Face-mask Universal Classifier with Keras

Cornel Alexandru Badea

Principal objective

Create an Artificial Intelligence software for detecting the wearing of face-mask using live video input from the webcam.

Secondary objectives

The detection should have high accuracy, using state of the art image processing techniques.

To draw a rectangle on the detected face and display the accuracy.

Design an arduino based embedded system that signals the user if she wears face-mask based on serial information received from the software.

Bibliographic study

Keras and Tensorflow

TensorFlow is a free and open-source software library for dataflow and differentiable programming across a range of tasks. It is a symbolic math library, and is also used for machine learning applications such as neural networks. It is used for both research and production at Google.

Keras is an API designed for human beings, not machines. Keras follows best practices for reducing cognitive load: it offers consistent & simple APIs, it minimizes the number of user actions required for common use cases, and it provides clear & actionable error messages. It also has extensive documentation and developer guides.

Keras is a high-level interface and uses Theano or Tensorflow for its backend. It runs smoothly on both CPU and GPU. Keras supports almost all the models of a neural network – fully connected, convolutional, pooling, recurrent, embedding, etc.

We will use tensorflow and keras because they allow developers to create large-scale neural networks with many layers and it is is mainly used for Classification and Perception, the two problems that we want to solve for our live video face-mask classifier.

Keras contains 10 pretrained models for image classification which are trained on Imagenet data. Imagenet is a large collection of image data containing 1000 categories of images. These pretrained models are capable of classifying any image that falls into these 1000 categories of images. One of those 10 models is MobileNet model. That we will use for our final Neural Network Architecture.

What is MobileNet ?

MobileNet is a Convolutional Neural Network CNN architecture model for Image Classification and Mobile Vision.There are other models as well but what makes MobileNet special that it very less computation power to run or apply transfer learning to.This makes it a perfect fit for Mobile devices,embedded systems and computers without GPU or low computational efficiency with compromising significantly with the accuracy of the results.It is also best suited for web browsers as browsers have limitation over computation,graphic processing and storage.

What is a Newral Network[*][*]

A neural network is a network or circuit of neurons, or in a modern sense, an artificial neural network, composed of artificial neurons or nodes.

Artificial neural networks (ANNs), usually simply called neural networks (NNs), are computing systems vaguely inspired by the biological neural networks that constitute animal brains.

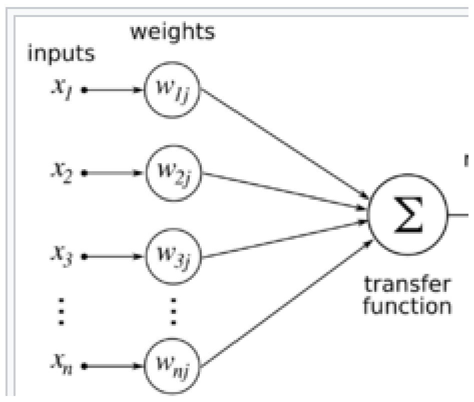
Neural networks learn (or are trained) by processing examples, each of which contains a known "input" and "result," forming probability-weighted associations between the two, which are stored within the data structure of the net itself. The training of a neural network from a given example is usually conducted by determining the difference between the processed output of the network (often a prediction) and a target output. This is the error. The network then adjusts its weighted associations according to a learning rule and using this error value. Successive adjustments will cause the neural network to produce output which is increasingly similar to the target output. After a sufficient number of these adjustments the training can be terminated based upon certain criteria. This is known as supervised learning.

Basically Artificial Neural Network is a correction a connected newrons called perceptrons. A perceptron is activated only if it receives the right input. This is how it looks look:

$$f(\mathbf{x}) = \begin{cases} 1 & \text{if } \mathbf{w} \cdot \mathbf{x} + b > 0, \\ 0 & \text{otherwise} \end{cases}$$

The above is the function for the a binary perceptron it returns 1 if the weighted input x summed with b for non-zero assurance is greater than 0.

What if we have a lot of inputs ? We simply weight them all and sum the results like in the below image:



The b values called biases are always present in these newrons we don’t want the values of the output to be 0, we will se bellow why.

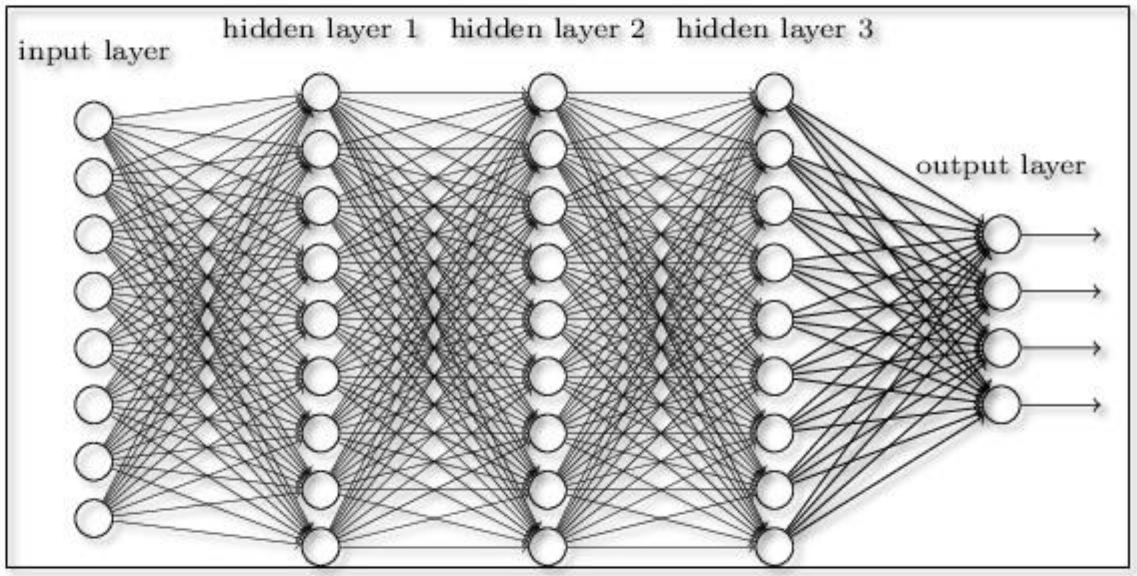
Can the output of a neuron be input for another newron? Yes it can and we just defined Deep Newral Networks.

Analysis

Deep Layered Network Architecture
Deep neural networks compose computations performed by many layers. Denoting the output of hidden layers by $h^{(l)}(x)$, the computation for a network with L hidden layers is:

$$\mathbf{f}(\mathbf{x}) = \mathbf{f}\left[\mathbf{a}^{(L+1)}\left(\mathbf{h}^{(L)}\left(\mathbf{a}^{(L)}\left(\ldots\left(\mathbf{h}^{(2)}\left(\mathbf{a}^{(2)}\left(\mathbf{h}^{(1)}\left(\mathbf{a}^{(1)}(\mathbf{x})\right)\right)\right)\right)\right)\right)\right)\right].$$

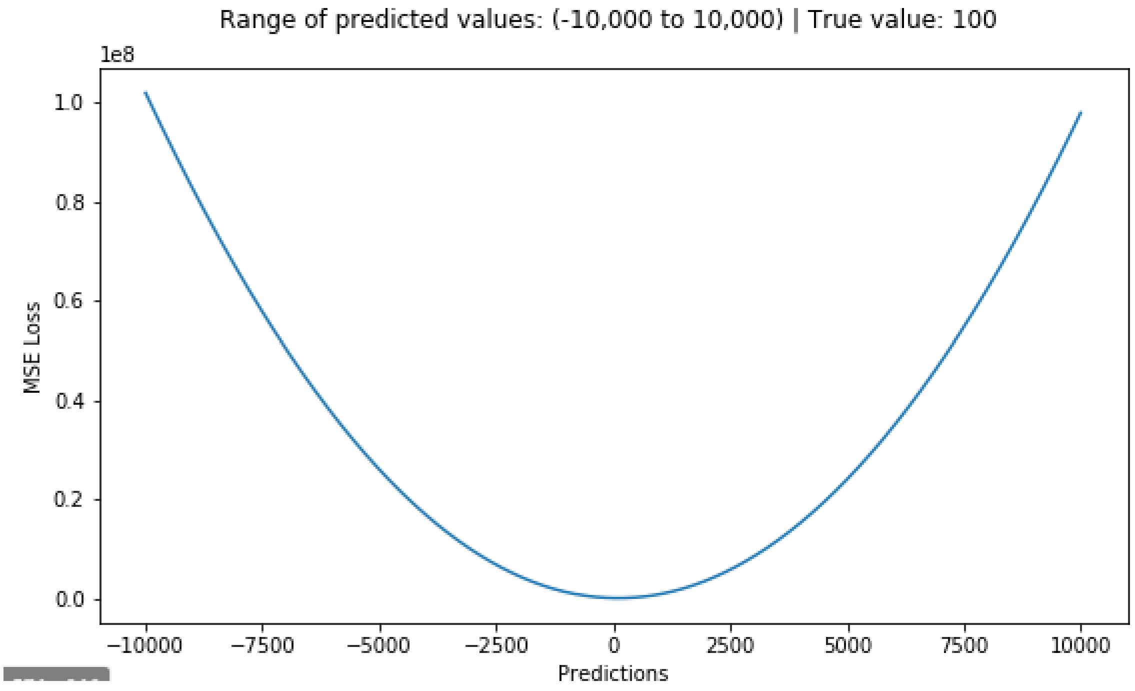
We just apply the same function over and over. The visual representation looks like this:



But what if we don’t want outputs to be 0 or 1, what if we want to obtains something usefull out of these perceptrons. This is where the cost function comes intp place, especially in supervised learning. Cost function or loss function or error function they are synonimes and the definition is symple.

What’s a Loss Function?
At its core, a loss function is incredibly simple: it’s a method of evaluating how well your algorithm models your dataset. If your predictions are totally off, your loss function will output a higher number. If they’re pretty good, it’ll output a lower number. As you change pieces of your algorithm to try and improve your model, your loss function will tell you if you’re getting anywhere.[*]

Here is a graph that plost the neural network outputs(predictions) versus what we want the outputs to be.



MSE Loss is a type of loss that represents the mean squared difference between the target and output but is not relevant for our project.

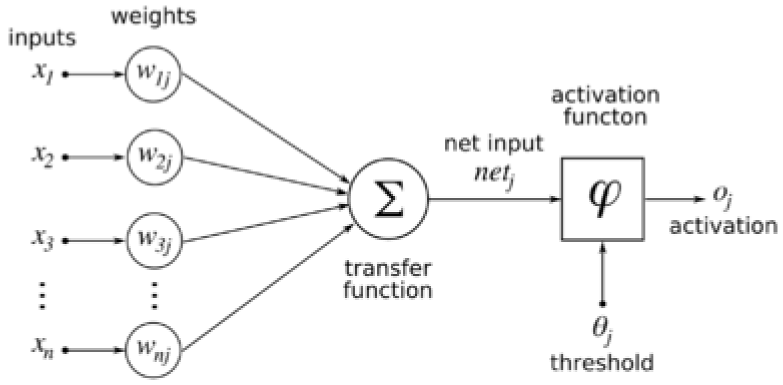
The relevant part is how do we change the output i.e the predictions of our newral network.

1. We can change the inputs – that is not what we want
2. We can change our weights since they are the ones that transform our input to a desirable output.

How do we change our weights to minimize the loss function?

Backpropagation
Backpropagation is a method to adjust the connection weights to compensate for each error found during learning. The error amount is effectively divided among the connections. Technically, backprop calculates the gradient (the derivative) of the cost function associated with a given state with respect to the weights. The weight updates can be done via stochastic gradient descent or other methods.

Finding the derivative of the error



Activation function is simply a function that tells the newron when to output data.

For the simple preceptron example the activation function was the output to be greater than 1.

Diagram of an artificial neural network to illustrate the notation used here.

Calculating the partial derivative of the error with respect to a weight wij i*s done using the chain rule* twice:

$$\frac{\partial E}{\partial w_{ij}} = \frac{\partial E}{\partial o_j}\frac{\partial o_j}{\partial w_{ij}} = \frac{\partial E}{\partial o_j}\frac{\partial o_j}{\partial \mathrm{net}_j}\frac{\partial \mathrm{net}_j}{\partial w_{ij}}$$

The error depends on the output of the network, that in turn depends on the output of a neuron that is also dependend on the assigned weights.

But how do we change our weights with respect to the Error function? We just showed above, here is the logic: if the error is high we want to take an input that makes it lower i.e we want an input in the direction of the derivative if the derivative is negative and in the oposite direction of the derivative if the derivative is positive. This process is called optimisation(of the inputs with respect to outputs).

Loss Functions and Optimizers

Loss functions provide more than just a static representation of how your model is performing–they're how your algorithms fit data in the first place. Most machine learning algorithms use some sort of loss function in the process of optimization, or finding the best parameters (weights) for your data.

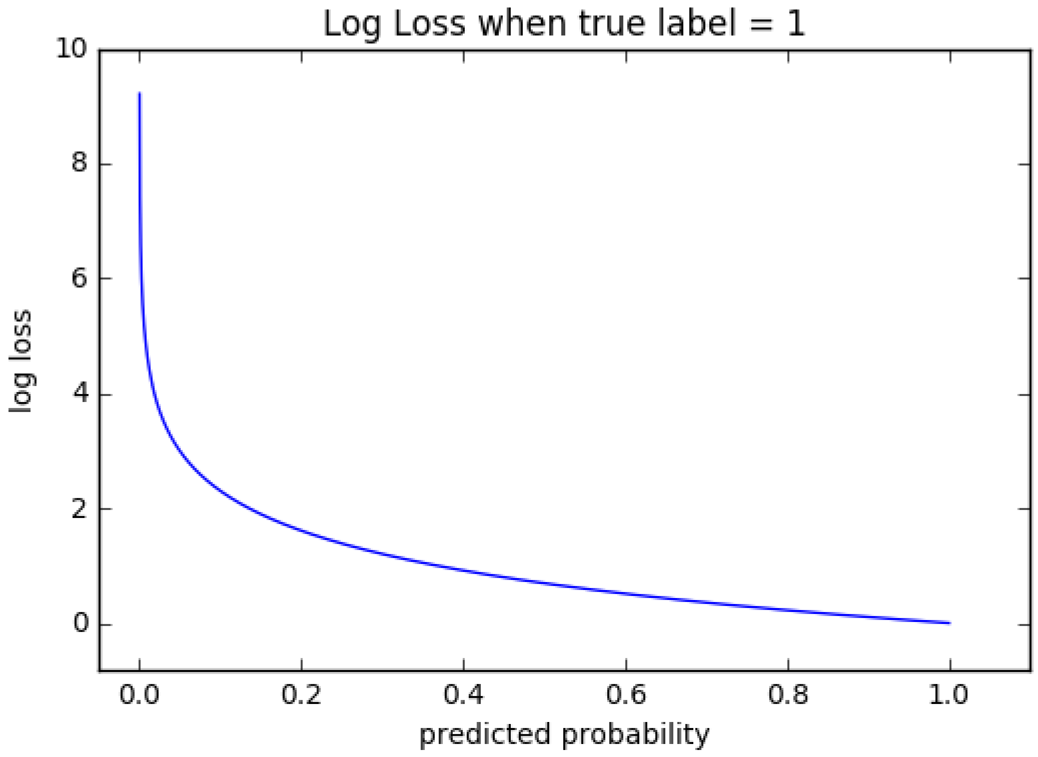
For a simple example, consider linear regression. In traditional "least squares" regression, the line of best fit is determined through none other than MSE (hence the least squares moniker)! For each set of weights that the model tries, the MSE is calculated across all input examples. The model then optimizes the MSE functions—or in other words, makes it the lowest possible—through the use of an optimizer algorithm like Gradient Descent.

Just like there are different flavors of loss functions for unique problems, there is no shortage of different optimizers as well. That's beyond the scope of this post, but in essence, the loss function and optimizer work in tandem to fit the algorithm to your data in the best way possible.

What do we use? *Log Loss (Cross Entropy Loss)*
$-(y\log(p) + (1-y)\log(1-p))$

Because it is actually exactly the same formula as the regular likelihood function, but with logarithms added in. You can see that when the actual class is 1, the second half of the function disappears, and when the actual class is 0, the first half drops. That way, we just end up multiplying the log of the actual predicted probability for the ground truth class.

The cool thing about the log loss loss function is that is has a kick: it penalizes heavily for being *very confident* and *very wrong*. The graph below is for when the true label =1, and you can see that it skyrockets as the predicted probability for label = 0 approaches 1. [*]



What is an Optimizer in Machine Learning?

During the training process, we tweak and change the parameters (weights) of our model to try and minimize that loss function, and make our predictions as correct and optimized as possible. But how exactly do you do that? How do you change the parameters of your model, by how much, and when?

This is where optimizers come in. They tie together the loss function and model parameters by updating the model in response to the output of the loss function. In simpler terms, optimizers shape and mold your model into its most accurate possible form by futzing with the weights. The loss function is the guide to the terrain, telling the optimizer when it’s moving in the right or wrong direction.

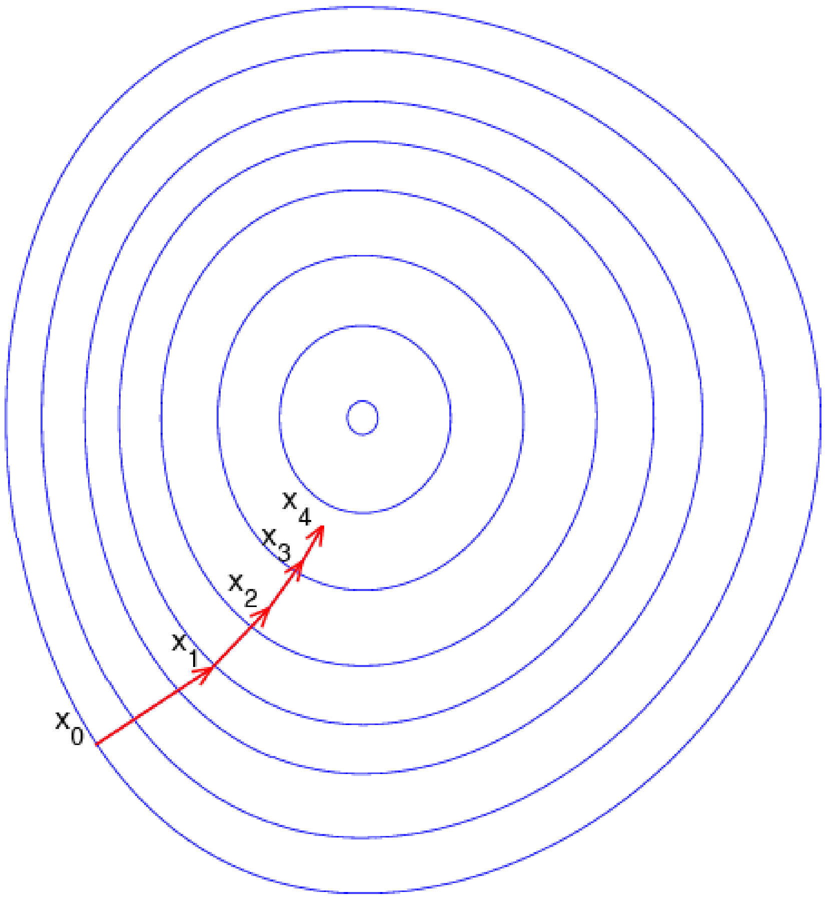
For a useful mental model, you can think of a hiker trying to get down a mountain with a blindfold on. It’s impossible to know which direction to go in, but there’s one thing she can know: if she’s going down (making progress) or going up (losing progress). Eventually, if she keeps taking steps that lead her downwards, she’ll reach the base.

Similarly, it’s impossible to know what your model’s weights should be right from the start. But with some trial and error based on the loss function (whether the hiker is descending), you can end up getting there eventually.

What is a Gradient?[*]

A gradient simply measures the change in all weights with regard to the change in error. You can also think of a gradient as the slope of a function. The higher the gradient, the steeper the slope and the faster a model can learn. But if the slope is zero, the model stops learning. In mathematical terms, a gradient is a partial derivative with respect to its inputs.

Imagine the image below illustrates our hill from a top-down view and the red arrows are the steps of our climber. Think of a gradient in this context as a vector that contains the direction of the steepest step the blindfolded man can take and also how long that step should be.



Gradient descent is based on the observation that if the multi-variable function F(x) and differentiable in a neighborhood of a point a, then F(x) decreases fastest if one goes from a in the direction of the negative gradient of F at a.

$$\mathbf{b} = \mathbf{a} - \gamma \nabla \mathbf{f}(\mathbf{a})$$

b is the next position of our climber, while *a* represents his current position. The minus sign refers to the minimization part of gradient descent. The gamma in the middle is a learning rate and the gradient term (Δf(a)) is simply the direction of the steepest descent.

Stochastic Gradient Descent (SGD)[*]
There are a few downsides of the gradient descent algorithm. We need to take a closer look at the amount of computation we make for each iteration of the algorithm.

Say we have 10,000 data points and 10 features. The sum of squared residuals consists of as many terms as there are data points, so 10000 terms in our case. We need to compute the derivative of this function with respect to each of the features, so in effect we will be doing 10000 * 10 = 100,000 computations per iteration. It is common to take 1000 iterations, in effect we have 100,000 * 1000 = 100000000 computations to complete the algorithm. That is pretty much an overhead and hence gradient descent is slow on huge data.

Stochastic gradient descent comes to our rescue.“Stochastic”, in plain terms means “random”.

SGD randomly picks one data point from the whole data set at each iteration to reduce the computations enormously.

Adagrad
Adagrad [*] is an algorithm for gradient-based optimization that does just this: It adapts the learning rate to the parameters, performing smaller updates

(i.e. low learning rates) for parameters associated with frequently occurring features, and larger updates (i.e. high learning rates) for parameters associated with infrequent features. For this reason, it is well-suited for dealing with sparse data. Dean et al. [*] have found that Adagrad greatly improved the robustness of SGD and used it for training large-scale neural nets at Google, which -- among other things -- learned to recognize cats in Youtube videos.

Previously, we performed an update for all parameters θ at once as every parameter θi used the same learning rate η. As Adagrad uses a different learning rate for every parameter θi at every time step t, we first show Adagrad's per-parameter update, which we then vectorize. For brevity, we use gt to denote the gradient at time step t. gt,i is then the partial derivative of the objective function w.r.t. to the parameter θi at time step t:

$$g_{t,i} = \nabla_\theta J(\theta_{t,i}).$$

The SGD update for every parameter θi at each time step t then becomes:

$$\theta_{t+1,i} = \theta_{t,i} - \eta \cdot g_{t,i}.$$

In its update rule, Adagrad modifies the general learning rate η at each time step t for every parameter θi based on the past gradients that have been computed for θi:

$$\theta_{t+1,i} = \theta_{t,i} - \frac{\eta}{\sqrt{G_{t,ii} + \epsilon}} \cdot g_{t,i}.$$

Gt∈Rd×d here is a diagonal matrix where each diagonal element i,i is the sum of the squares of the gradients w.r.t. θi up to time step t , while ϵ is a smoothing term that avoids division by zero (usually on the order of 1e−8). Interestingly, without the square root operation, the algorithm performs much worse.

Adadelta[*]
Adadelta is an extension of Adagrad that seeks to reduce its aggressive, monotonically decreasing learning rate. Instead of accumulating all past squared gradients, Adadelta restricts the window of accumulated past gradients to some fixed size w.

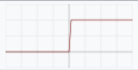
Adam[*]
Adaptive Moment Estimation (Adam) is another method that computes adaptive learning rates for each parameter. In addition to storing an exponentially decaying average of past squared gradients vt.

like Adadelta and RMSprop, Adam also keeps an exponentially decaying average of past gradients mt, similar to momentum. Whereas momentum can be seen as a ball running down a slope, Adam behaves like a heavy ball with friction, which thus prefers flat minima in the error surface.

We will use Adam optimizer for our problem because it provides an optimization algorithm that can handle sparse gradients on noisy problems, being perfect for our frame classification problem.

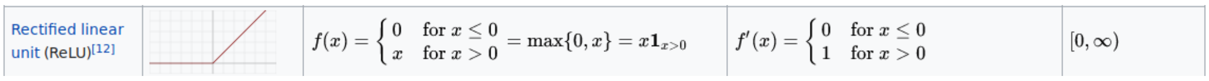
Activation Functions[*]

In artificial neural networks, the activation function of a node defines the output of that node given an input or set of inputs. A standard integrated circuit can be seen as a digital network of activation functions that can be "ON" (1) or "OFF" (0), depending on input.

Identity		$f(x) = x$	$f'(x) = 1$	$(-\infty, \infty)$
Binary step		$f(x) = \begin{cases} 0 & \text{for } x < 0 \\ 1 & \text{for } x \geq 0 \end{cases}$	$f'(x) = \begin{cases} 0 & \text{for } x \neq 0 \\ ? & \text{for } x = 0 \end{cases}$	$\{0, 1\}$

Rectified Linear Unit function[*]

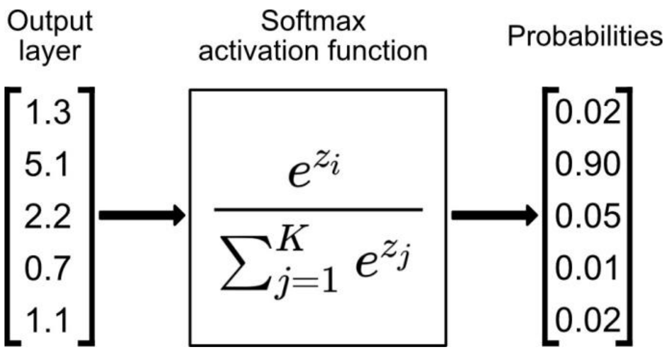
The rectified linear activation function or ReLU for short is a piecewise linear function that will output the input directly if it is positive, otherwise, it will output zero. It has become the default activation function for many types of neural networks because a model that uses it is easier to train and often achieves better performance.



Rectified linear unit (ReLU)[12]		$f(x) = \begin{cases} 0 & \text{for } x \le 0 \\ x & \text{for } x > 0 \end{cases} = \max\{0, x\} = x\mathbf{1}_{x>0}$	$f'(x) = \begin{cases} 0 & \text{for } x \le 0 \\ 1 & \text{for } x > 0 \end{cases}$	$[0, \infty)$

Softmax function[*]

Softmax turn *logits* (numeric output of the last linear layer of a multi-class classification neural network) into probabilities by take the exponents of each output and then normalize each number by the sum of those exponents so the entire output vector adds up to one — all probabilities should add up to one.


Output layer
Softmax activation function
Probabilities
1.3
5.1
2.2
0.7
1.1
$\frac{e^{z_i}}{\sum_{j=1}^{K} e^{z_j}}$
0.02
0.90
0.05
0.01
0.02

We will use the **softmax activation function** for our last layer activation function because it is used in neural networks when we want to build a multi-class classifier, in our case we have two classes with mask and without mask.

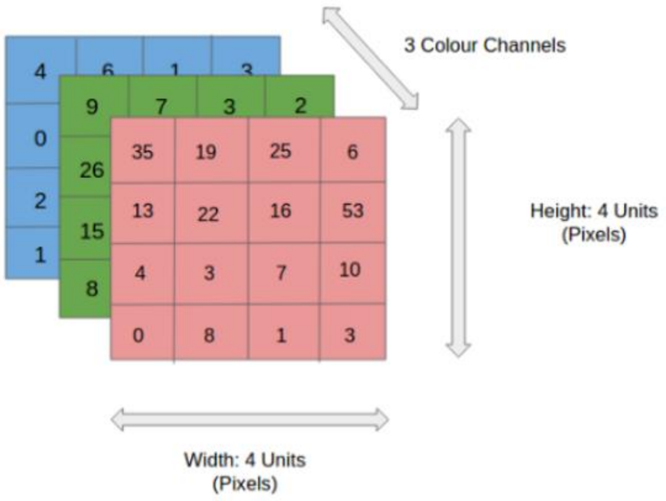
What is Convolutional Neural Network

In deep learning, a convolutional neural network (CNN, or ConvNet) is a class of deep neural networks, most commonly applied to analyzing visual imagery. They are also known as shift invariant or space invariant artificial neural networks (SIANN), based on their shared-weights architecture and translation invariance characteristics.

The architecture of a ConvNet is analogous to that of the connectivity pattern of Neurons in the Human Brain and was inspired by the organization of the Visual Cortex. Individual neurons respond to stimuli only in a restricted region of the visual field known as the Receptive Field. A collection of such fields overlap to cover the entire visual area.

A ConvNet is able to successfully capture the Spatial and Temporal dependencies in an image through the application of relevant filters. The architecture performs a better fitting to the image dataset due to the reduction in the number of parameters involved and reusability of weights. In other words, the network can be trained to understand the sophistication of the image better.

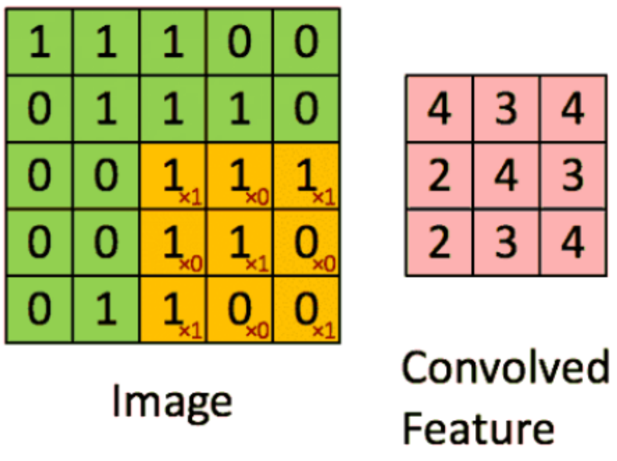
Input Image



4x4x3 RGB Image

We will aplly two steps to the input image: convolution and pooling

Convolution Layer — The Kernel

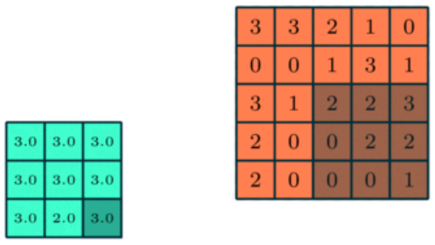


Convoluting a 5x5x1 image with a 3x3x1 kernel to get a 3x3x1 convolved feature

Image Dimensions = 5 (Height) x 5 (Breadth) x 1 (Number of channels, eg. RGB)

In the above demonstration, the green section resembles our 5x5x1 input image, I. The element involved in carrying out the convolution operation in the first part of a Convolutional Layer is called the Kernel/Filter, K, represented in the color yellow. We have selected K as a 3x3x1 matrix.

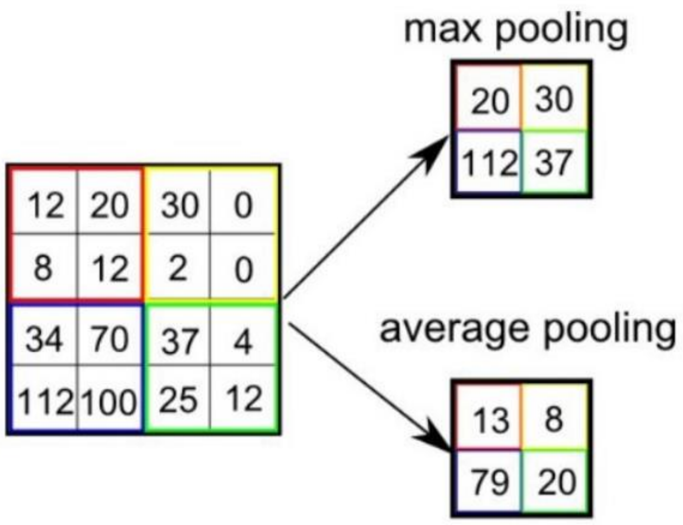
Pooling Layer



3x3 pooling over 5x5 convolved feature

Similar to the Convolutional Layer, the Pooling layer is responsible for reducing the spatial size of the Convolved Feature. This is to decrease the computational power required to process the data through dimensionality reduction. Furthermore, it is useful for extracting dominant features which are rotational and positional invariant, thus maintaining the process of effectively training of the model.

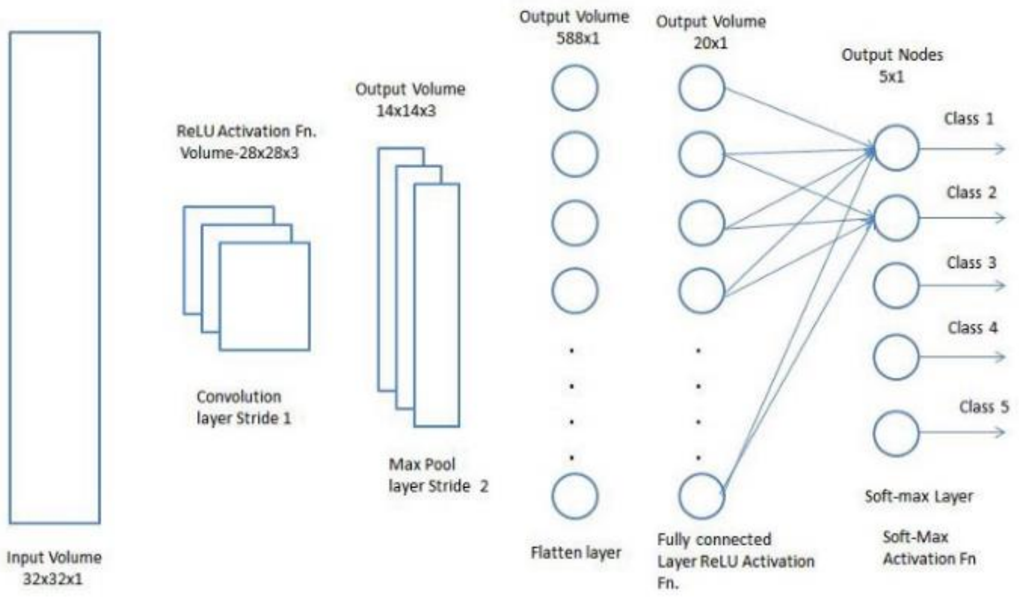
There are two types of Pooling: Max Pooling and Average Pooling. Max Pooling returns the maximum value from the portion of the image covered by the Kernel. On the other hand, Average Pooling returns the average of all the values from the portion of the image covered by the Kernel.



Types of Pooling

After going through the above process, we have successfully enabled the model to understand the features. Moving on, we are going to flatten the final output and feed it to a regular Neural Network for classification purposes.

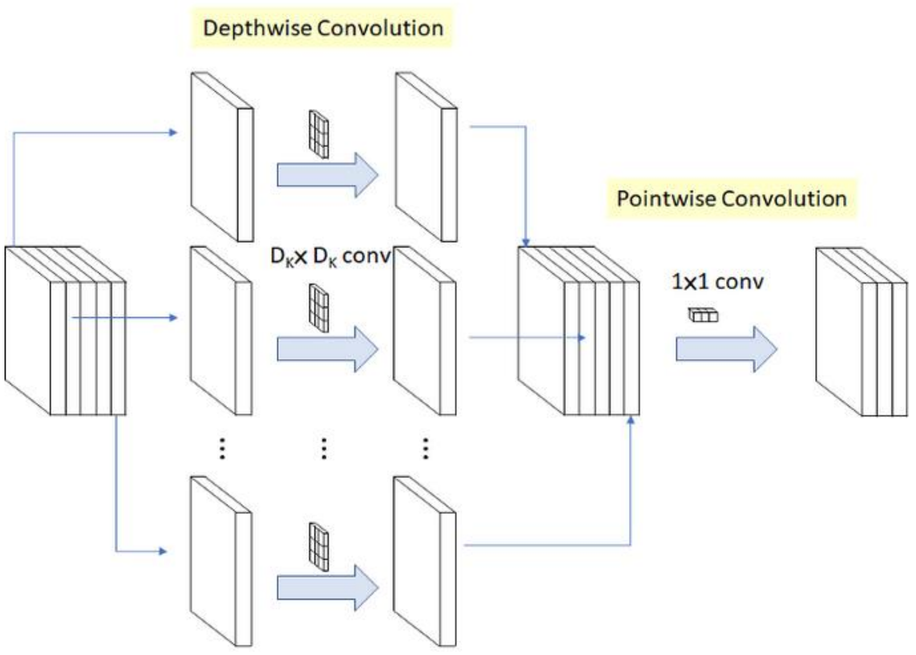
Classification — Fully Connected Layer (FC Layer)



In MobilenNet architecture, Depthwise Separable Convolution is used to reduce the model size and complexity. It is particularly useful for mobile and embedded vision applications.

1. Depthwise Separable Convolution

Depthwise separable convolution is a depthwise convolution followed by a pointwise convolution as follows:



1. Depthwise convolution is the channel-wise DK×DK spatial convolution. Suppose in the figure above, we have 5 channels, then we will have 5 DK×DK spatial convolution.

2. Pointwise convolution actually is the 1×1 convolution to change the dimension.

With above operation, the operation cost is:

$$D_K \cdot D_K \cdot M \cdot D_F \cdot D_F + M \cdot N \cdot D_F \cdot D_F$$

where M: Number of input channels, N: Number of output channels, DK: Kernel size, and DF: Feature map size.

For standard convolution, it is:

$$D_K \cdot D_K \cdot M \cdot N \cdot D_F \cdot D_F$$

Thus, the computation reduction is:

$$\frac{D_K \cdot D_K \cdot M \cdot D_F \cdot D_F + M \cdot N \cdot D_F \cdot D_F}{D_K \cdot D_K \cdot M \cdot N \cdot D_F \cdot D_F}$$
$$= \frac{1}{N} + \frac{1}{D_K^2}$$

When DK×DK is 3×3, 8 to 9 times less computation can be achieved, but with only small reduction in accuracy.

We will use Convolutional Neural Networks, or CNNs, because were designed to map image data to an output variable. They have proven so effective that they are the go-to method for any type of prediction problem involving image data as an input. Also the depthwise convolution approach makes it really light weight for our live video frame classification.

How to Convert Categorical Data to Numerical Data?

This involves two steps:

1. Integer Encoding
2. One-Hot Encoding

1. Integer Encoding
As a first step, each unique category value is assigned an integer value.

For example, “*red*” is 1, “*green*” is 2, and “*blue*” is 3.

This is called a label encoding or an integer encoding and is easily reversible.

For some variables, this may be enough.

The integer values have a natural ordered relationship between each other and machine learning algorithms may be able to understand and harness this relationship.

For example, ordinal variables like the “place” example above would be a good example where a label encoding would be sufficient.

2. One-Hot Encoding

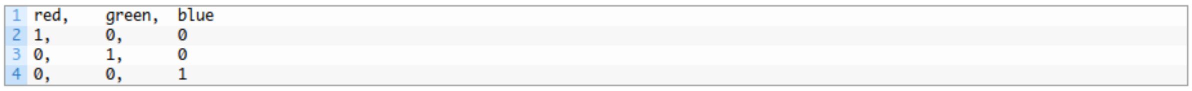
For categorical variables where no such ordinal relationship exists, the integer encoding is not enough.

In fact, using this encoding and allowing the model to assume a natural ordering between categories may result in poor performance or unexpected results (predictions halfway between categories).

In this case, a one-hot encoding can be applied to the integer representation. This is where the integer encoded variable is removed and a new binary variable is added for each unique integer value.

In the “*color*” variable example, there are 3 categories and therefore 3 binary variables are needed. A “1” value is placed in the binary variable for the color and “0” values for the other colors.

For example:

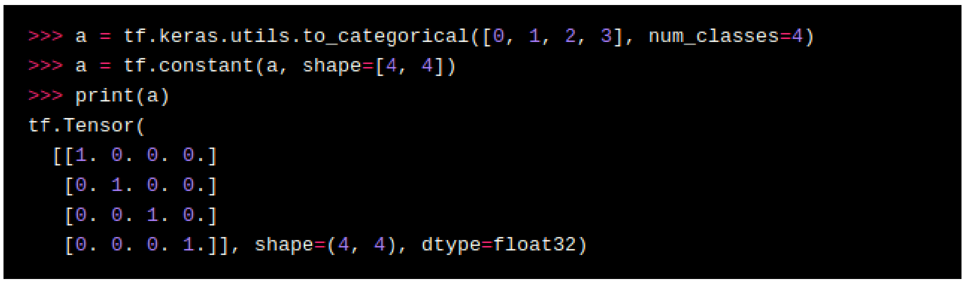


```
red,    green,  blue
1,      0,      0
0,      1,      0
0,      0,      1
```

to_categorical function[*]

tf.keras.utils.to_categorical(y, num_classes=None, dtype="float32")

Converts a class vector (integers) to binary class matrix.



```
>>> a = tf.keras.utils.to_categorical([0, 1, 2, 3], num_classes=4)
>>> a = tf.constant(a, shape=[4, 4])
>>> print(a)
tf.Tensor(
  [[1. 0. 0. 0.]
   [0. 1. 0. 0.]
   [0. 0. 1. 0.]
   [0. 0. 0. 1.]], shape=(4, 4), dtype=float32)
```

We will use One Hot encoding for our project because the natural language labels (mask and without mask) should be transformed into logits in order to be used by the neural network architecture.

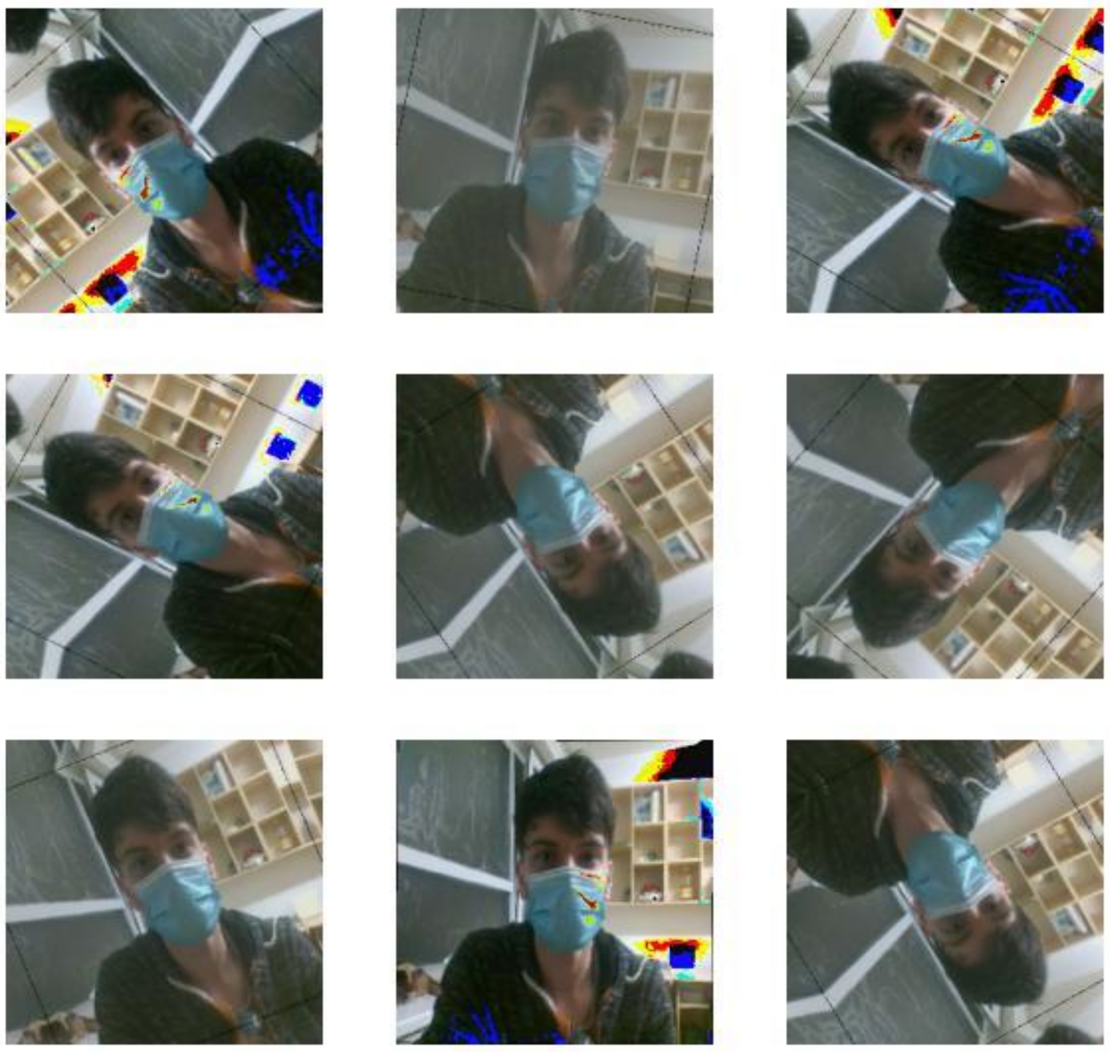
What is data augmentation?

Data augmentation is a technique to increase the diversity of your training set by applying random (but realistic) transformations such as image rotation.

```
data_augmentation = tf.keras.Sequential([
    layers.Input(shape=(224, 224, 3)) ,
  layers.experimental.preprocessing.RandomFlip("horizontal_and_vertical"),
  tf.keras.layers.experimental.preprocessing.RandomContrast(0.4),
  layers.experimental.preprocessing.RandomRotation(0.2),
])
```

The above created model outputs a random fliped, with contrast and rotated image for every input.

```
%matplotlib inline
plt.figure(figsize=(10, 10))
for i in range(9):
  augmented_image = data_augmentation(data)
  ax = plt.subplot(3, 3, i + 1)
  plt.imshow(augmented_image[0].numpy().astype("uint8"))
  plt.axis("off")
```



We will use data augmentation on our current dataset because we want more training data since it is known that the performance of a neural network is direct proportional with the size of the training data.

Is it gonna produce accurate predictions for every position of the face?

Yes and no. Using the image augmentation technique we make sure that our model is agile to subtle changes in the positional input, beside the actual face there is a lot of noise in the image like the relative background. We need to use another model (with the same convolutional neural networks based architecture) to select the specific set of pixels that represent the face.We use this method because the model will be much faster with the small amount of imput, but also it allows us to draw a nice box bounding the detected face.

Signaling the user for not wearing the mask with arduino

In computing, serialization the process of translating a data structure or object state into a format that can be stored or transmitted and reconstructed later (possibly in a different computer environment). When the resulting series of bits is reread according to the serialization format, it can be used to create a semantically identical clone of the original object.

Computers and communication equipment represent characters using a character encoding that assigns each character to something — an integer quantity represented by a sequence of digits, typically — that can be stored or transmitted through a network. Two examples of usual encodings are ASCII and the UTF-8 encoding for Unicode.

Since strings are sequences of characters we can assume that strings can be safely transmitted through a serial interface.

All Arduino boards have at least one serial port (also known as a UART or USART), and some have several so we can communicate with arduino in serial format that represents strings.

Design

Below is the MobileNet Architecture:

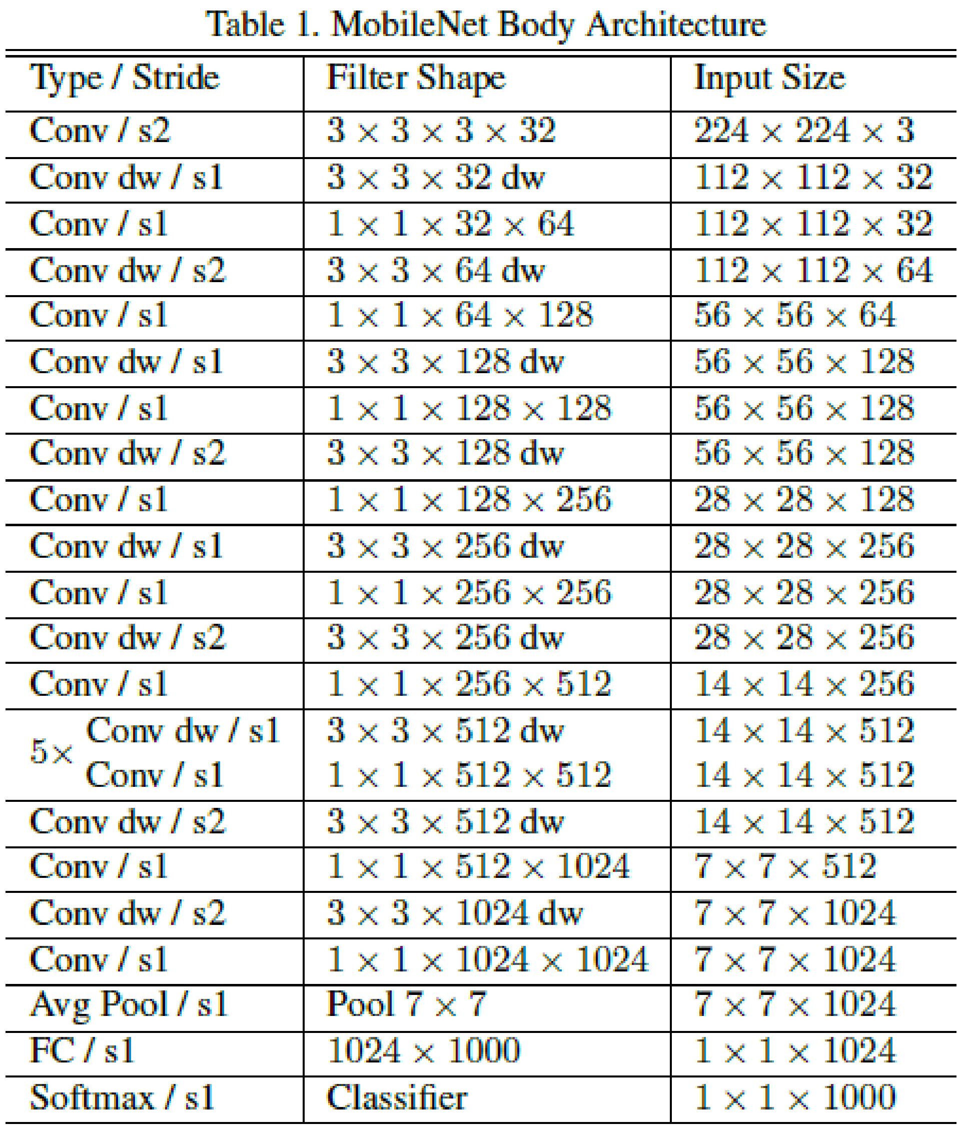


Table 1. MobileNet Body Architecture

Type / Stride	Filter Shape	Input Size
Conv / s2	$3 \times 3 \times 3 \times 32$	$224 \times 224 \times 3$
Conv dw / s1	$3 \times 3 \times 32$ dw	$112 \times 112 \times 32$
Conv / s1	$1 \times 1 \times 32 \times 64$	$112 \times 112 \times 32$
Conv dw / s2	$3 \times 3 \times 64$ dw	$112 \times 112 \times 64$
Conv / s1	$1 \times 1 \times 64 \times 128$	$56 \times 56 \times 64$
Conv dw / s1	$3 \times 3 \times 128$ dw	$56 \times 56 \times 128$
Conv / s1	$1 \times 1 \times 128 \times 128$	$56 \times 56 \times 128$
Conv dw / s2	$3 \times 3 \times 128$ dw	$56 \times 56 \times 128$
Conv / s1	$1 \times 1 \times 128 \times 256$	$28 \times 28 \times 128$
Conv dw / s1	$3 \times 3 \times 256$ dw	$28 \times 28 \times 256$
Conv / s1	$1 \times 1 \times 256 \times 256$	$28 \times 28 \times 256$
Conv dw / s2	$3 \times 3 \times 256$ dw	$28 \times 28 \times 256$
Conv / s1	$1 \times 1 \times 256 \times 512$	$14 \times 14 \times 256$
5× Conv dw / s1	$3 \times 3 \times 512$ dw	$14 \times 14 \times 512$
Conv / s1	$1 \times 1 \times 512 \times 512$	$14 \times 14 \times 512$
Conv dw / s2	$3 \times 3 \times 512$ dw	$14 \times 14 \times 512$
Conv / s1	$1 \times 1 \times 512 \times 1024$	$7 \times 7 \times 512$
Conv dw / s2	$3 \times 3 \times 1024$ dw	$7 \times 7 \times 1024$
Conv / s1	$1 \times 1 \times 1024 \times 1024$	$7 \times 7 \times 1024$
Avg Pool / s1	Pool 7×7	$7 \times 7 \times 1024$
FC / s1	1024×1000	$1 \times 1 \times 1024$
Softmax / s1	Classifier	$1 \times 1 \times 1000$

Whole Network Architecture for MobileNet

We use the pretarined weights of the model to bring our image into the desired encoded form then instead of using the fully conected layer to compute classifications for 1000 labels we use only 2 (with-mask and without-mask) and then we train those weights for our dataset.

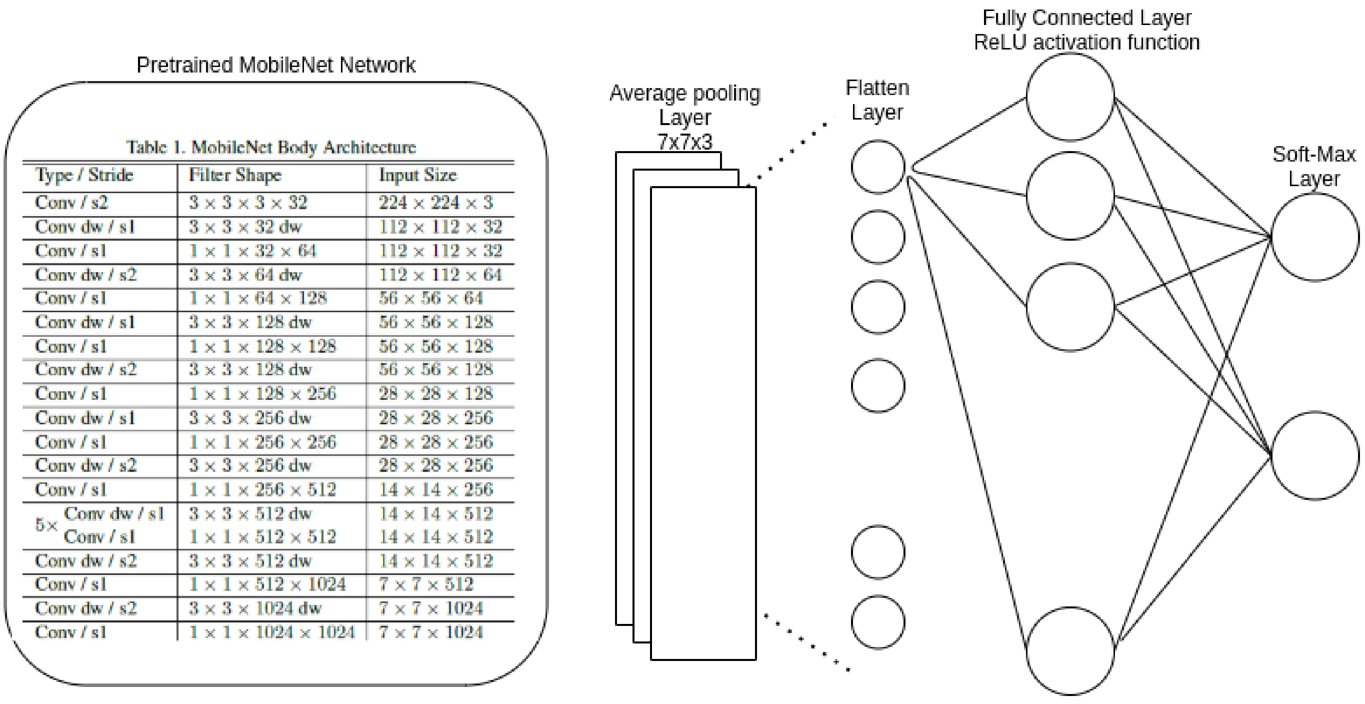

Pretrained MobileNet Network
Average pooling Layer 7x7x3
Flatten Layer
Fully Connected Layer ReLU activation function
Soft-Max Layer

Table 1. MobileNet Body Architecture

Type / Stride	Filter Shape	Input Size
Conv / s2	3 × 3 × 3 × 32	224 × 224 × 3
Conv dw / s1	3 × 3 × 32 dw	112 × 112 × 32
Conv / s1	1 × 1 × 32 × 64	112 × 112 × 32
Conv dw / s2	3 × 3 × 64 dw	112 × 112 × 64
Conv / s1	1 × 1 × 64 × 128	56 × 56 × 64
Conv dw / s1	3 × 3 × 128 dw	56 × 56 × 128
Conv / s1	1 × 1 × 128 × 128	56 × 56 × 128
Conv dw / s2	3 × 3 × 128 dw	56 × 56 × 128
Conv / s1	1 × 1 × 128 × 256	28 × 28 × 128
Conv dw / s1	3 × 3 × 256 dw	28 × 28 × 256
Conv / s1	1 × 1 × 256 × 256	28 × 28 × 256
Conv dw / s2	3 × 3 × 256 dw	28 × 28 × 256
Conv / s1	1 × 1 × 256 × 512	14 × 14 × 256
5× Conv dw / s1	3 × 3 × 512 dw	14 × 14 × 512
Conv / s1	1 × 1 × 512 × 512	14 × 14 × 512
Conv dw / s2	3 × 3 × 512 dw	14 × 14 × 512
Conv / s1	1 × 1 × 512 × 1024	7 × 7 × 512
Conv dw / s2	3 × 3 × 1024 dw	7 × 7 × 1024
Conv / s1	1 × 1 × 1024 × 1024	7 × 7 × 1024

What was the MobileNet model previously trained on:

ImageNet dataset[*]

ImageNet is a project which aims to provide a large image database for research purposes. It contains more than 14 million images which belong to more than 20,000 classes (or synsets). They also provide bounding box annotations for around 1 million images, which can be used in Object Localization tasks. It should be noted that they only provide urls of images and you need to download those images.

The pretrained model previously learned the features for the 20, 000 classes that include persons and faces and it is a much better starting point for our weights than random initialisation.

Like mentioned in the image augmentation section we will speed up even more the time detection by selecting the subset of pixels that are corelated to a human face using a pretraned CNN-based model.

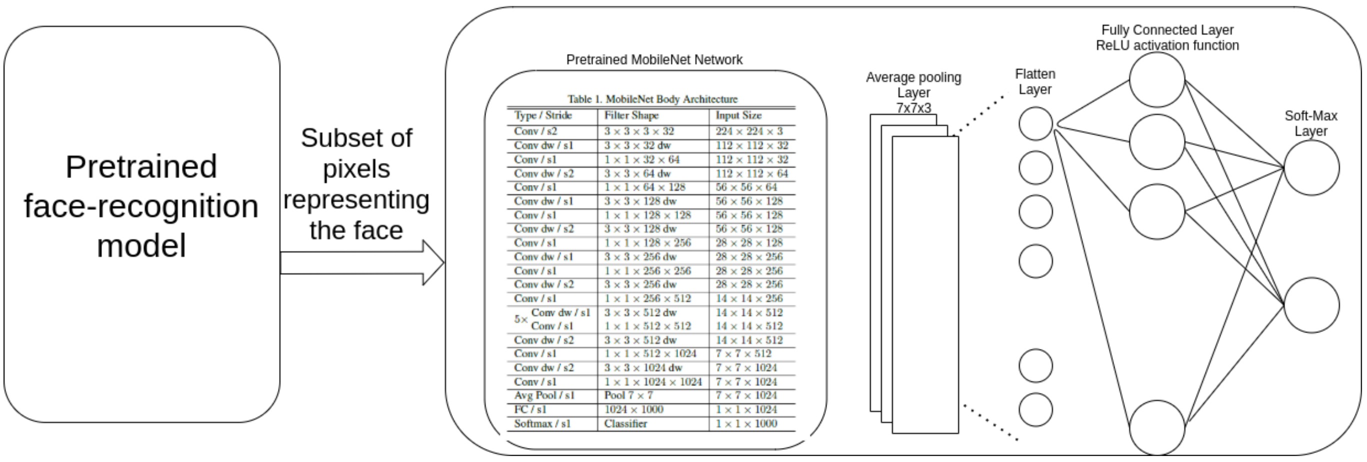
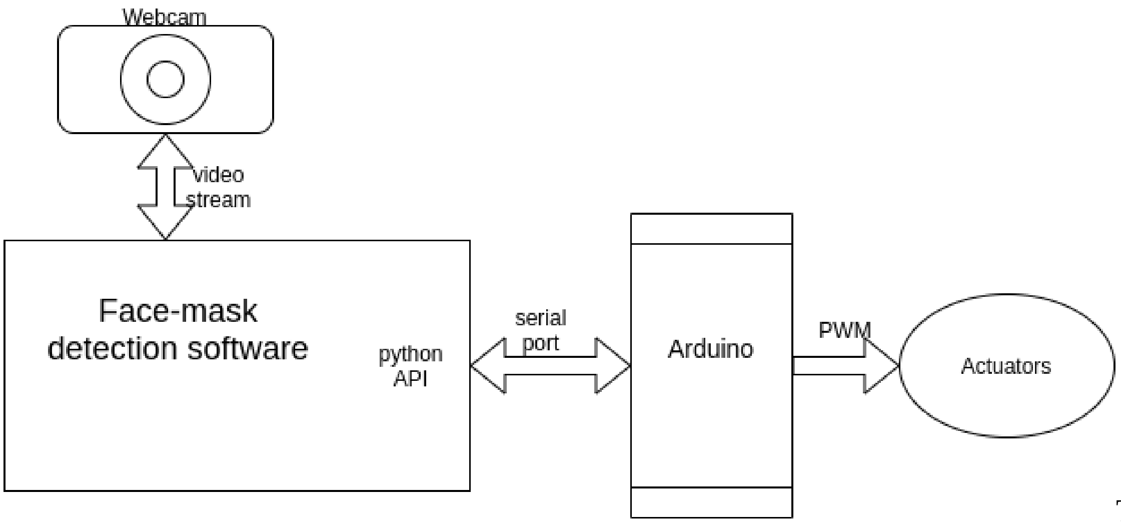

Pretrained face-recognition model
Subset of pixels representing the face
Pretrained MobileNet Network
Average pooling Layer 7x7x3
Flatten Layer
Fully Connected Layer ReLU activation function
Soft-Max Layer

Table 1. MobileNet Body Architecture

Type / Stride	Filter Shape	Input Size
Conv / s2	3 × 3 × 3 × 32	224 × 224 × 3
Conv dw / s1	3 × 3 × 32 dw	112 × 112 × 32
Conv / s1	1 × 1 × 32 × 64	112 × 112 × 32
Conv dw / s2	3 × 3 × 64 dw	112 × 112 × 64
Conv / s1	1 × 1 × 64 × 128	56 × 56 × 64
Conv dw / s1	3 × 3 × 128 dw	56 × 56 × 128
Conv / s1	1 × 1 × 128 × 128	56 × 56 × 128
Conv dw / s2	3 × 3 × 128 dw	56 × 56 × 128
Conv / s1	1 × 1 × 128 × 256	28 × 28 × 128
Conv dw / s1	3 × 3 × 256 dw	28 × 28 × 256
Conv / s1	1 × 1 × 256 × 256	28 × 28 × 256
Conv dw / s2	3 × 3 × 256 dw	28 × 28 × 256
Conv / s1	1 × 1 × 256 × 512	14 × 14 × 256
5× Conv dw / s1	3 × 3 × 512 dw	14 × 14 × 512
Conv / s1	1 × 1 × 512 × 512	14 × 14 × 512
Conv dw / s2	3 × 3 × 512 dw	14 × 14 × 512
Conv / s1	1 × 1 × 512 × 1024	7 × 7 × 512
Conv dw / s2	3 × 3 × 1024 dw	7 × 7 × 1024
Conv / s1	1 × 1 × 1024 × 1024	7 × 7 × 1024
Avg Pool / s1	Pool 7 × 7	7 × 7 × 1024
FC / s1	1024 × 1000	1 × 1 × 1024
Softmax / s1	Classifier	1 × 1 × 1000

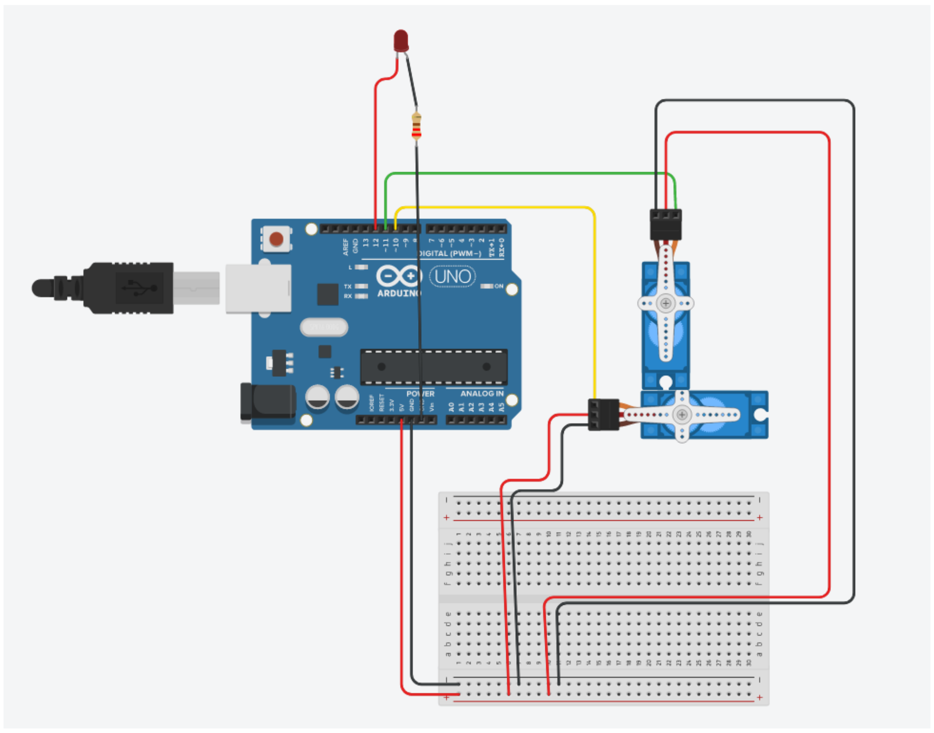
Introducing Arduino microcontroller



The design of our project is very simple. Out Face-mask detection software detects from the video stream of the webcam if the user has or not the mask on and also the position of her face. Next we use python APIs to send the custom position of the center of the face through the serial port to arduino. The arduino board will receive a string every N milliseconds that will contain the coordinates of the center of the face. With a minimum logic arduino parses the string and sends the coordinates along with the boolean value of wearing a mask or not to the actuators in order to perform the task of alerting the user.

As actuators we choose to use for this version two servomotors and a led. The led is on when the mask is off and it is positioned by the servomotors perpendicular with the field of view of the user in order to alert him/her.

Tinkercad Schema



Implementation

Face-Mask Classiffication Software

The Face-Mask Classiffication Software will be split into two parts:

- The part responsible for the training of our model
- The part that implements takes information from the web camera and feeds it to the classifier getting all the information that we need.

The training part

This part will be implemented into the train_model.py file and has the principar purpose to build and train the image face-mask classifier.

For training we need a dataset (it will be composed from preexistent datasets from the internet) that will be present in the working folder and can be passed to our program via an argument.

```
ap.add_argument("-d", "--dataset", required=True,
   help="path to the dataset")
```

The output model will have it's path for storing also defined via an argument.

```
ap.add_argument("-o", "--output", type=str,
   default="facemask_model.model",
   help="path to the output model")
```

With this solved, the actuall building process of the model start with the definition of the hyperparameters:

- Learning Rate: 1e-4 => As discussed in the previous sections it should be small enough for not skipping the glabal minima of the loss function but also big enough so the training process takes a resonable amount of time to converge
- EPOCHS: the number times that the learning algorithm will work through the entire training dataset – we choose to be 20 because the accuracy yelds above 99% values after this also the loss yields values less than .08
- Batch Size : 32 => the number of training examples utilized in one iteration – since the amount of training data present in memory is limited at one given time, we choose to split in in batches of 32 – it is a standsrd power of 2 value also dependent of the hardware that we use.

After this we need to fetch and preprocess the data.

We initialize the list of paths of the images in the dataset :

```
imagePaths = list(paths.list_images(args["dataset"]))
```

And then for evey image we load the input image (224x224) and preprocess it. The preprocessing method is imported form the tf.keras library that will represent the preprocessing of the imge that is needed for the MobineNet2 model.

```
    image = load_img(imagePath, target_size=(224, 224))

    image = img_to_array(image)

    image = preprocess_input(image)
```

We store them after in a list called data.

Since it is a Supervised Learning model we need to have the labels for those images so as we insert the preprocessed images into the data array we also obtain the labels by fetching the name of the folder that the image was read from:

```
label = imagePath.split(os.path.sep)[-2]

labels.append(label)
```

The names shoud be with_mask and wothout_mask but it can be any 2 values since we transform those words into logits using the One Hot Encoding method mentioned in the previous sections:

```
lb = LabelBinarizer()

labels = lb.fit_transform(labels)

labels = to_categorical(labels)
```

We then split the data into training and testing data in proportion of 80-20 percent, a standart for those classifiers and we want to shuffle it (for our model to not be order biased) with a random state equal with 42 (usually is 42) .

```
(trainX, testX, trainY, testY) = train_test_split(data, labels,

    test_size=0.20, stratify=labels, random_state=42)
```

After the split begins the augmentation part the data as explained in the above sections we do this using ImageDataGenerator from keras where we specify the rotation_range (how much the augmented image can be rotated) zoom_range, the space shifts if it can be flipped etc. As discussed it will generate augmented images for a bigger dataset and a better generalisation of the image.

```
aug = ImageDataGenerator(

    rotation_range=20,

    zoom_range=0.15,
    width_shift_range=0.2,
    height_shift_range=0.2,
    shear_range=0.15,
    horizontal_flip=True,
    fill_mode="nearest")
```

We than are ready to load the MobileNetV2 model with weights trained from imagenet dataset and with an input layer that consists of images 224x244 px matrix dimension with a color RGB depth 3 .

```
mobilenet = MobileNetV2(weights="imagenet", include_top=False,
input_tensor=Input(shape=(224, 224, 3)))
```

On top of that we construct the classifier part that takes the output from the MobileNet model, gets it through a pooling layer size 7x7 then it flattens it for the fully connected layer. We also add a dropout of 0.5 to it that will induce a probability of 50% for a perceptron to be innactive during a training iteration so we are not overfitting.

```
classifier = mobilenet.output
classifier = AveragePooling2D(pool_size=(7, 7))(classifier)
classifier = Flatten(name="flatten")(classifier)
classifier = Dense(128, activation="relu")(classifier)
classifier = Dropout(0.5)(classifier)
classifier = Dense(2, activation="softmax")(classifier)
```

It is mandatory to freeze (block training) the weights of the MobileNet during the backpropagation process because they are already pretrained to extract the image features.

```
for layer in mobilenet.layers:
   layer.trainable = False
```

We choose the Adam optimizer discussed in the previous sections:

```
opt = Adam(lr=L_RATE, decay=L_RATE / EPOCHS)
model.compile(loss="binary_crossentropy", optimizer=opt,
   metrics=["accuracy"])
```

And we begin the training of the model

```
Mtrain = model.fit(
    aug.flow(trainX, trainY, batch_size=BS),
    steps_per_epoch=len(trainX) // BS,
    validation_data=(testX, testY),
    validation_steps=len(testX) // BS,
    epochs=EPOCHS)
```

Having of course the training data for train and test data for validation so the testing is performed on data that was not saw previously by the model.

After the training we save the model in format h5.

```
model.save(args["output"], save_format="h5")
```

And now our model is ready to be used to classify images with persons with mask or without.

The mask_detect.py file contains the logic for :

- Localising a face on an image and perform classification on that
- Get the video stream from the web cam so we can get the frames
- Send data to the serial port about the position and the classification of the face

We have a few key parameters:

- --face : that represents the path to the pretrained model that will detect the pixels representing a face
- --model : that represents the path to the classifier that we trained
- --confidence : represents a treshold for face detection confidence set by default to 0.5 which means 50% confidence
- --port : the serial port that we will trainsmit data – it will be set to /dev/tttyUSB0 by default because this is the port that arduino uses.

First we want to load the face detection model.

```
        prototxtPath = os.path.sep.join([args["face"], "deploy.prototxt"])
        weightsPath = os.path.sep.join([args["face"],
           "res10_300x300_ssd_iter_140000.caffemodel"])
faceModel = cv2.dnn.readNet(prototxtPath, weightsPath)
```

Then our classifier:

```
        maskModel = load_model(args["model"])
```

We want to initialize the video stream and wait for it to start:

```
vs = VideoStream(src=0).start()
time.sleep(2.0)
```

Then we want to loop through the frames of the videos and perform detection and classification.

```
while True:
   # get the frame from the threaded video stream and resize it
   # to have a maximum width of 600 pixels
   frame = vs.read()
   frame = imutils.resize(frame, width=600)
...
```

The detection and classification is implemented in the function detect_face_and_classify(frame, faceModel, maskModel):

This function will first use the openCV capability to create a blob from an image/frame.

```
   blob = cv2.dnn.blobFromImage(frame, 1.0, (300, 300), (104.0, 177.0, 123.0))
```

And we simply have it as input to the face model:

```
faceModel.setInput(blob)
face_detections = faceModel.forward()
```

For evey detected face we will check the confidence:

```
if conf > args["confidence"]:
```

Will compute the coordinates of the bounding box of the face:

```
box = face_detections[0, 0, i, 3:7] * np.array([w, h, w, h])
 (startX, startY, endX, endY) = box.astype("int")
```

Then we will clipp it agains the size of the window and then take all the pixels inside that bounding box and feed it to the face-mask classifier after we preprocess it.

```
#get the face pixels
face = frame[startY:endY, startX:endX]
#convert from BGR to RGB
```

```
        face = cv2.cvtColor(face, cv2.COLOR_BGR2RGB)
        #resize the image 244x244 pixels
        face = cv2.resize(face, (224, 224))
        #flatten the image
        face = img_to_array(face)
        #preprocess the image
        face = preprocess_input(face)
        faces.append(face)
    if len(faces) > 0:
        #predict using the mask detection model
        faces = np.array(faces, dtype="float32")
        predicts = maskModel.predict(faces, batch_size=32)
```

At the end the function returns the location of the face by it’s bounding box in order to be drawn on the frame and also the prediction (with or without mask) in the form of probabilities.

(locations, predictions) = detect_face_and_classify(frame, faceModel, maskModel)

Unzipping then afterwards :

startX, startY, endX, endY) = box

(withMask, withoutMask) = prediction

After this we want to draw the box on the frame and write the confidence from the mask classifier to it. We do this using opencv putText and rectangle methods:

cv2.putText(frame, label, (startX, startY - 5), cv2.FONT_HERSHEY_COMPLEX, 0.30, color, 1)

cv2.rectangle(frame, (startX, startY), (endX, endY), color, 1)

Where the label is:

label = "{}: {:.2f}%".format(("Mask" if withMask > withoutMask else "No Mask"), max(withMask, withoutMask) * 100)

For the writing to the port part we implement the writeToPort function that will get the serialX and serialY integer arguments. But first how do we compute them.

We want our actuator to point to the upper center of the face. For this we simply calculate the upper center of the image:

```
    serialX = startX + (endX - startX)/2
```

```
        serialY = startY + (1/3)*(endY - startY)
```

then we have to map it to a rotation the the servo in the x direction and the y direction (the max directional rotaion is 180 degrees):

```
            serialX = 180 - (serialX*180)/w
            serialY = (serialY*180)/h
```

After that we simply pass the arguments to the writeToPort function:

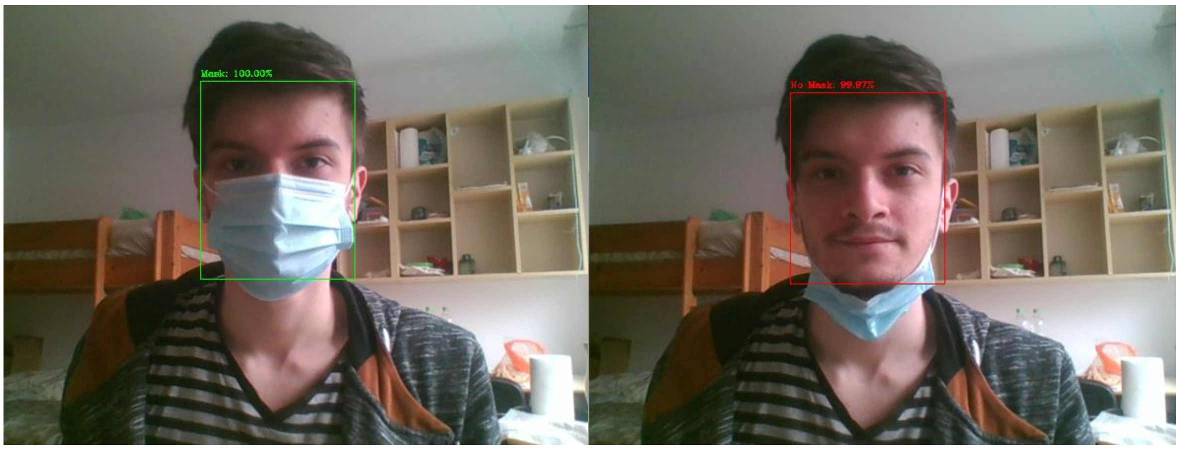
```
def writeToPort(portSer, x, y):
    #by default we got /dev/ttyUSB0
    port = portSer
```

In here we just use the serial library to sent to the specified port with a speed of 9600 (specific for arduino) and encoded string with the coordinates:

```
        # serial speed 9600 as for arduino
        # timeout of 5 second
        ard = Serial(port,9600,timeout=5)
        # the encoded string to be send
        ard.write("X{}Y{}".format(x, y).encode())
```

And this concludes the face detection and mask classification from a video stream form webcam. Using Python API we sent the coordinates of the detected face without mask to our arduino can decode and alert the user.

Here is a side-by-side comparison that represents the with mask case and without mask case:



Arduino Software

The Arduino Software is represented by a simgle file used to program the board:

For the python software we assume that every N milliseconds a string will be sent to the serial port that our arduino is connected to.

If the user wears a mask then the coordinated x=0 and y=0 will be sent thus moving the actuators in an idle state.

Our arduino software begins by importing the Servo.h library in order to control the servomotors.

We will have two servomotors called:

Servo serX;

Servo serY;

We will attach to them the ports 10 and 11:

serX.attach(11);

serY.attach(10);

Every servomotor handles the movement on one axis.

We also have a led as output connected to port 12.

For the serial reading we have to declare a timeout for our reading state. We set it to 20 seconds:

Serial.setTimeout(20);

Whenever a serial event is triggered (like our reading) a function called serialEvent()[4] is called. We will overwrite this function to contain our logic for consuming and parsing our received data.

We will first read the string with the method readString():

serialData = Serial.readString();

Our string will have the following format: “X<coordx>Y<coordy>” where <coordx> and <coordy> consists of the x y coordinates.

For parsing our string we define two separate functions called parseDataX(String data) and parseDataY(String data) that returns the integer value for x and y coordinates:

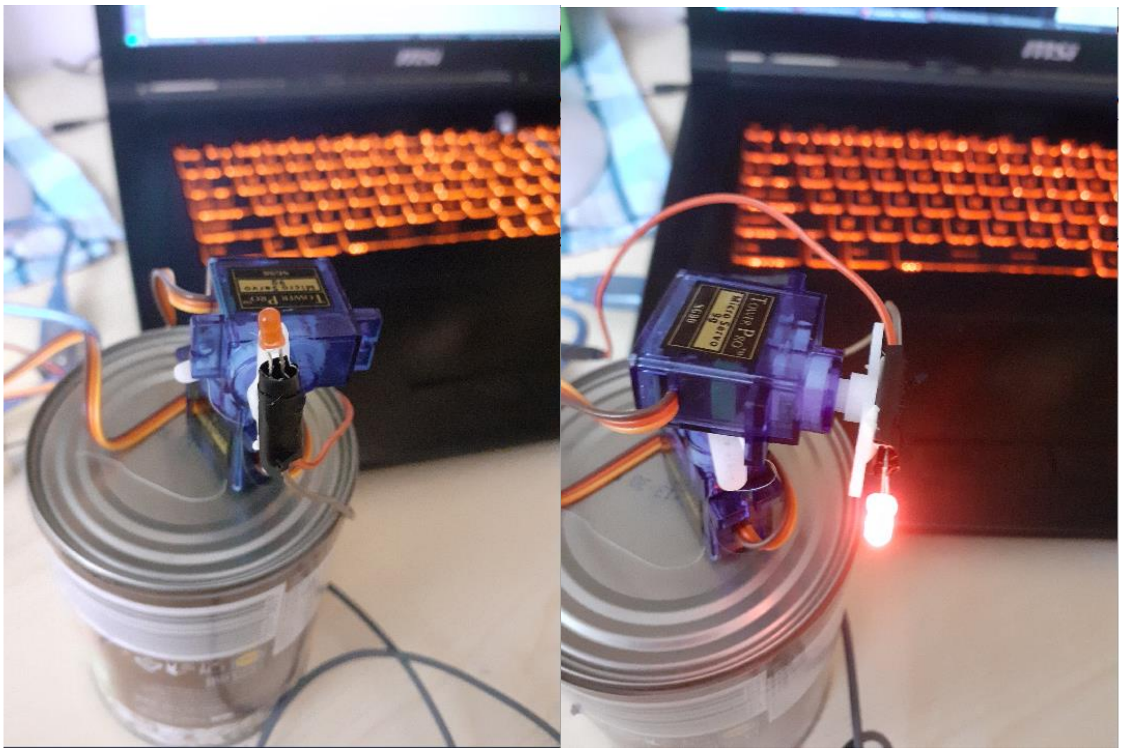
```
int parseDataX(String data) {
  data.remove(data.indexOf("Y"));
  data.remove(data.indexOf("X"), 1);
  return data.toInt();
}
int parseDataY(String data) {
  data.remove(0, data.indexOf("Y") + 1);
  return data.toInt();
}
```

We send the values to the corresponding servomotor with the method write():

```
serY.write(parseDataY(serialData));
```

We also check if the x value is 0 we set the led to LOW since we don’t want to alert the user, otherwise we set it to HIGH.

We have s side-by-side comparison for idle and without-mask mode:



A more comprehensive demo for the entire process can be found at the link.

Testing

As part of the training process after every epoch (i.e after the model parsed the whole training set) we want to test the accuracy of the model as well as the loss. As explained in the previous sections this is the reason why we split the data into 20% testing data and 80% training data. We can use the testing data to test the accuracy of the model without worring that the model overfits the training data thus not generalising the task at hand. If we compute the accuracy for the testing data for every 20 epochs that we use, an ascending trend should be observed. Likewise a descending trend is observed if we plot the loss function against the epochs. This way we can say with certitude that the model can correctly classify unseen faces in conditions restricted by the dataset at hand.

During the training process the accuracy is automatically computed by the keras library for training as well as for testing.

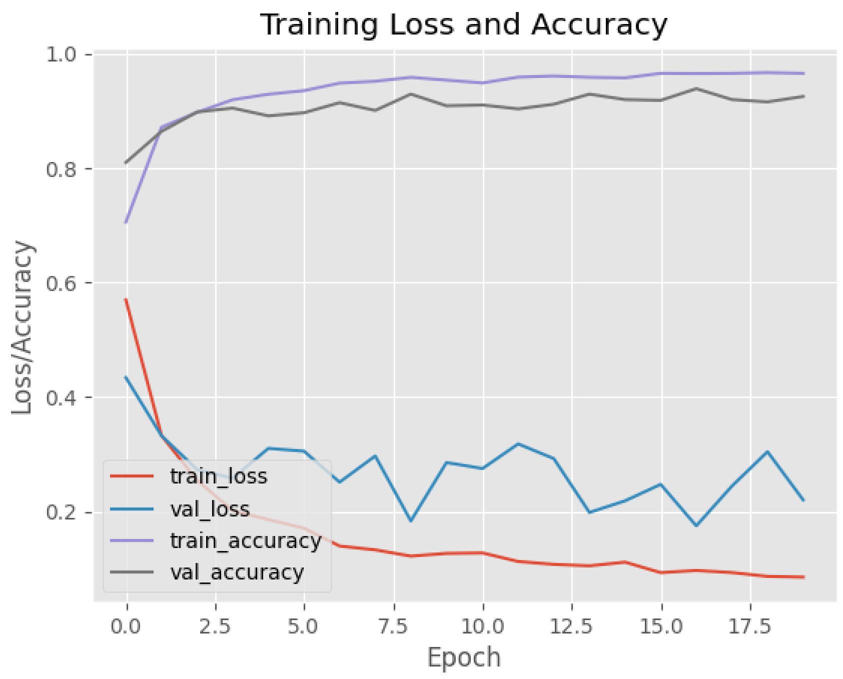
The formula for accuracy is:

$$\text{Accuracy} = \frac{\text{True Positives} + \text{True Negatives}}{\text{All Samples}}$$

After the training is complete we can use matplotlib to plot the results like in the code below.

```
N = EPOCHS
plt.style.use("ggplot")
plt.figure()
plt.plot(np.arange(0, N), Mtrain.history["loss"], label="train_loss")
plt.plot(np.arange(0, N), Mtrain.history["val_loss"], label="val_loss")
plt.plot(np.arange(0, N), Mtrain.history["accuracy"], label="train_acc")
plt.plot(np.arange(0, N), Mtrain.history["val_accuracy"], label="val_acc")
plt.title("Training Loss and Accuracy")
plt.xlabel("Epoch #")
plt.ylabel("Loss/Accuracy")
plt.legend(loc="lower left")
plt.savefig(args["plot"])
```

This outputs the graph:



As expected the accuracy is increasing as the loss decreases. A more proeminent pattern is seen for the training data since the model have seen the data before during training while at the validation step, the model is not updated, this way the images remain new to the model.

Conclusion

We have presented the construction of a face-mask classifier using Keras library with Tensorflow backend that uses transfer learning for acquiring the weights of the Convolutional Neural Network MobileNetV2 for real time high accuracy image classification from the webcam stream. Using an embedded device such as Arduino for real time motion and visual user feedback via the fast serial communication we add a new dimension to the user experience thus cognitively determining the wearing of the face-mask.

Bibliography

Loss function and optimizers:

https://algorithmia.com/blog/introduction-to-loss-functions

https://algorithmia.com/blog/introduction-to-optimizers

https://builtin.com/data-science/gradient-descent

https://en.wikipedia.org/wiki/Gradient_descent

https://towardsdatascience.com/stochastic-gradient-descent-clearly-explained-53d239905d31

AdaGrad optimizer:

https://jmlr.org/papers/v12/duchi11a.html

http://papers.nips.cc/paper/4687-large-scale-distributed-deep-networks.pdf

Adadelta optimizer:

https://arxiv.org/abs/1212.5701

Adam optimizer:

https://arxiv.org/abs/1412.6980

Keras resources:

https://keras.io/guides/

https://keras.io/api/applications/

CNN:

https://en.wikipedia.org/wiki/Convolutional_neural_network

https://towardsdatascience.com/a-comprehensive-guide-to-convolutional-neural-networks-the-eli5-way-3bd2b1164a53

ANN:

https://en.wikipedia.org/wiki/Artificial_neural_network

MobileNet:

https://arxiv.org/abs/1704.04861

https://towardsdatascience.com/review-mobilenetv1-depthwise-separable-convolution-light-weight-model-a382df364b69

Activation Functions:

https://en.wikipedia.org/wiki/Activation_function

https://machinelearningmastery.com/rectified-linear-activation-function-for-deep-learning-neural-networks/

https://medium.com/data-science-bootcamp/understand-the-softmax-function-in-minutes-f3a59641e86d

ImageNet:

http://www.image-net.org/

Servo.h:

https://www.arduino.cc/reference/en/libraries/servo/