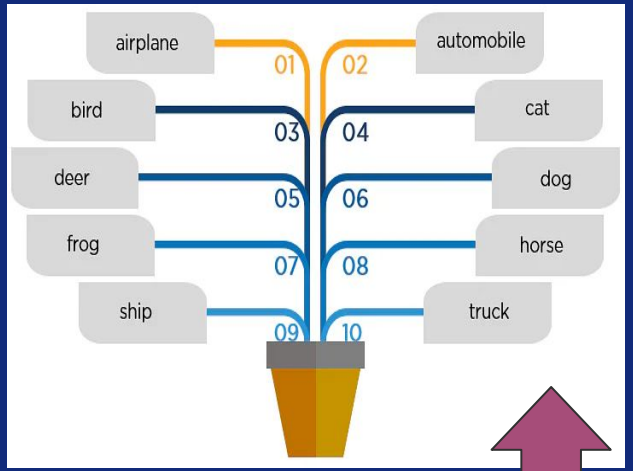

CAR
TRUCK
VAN
BICYCLE
INPUT
CONVOLUTION + RELU
POOLING
CONVOLUTION + RELU
POOLING
FLATTEN
FULLY CONNECTED
SOFTMAX
FEATURE LEARNING
CLASSIFICATION


Input
Feature Map


Input Feature Map

CNN


Kernel
3x3


airplane
automobile
bird
cat
deer
dog
frog
horse
ship
truck
01
02
03
04
05
06
07
08
09
10

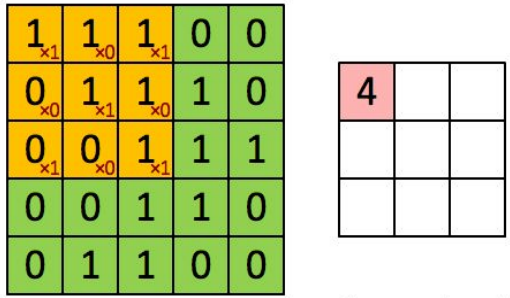


Image
Convolved Feature

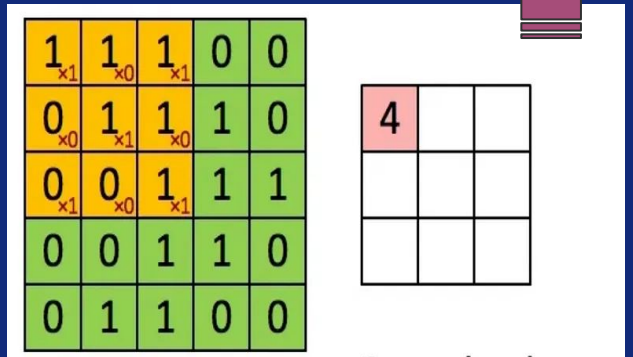
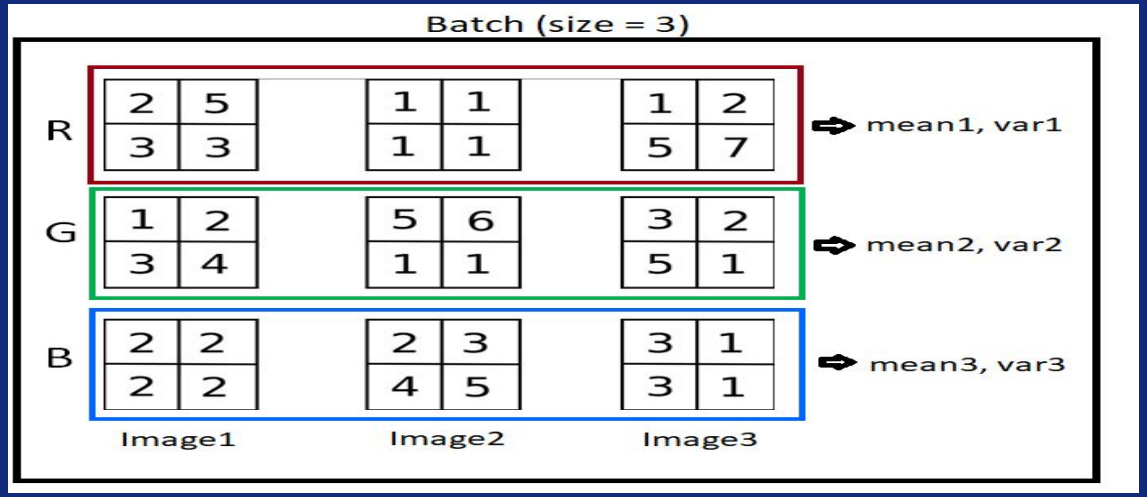


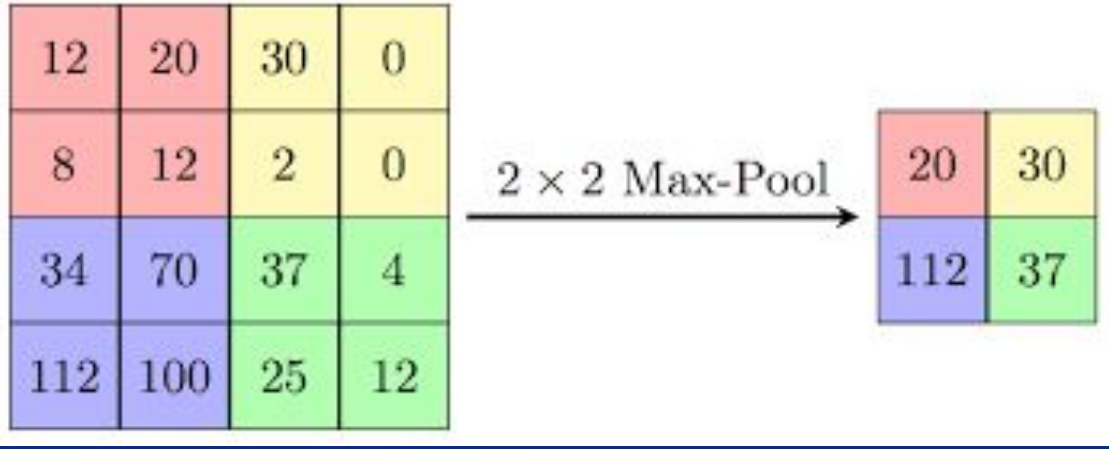
Image
Convolved Feature

CNN

Batch Normalization

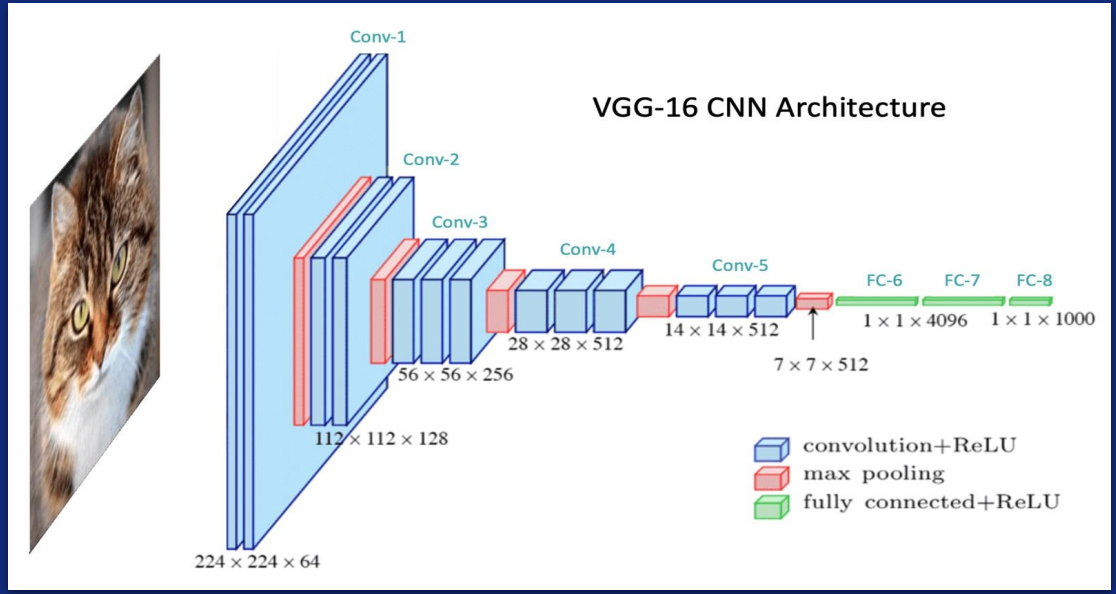


Max Pool



CNN

Example of a CNN Model:



- acc@1 (on ImageNet-1K) 71.592
- acc@5 (on ImageNet-1K) 90.382

Source: https://pytorch.org/vision/main/models/generated/torchvision.models.vgg16.html

CNN

Face Model Architecture

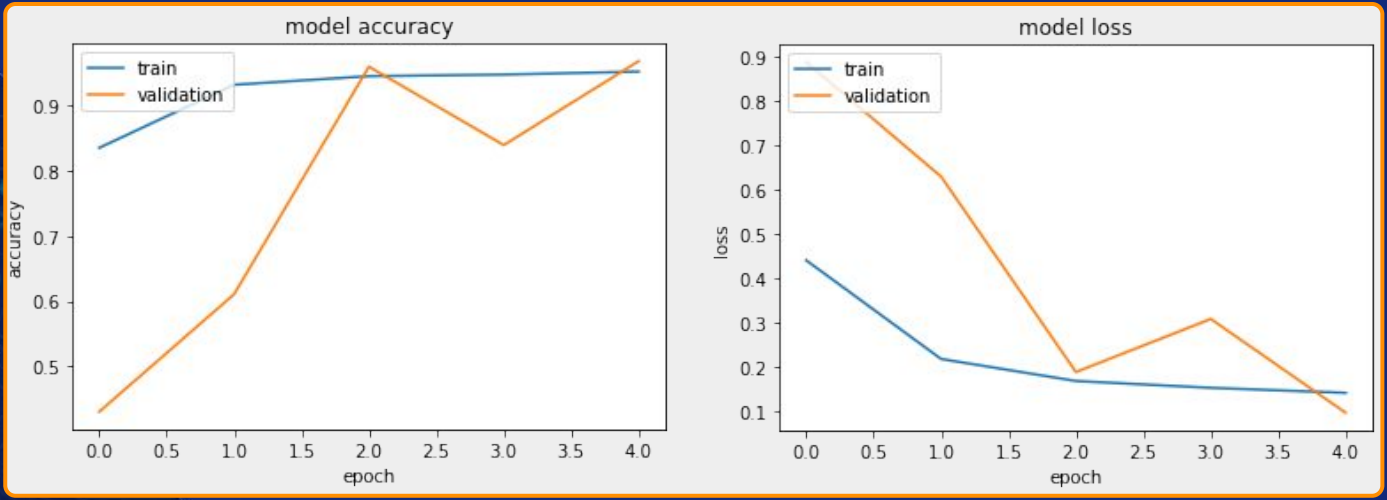
```
# building a linear stack of layers with the sequential model
model = Sequential()
activ = 'relu'

# convolutional layer
model.add(Conv2D(32, kernel_size=3, padding='valid',
                 activation=activ,
                 input_shape=X_train[0].shape))
model.add(BatchNormalization())
model.add(MaxPool2D(pool_size=(2, 2)))
model.add(Dropout(0.25))

model.add(Conv2D(64, kernel_size=3, padding='valid',
                 activation=activ))
model.add(BatchNormalization())
model.add(MaxPool2D(pool_size=(2, 2)))
model.add(Dropout(0.25))

# flatten output of conv
model.add(Flatten())

# hidden layer
model.add(Dense(128, activation= activ))
model.add(BatchNormalization())
model.add(Dropout(0.5))

model.add(Dense(64, activation= activ))
model.add(BatchNormalization())
model.add(Dropout(0.5))

# output layer
model.add(Dense(2, activation='sigmoid'))

# compiling the sequential model
model.compile(loss='binary_crossentropy',
              metrics=['accuracy'], optimizer= 'sgd')
```

Emotion Model Architecture

```
# building a linear stack of layers with the sequential model
cnn6 = Sequential()
activ = 'swish'

# convolutional layer
cnn6.add(Conv2D(32, kernel_size=4, padding='valid',
                activation=activ, input_shape=input_shape))
cnn6.add(BatchNormalization())

cnn6.add(Conv2D(32, kernel_size=4, activation=activ))
cnn6.add(BatchNormalization())
cnn6.add(MaxPool2D(pool_size=(2, 2)))
cnn6.add(Dropout(0.25))

cnn6.add(Conv2D(64, kernel_size=3, activation=activ))
cnn6.add(BatchNormalization())
cnn6.add(Dropout(0.25))

cnn6.add(Conv2D(64, kernel_size=3, activation=activ))
cnn6.add(BatchNormalization())
cnn6.add(MaxPool2D(pool_size=(2, 2)))
cnn6.add(Dropout(0.25))

cnn6.add(Conv2D(128, kernel_size=2, activation=activ))
cnn6.add(BatchNormalization())
cnn6.add(Dropout(0.25))

cnn6.add(Conv2D(128, kernel_size=2, padding='valid',
                activation=activ))
cnn6.add(BatchNormalization())
cnn6.add(MaxPool2D(pool_size=(2, 2)))
cnn6.add(Dropout(0.25))

cnn6.add(Flatten())

cnn6.add(Dense(512, activation=activ))
cnn6.add(BatchNormalization())
cnn6.add(Dropout(0.5))

cnn6.add(Dense(128, activation=activ))
cnn6.add(BatchNormalization())
cnn6.add(Dropout(0.5))
# output layer
cnn6.add(Dense(7, activation='softmax'))

# compiling the sequential model
cnn6.compile(loss='categorical_crossentropy',
             metrics=['accuracy'], optimizer='adam')
```

Performance

Face Model



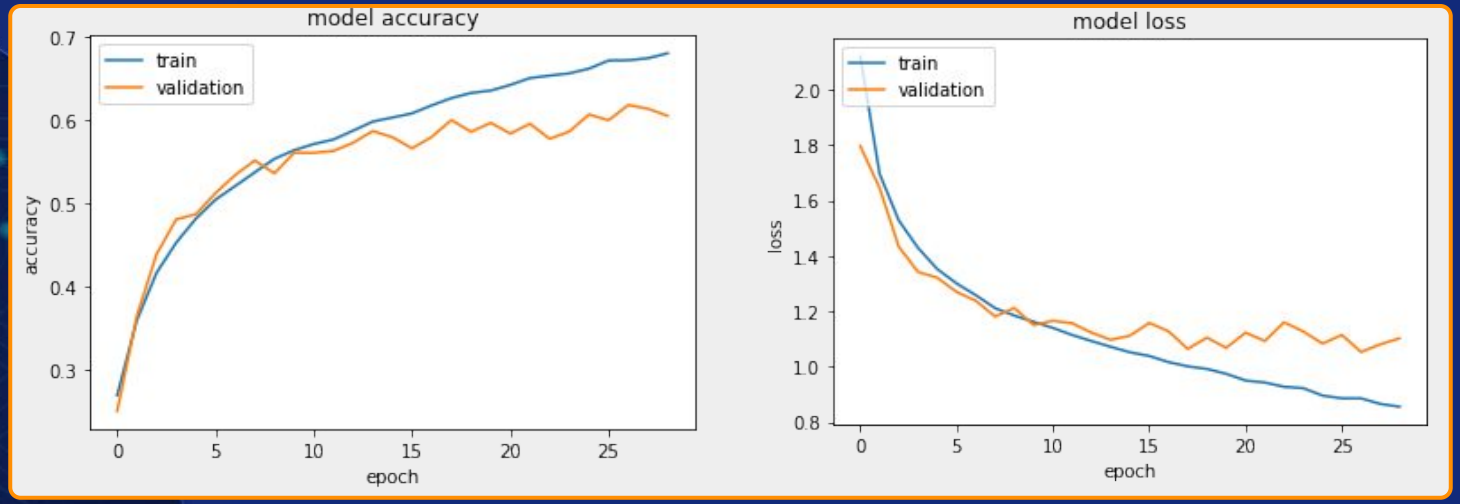

```
# Evaluate the model using the training data
model_loss, model_accuracy = model.evaluate(X_test, y_test, verbose=2)
print(f"Loss: {model_loss}, Accuracy: {model_accuracy}")
```

21] ✓ 6.9s

```
112/112 - 7s - loss: 0.0978 - accuracy: 0.9680 - 7s/epoch - 58ms/step
Loss: 0.09782330691814423, Accuracy: 0.9680314064025879
```

Performance

Emotion Model




```
# Evaluate the model using the training data
model_loss, model_accuracy = cnn6.evaluate(X_test, y_test, verbose=2)
print(f"Loss: {model_loss}, Accuracy: {model_accuracy}")
```

```
225/225 - 18s - loss: 1.0759 - accuracy: 0.6151 - 18s/epoch - 82ms/step
Loss: 1.0759339332580566, Accuracy: 0.6150639057159424
```

Model Evaluation

Input Limitation:
Our models are trained with images of human face only.



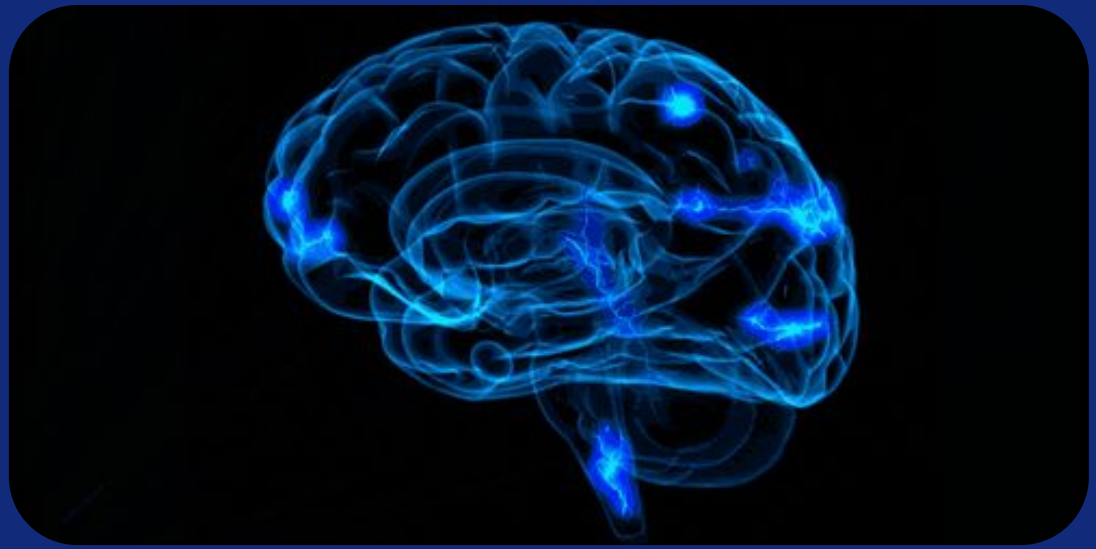

Demonstration:

Conclusion

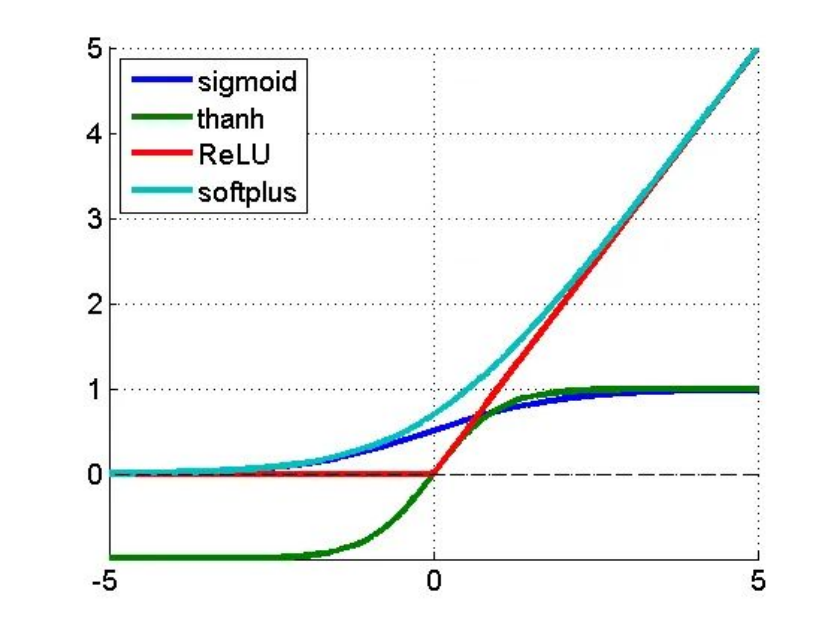
- Emotion recognition is a challenging field, due to the diversity in how people express themselves
- Human Face Recognition validation accuracy >90%
- Emotion Recognition validation accuracy Top1 >60%, Top3 > 88%
- Overall, human face recognition was extremely successful
  - Emotion recognition is still a work in progress
- Limitations:
  - Time
  - Relatively low processing power
- With more time and resources, model accuracy could continue to be further improved

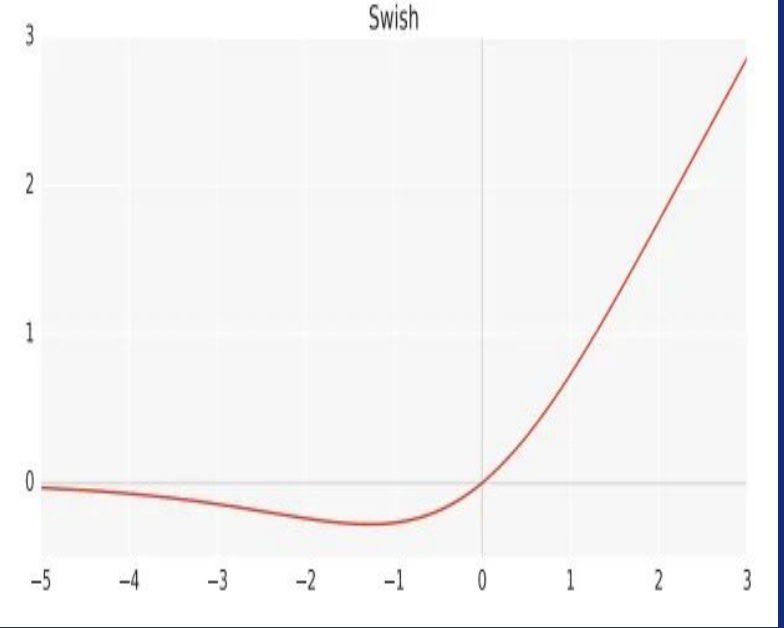
Thanks for listening!

Q&A



Activation function





Top 3 categorical accuracy of emotion model

```
Epoch 1/20
338/338 [==============================] - 247s 731ms/step - loss: 2.1785 - accuracy: 0.2553 - <lambda>: 0.5894 - val_loss: 1.8391 - val_accuracy: 0.2668 - val_<lambda>: 0.5996
Epoch 2/20
338/338 [==============================] - 248s 734ms/step - loss: 1.6989 - accuracy: 0.3554 - <lambda>: 0.6987 - val_loss: 1.5513 - val_accuracy: 0.3919 - val_<lambda>: 0.7371
Epoch 3/20
338/338 [==============================] - 248s 734ms/step - loss: 1.5221 - accuracy: 0.4130 - <lambda>: 0.7530 - val_loss: 1.7051 - val_accuracy: 0.3621 - val_<lambda>: 0.7093
Epoch 4/20
338/338 [==============================] - 248s 735ms/step - loss: 1.4134 - accuracy: 0.4615 - <lambda>: 0.7935 - val_loss: 1.3273 - val_accuracy: 0.4901 - val_<lambda>: 0.8267
Epoch 5/20
338/338 [==============================] - 248s 733ms/step - loss: 1.3505 - accuracy: 0.4825 - <lambda>: 0.8123 - val_loss: 1.2568 - val_accuracy: 0.5203 - val_<lambda>: 0.8378
Epoch 6/20
338/338 [==============================] - 247s 730ms/step - loss: 1.2886 - accuracy: 0.5060 - <lambda>: 0.8344 - val_loss: 1.2188 - val_accuracy: 0.5302 - val_<lambda>: 0.8517
Epoch 7/20
338/338 [==============================] - 248s 733ms/step - loss: 1.2418 - accuracy: 0.5278 - <lambda>: 0.8438 - val_loss: 1.2345 - val_accuracy: 0.5231 - val_<lambda>: 0.8389
Epoch 8/20
338/338 [==============================] - 247s 730ms/step - loss: 1.2147 - accuracy: 0.5385 - <lambda>: 0.8559 - val_loss: 1.1891 - val_accuracy: 0.5500 - val_<lambda>: 0.8588
Epoch 9/20
338/338 [==============================] - 247s 730ms/step - loss: 1.1781 - accuracy: 0.5544 - <lambda>: 0.8622 - val_loss: 1.1523 - val_accuracy: 0.5624 - val_<lambda>: 0.8638
Epoch 10/20
338/338 [==============================] - 245s 725ms/step - loss: 1.1556 - accuracy: 0.5617 - <lambda>: 0.8699 - val_loss: 1.2028 - val_accuracy: 0.5409 - val_<lambda>: 0.8580
Epoch 11/20
338/338 [==============================] - 245s 726ms/step - loss: 1.1344 - accuracy: 0.5747 - <lambda>: 0.8756 - val_loss: 1.1563 - val_accuracy: 0.5627 - val_<lambda>: 0.8745
Epoch 12/20
338/338 [==============================] - 246s 727ms/step - loss: 1.1096 - accuracy: 0.5807 - <lambda>: 0.8809 - val_loss: 1.1308 - val_accuracy: 0.5767 - val_<lambda>: 0.8723
Epoch 13/20
338/338 [==============================] - 245s 726ms/step - loss: 1.0858 - accuracy: 0.5936 - <lambda>: 0.8885 - val_loss: 1.1150 - val_accuracy: 0.5866 - val_<lambda>: 0.8760
Epoch 14/20
338/338 [==============================] - 247s 732ms/step - loss: 1.0650 - accuracy: 0.6004 - <lambda>: 0.8920 - val_loss: 1.1075 - val_accuracy: 0.5838 - val_<lambda>: 0.8755
Epoch 15/20
338/338 [==============================] - 249s 736ms/step - loss: 1.0531 - accuracy: 0.6067 - <lambda>: 0.8962 - val_loss: 1.0895 - val_accuracy: 0.5874 - val_<lambda>: 0.8809
Epoch 16/20
338/338 [==============================] - 249s 737ms/step - loss: 1.0341 - accuracy: 0.6118 - <lambda>: 0.9000 - val_loss: 1.0994 - val_accuracy: 0.5856 - val_<lambda>: 0.8837
Epoch 17/20
338/338 [==============================] - 249s 736ms/step - loss: 1.0068 - accuracy: 0.6234 - <lambda>: 0.9053 - val_loss: 1.0819 - val_accuracy: 0.5914 - val_<lambda>: 0.8899
Epoch 18/20
338/338 [==============================] - 248s 735ms/step - loss: 0.9971 - accuracy: 0.6294 - <lambda>: 0.9058 - val_loss: 1.1057 - val_accuracy: 0.5850 - val_<lambda>: 0.8783
Epoch 19/20
338/338 [==============================] - 248s 735ms/step - loss: 0.9821 - accuracy: 0.6348 - <lambda>: 0.9090 - val_loss: 1.1077 - val_accuracy: 0.5848 - val_<lambda>: 0.8772
Epoch 20/20
338/338 [==============================] - 247s 730ms/step - loss: 0.9657 - accuracy: 0.6396 - <lambda>: 0.9157 - val_loss: 1.1124 - val_accuracy: 0.5871 - val_<lambda>: 0.8877
<keras.callbacks.History at 0x7f00996066a0>
```