Machine Learning and Python: Elucidating the Connection Between Calculus and Programming to Facilitate Artificial Intelligence

Aaryamitra Pateriya

Advance Praise

“Aaryamitra Pateriya’s 'Machine Learning and Python: Elucidating the Connection Between Calculus and Programming to Facilitate Artificial Intelligence' lives up to its title. Written using lucid and simple language, it provides a comprehensive guide to many aspects of Machine Learning (ML) and Artificial Intelligence (AI) and how the applications can be developed and deployed in real-world scenarios using Python. Data scientists and researchers will also find this book very useful in understanding the mathematical foundation of Machine Learning for predictive modelling. The author has elaborated essential mathematical concepts of optimization techniques and fundamentals of neural networks which form the basis of AI and ML. This book is excellent, easy to read, and offers great insights on how AI and ML applications are developed and deployed in the enterprise world."

- Mr. Debajyoti Dhar
Group Head | Signal and Image Processing Group
Space Applications Centre | Indian Space Research Organisation (ISRO) | Govt. of India

"This book is a treatise on the advanced concepts of machine learning. It is informative and easy to understand. It is well-written and easy to follow. I congratulate the author on having written a book on the current state-of-art topic and wish him the best for his future endeavours."

- Mr. Prasun Kumar Gupta
Scientist/Engineer 'SE' | Geoinformatics Department
Indian Institute of Remote Sensing | Indian Space Research Organisation (ISRO) | Govt. of India

"The book succeeds as an exceptionally well-written text for its intended readers. The presentation is exquisitely straightforward and includes interesting basic examples. I strongly recommend that students, particularly those starting their career in the subject, should look into this book."

- Dr. Nutan Kumar Tomar
Associate Professor | Department of Mathematics
Indian Institute of Technology (IIT)

To,

My grandfather, for always having the utmost confidence in me and for always being there.

Table of Contents

2: More OpenCV

Chapter 1: Manipulating Pixels

Understanding the working and implementation of various opencv functions for manipulating image pixels.

Chapter 2: Haar Cascades

Having a thorough look at Haar cascades and implementing their use in object detection.

Chapter 3: Grow Me a Moustache

A quick and simple object detection project.

3: Data Management and Python

Chapter 1: Just Another .py File

Getting started with data management basics using just another python file.

Chapter 2: JSON

Understanding JSON and accessing it with python.

Chapter 3: XML

Understanding XML and accessing it with python.

Chapter 4: CSV and Pandas

Understanding CSV and accessing it with python using standard library and Pandas.

Appendix

Section 1: Cryptography

Exploring cryptography and implementing it with python.

Section 2: Overclocking

Understanding and achieving an overclocked CPU.

Section 3: Rise of the Machines

Questioning their “intelligence”.

Preface

Introduction

"No great discovery was ever made without a bold guess."

– Sir Isaac Newton

It is fascinating to try and understand just the concept of Machine Learning, which I describe as an effort to reach the goal of inducing consciousness into a computer. So far, this effort has been successful in achieving that goal to a noteworthy extent. It took algorithms that were constructed under the guidance of calculus. Machine Learning models are supposed to make guesses, and calculus, or the study of change in mathematical systems/interpretations, helps in making these guesses "bold". What calculus does, is it creates a process bounded with certain rules that try to enforce the best configuration for a continuous set of circumstances. In other words, calculus implemented with adequate programming creates a process or algorithm which is sensitive towards "learning". And that is the core and the most productive aspect of our effort.

Truth be told, there is a lot which can make the whole process even more productive. And as I mentioned, there has already been a noteworthy extent of success in our effort. This book promises to cover all of that starting off with the very basics to an exploration of all aspects and techniques of Machine Learning.

Pre-requisites

The programming language of choice in this book is Python. The reader

doesn't require a professional level of experience in Python, but it's expected that all concepts of the language are intelligible. Concepts in calculus, at least till multi-variable, must be clear.

Conventions Used in This Book

The following typographical conventions are used in this book:

`//`

Indicates the start of a section.

`>>`

Indicates the start of a subsection.

Italic

Indicates emphasis, or the beginning of a part of a subsection.

Bold

Indicates emphasis.

Machine Learning

learning /ˈləːnɪŋ/ noun

the acquisition of knowledge or skills through study, **experience**, or being taught.

intelligence /ɪnˈtɛlɪdʒ(ə)ns/ noun

the ability to acquire and apply knowledge and skills.

It's almost like asking the machine to "help itself". To which, the machine replies with a neural network. A neural network model uses a set of magnitudes which can further be tested and learnt from called *features*. Features are extracted by boiling down a given training entity and are treated as input for the neural network. Extraction of features can both be done manually and by the neural network itself. But more on that and how a neural network uses these features to learn from, further in the unit.

Neural Networks

"math, not magic!"

A neural network is fed a set of features, desired outputs and the freedom to calculate possibilities. On behalf of which, the network adjusts a few factors, and learns about the given set of features with respect to the desired outputs. By applying these adjusted factors, the network could summarize information about a new set of features to its "understanding". Say for instance, features extracted from 100 images of dogs are fed to a neural network, which then goes through each image while adjusting a few factors to ensure that the output (classification) points to a desired result (or in this case, a dog). After this process of learning about the images of dogs, the trained neural network can be given features of a *test* image. On this test image, the network would calculate about the possibility of it being of a dog. To summarize: in a neural network, an outcome or prediction is calculated through performing operations on the given test entity using the factors that were adjusted during the process of learning.

When you look deeper into a neural network, you find a bunch of layers. All layers consist and connect to each other via *neurons*. Neurons are nodes of a layer inside a neural network, which can perform tasks depending on the purpose of the layer they are a part of. There can be different kinds of layers sharing different purposes and different number of neurons. Neural networks, due to this acquisition of a vast number and use of neurons in different layers, can work further deeply with features. This allows the network to work most productively with the very boiled down version of the training entity (e.g. an image), which results in better learning. And that should answer the question, "why neural networks?".

A set of features again, is a set of magnitudes which tell us something about their origin (the training entity). Whilst a set of features is being fed to a neuron, each feature is multiplied by a *weight* integer which is the first of the two factors that make it possible for the network to learn. Then, the neuron adds a *bias* integer to the sum of all products of the weights and fed features. The bias is the second factor. Each neuron can have a different and unique bias.

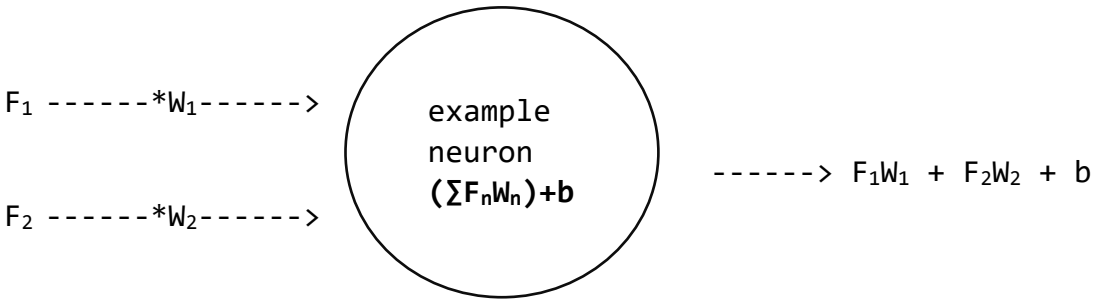
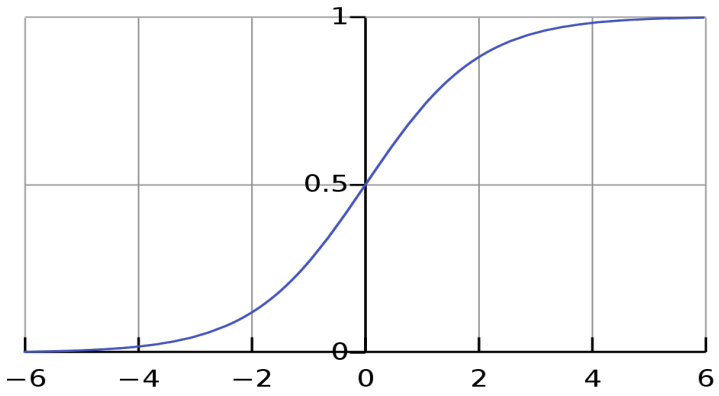


figure 1.1

In figure 1.1, F_1 and F_2 being the two example features, W_1 and W_2 being the two example weights and b being the bias. The result then, is passed through an *activation function* which is usually just a non-linear transformation. For example, the sigmoid function (denoted with σ):

$$\sigma(x) = \frac{1}{1+e^{-x}}$$

This makes the result lie in the range as shown below:



Therefore, the output is simplified and would lie between 0 and 1, which would tell how much the neuron *activates*. So, all in all, a neuron would finally output (z):

$$z = \sigma((\sum F_n W_n) + b)$$

Following the same, our example neuron would give out:

$$\sigma(F_1W_1 + F_2W_2 + b) \quad \ldots(\text{Eq.1})$$

// Getting Somewhere

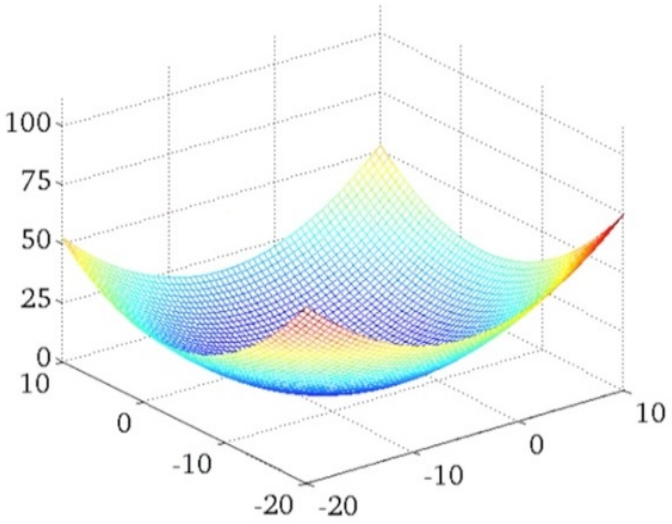
The value of features may remain constant but the value of weights and biases can be changed. Which is how the neural network tunes itself with respect to a desired outcome. Or, it's how the network learns about the given features. The current result (output) helps the neural network tune these two factors most desirably.

// Tuning the Factors (The Learning Part)

We haven't yet set a desired outcome, so from now on, $\boldsymbol{d}$ would be our desired outcome. And our current result as Eq.1 (processed with random values of weights and the bias) with respect to $\boldsymbol{d}$, is just a guess with a certain degree of *error*. This degree of error shows how far away the current result is from $\boldsymbol{d}$ and it can be calculated by using a *cost function* ($\boldsymbol{C}$).

$$\boldsymbol{C} = (\boldsymbol{z} - \boldsymbol{d})^2$$

The goal now, after getting $\boldsymbol{C}$, is to reduce $\boldsymbol{C}$'s value. In other words, the goal now is to reduce the degree of error and therefore tune the network towards the desired outcome. Graphing all the values of $\boldsymbol{C}$ would make a plot in space, while the inputs (weights and biases for $\boldsymbol{z}$) lay in the x and y axis. The minima of this plot would give the least value to which $\boldsymbol{C}$ could be reduced to. Example plot:



Gradient, represented by del (∇), of a function gives its steepest ascent. Therefore, the negative of the gradient would tell the steepest descent. The steepest descent is how much and in which way you should move, in order to see the values of the function decrease the fastest.

Note: $\nabla f(x,y) = < \frac{\partial f(x,y)}{\partial x}, \frac{\partial f(x,y)}{\partial y} >$

// Gradient Descent

$$\boldsymbol{g(C) = -\nabla C}$$

Where, $\boldsymbol{g}$ is the gradient descent function outputting gradient descent for a given value/instance of the cost function. The sign of each output tells us if the weight associated with the respective cost function value should either be increased or decreased. And the value of the output tells us by how much.

// Backpropagation

This is gradient descent in action, an immersive way of calculating $\boldsymbol{g}$ mentioned above. And it can be done with respect to any dependency of the cost function. Examining a single neuron given one feature, one weight and one bias, we start looking at what happens when you wiggle the values of the given weight and the bias.

The result of a neuron given just a single feature, weight and bias ($\boldsymbol{F_n}$, $\boldsymbol{W_n}$ and $\boldsymbol{b}$), without the activation function will be called $\boldsymbol{x}$. Therefore, $\boldsymbol{x = F_n W_n + b}$. And with the activation function, $\boldsymbol{z_n = \sigma(x)}$. The change of the cost function with the new $\boldsymbol{z_n}$, which is $\boldsymbol{C_0 = (z_n - d)^2}$, with respect to the given weight:

$$\frac{\boldsymbol{\partial C_0}}{\boldsymbol{\partial W_n}}$$

Note: Definition of multiple dependency chain/composition rule:

$$for:\ x = x(u,v);\ y = y(u,v);\ z = z(u,v);$$

$$and, f = f(x, y)$$
multiple dependency chain rule would apply as:

$$\frac{\partial f}{\partial u} = \frac{\partial f}{\partial x}\frac{\partial x}{\partial u} + \frac{\partial f}{\partial y}\frac{\partial y}{\partial u}$$

$$\frac{\partial f}{\partial v} = \frac{\partial f}{\partial x}\frac{\partial x}{\partial v} + \frac{\partial f}{\partial y}\frac{\partial y}{\partial v}$$

Brief: *In order to determine the change in a funtion, with respect to another related funtion, write down* ***all*** *dependencies of both the functions and sum the product of each relation* (*in terms of their partial*).

Applying the multiple dependency chain rule to $\frac{\partial C_0}{\partial W_n}$,

$$\frac{\partial \boldsymbol{C_0}}{\partial \boldsymbol{W_n}} = \frac{\partial \boldsymbol{x}}{\partial \boldsymbol{W_n}}\,\frac{\partial \boldsymbol{z_n}}{\partial \boldsymbol{x}}\,\frac{\partial \boldsymbol{C_0}}{\partial \boldsymbol{z_n}}$$

Which is how much the weight $\boldsymbol{W_n}$ wiggles $\boldsymbol{x}$, times how much $\boldsymbol{x}$ wiggles $\boldsymbol{z_n}$, times how $\boldsymbol{z_n}$ wiggles the cost function $\boldsymbol{C_0}$. The measurement becomes possible because $\boldsymbol{C_0}$ depends on $\boldsymbol{z_n}$, which depends on $\boldsymbol{x}$, and $\boldsymbol{x}$ depends on $\boldsymbol{W_n}$. So, we were breaking down from the very bottom to the endpoint, the cost function $\boldsymbol{C_0}$. After differentiating the equation:

$$\frac{\partial \boldsymbol{C_0}}{\partial \boldsymbol{W_n}} = \boldsymbol{F_n}\,\boldsymbol{\sigma'(x)}\,\boldsymbol{2(z_n - d)}$$

Note: $\boldsymbol{\sigma'(x) = \sigma(x)(1 - \sigma(x))}$

Breaking down $\boldsymbol{C_0}$ from the very bottom again, but with respect to the bias instead of the weight would give:

$$\frac{\partial \boldsymbol{C_0}}{\partial \boldsymbol{b}} = \frac{\partial \boldsymbol{x}}{\partial \boldsymbol{b}}\,\frac{\partial \boldsymbol{z_n}}{\partial \boldsymbol{x}}\,\frac{\partial \boldsymbol{C_0}}{\partial \boldsymbol{z_n}}$$

On differentiating:

$$\frac{\partial \boldsymbol{C_0}}{\partial \boldsymbol{b}} = \boldsymbol{\sigma'(x)}\,\boldsymbol{2(z_n - d)}$$

And we could even do the same, as what we did with the weight and bias, to the feature:

$$\frac{\partial C_0}{\partial F_n} = \frac{\partial x}{\partial F_n} \frac{\partial z_n}{\partial x} \frac{\partial C_0}{\partial z_n}$$

On differentiating:

$$\frac{\partial C_0}{\partial F_n} = W_n \, \sigma'(x) \, 2(z_n - d)$$

Instead of examining features going to a neuron, we can examine the output of a neuron going to another neuron. We can call that output, *activation* of that neuron (which, in this case, is equivalent to z_n) and it is denoted by a_l (l being the layer of which the neuron was a part of).

|---------**phase 1**----------| |---------**phase 2**----------|
(input) ----F_n------> (neuron in layer 1) ------a_1-------> (neuron in layer 2)

figure 1.2

Subscript of the activation (output) of the neuron from layer 1 going to the neuron in layer 2 can also be denoted by a_{21}. In the calculations above, x was with respect to F_n, therefore, it stored the output calculated from phase 1 (see figure 1.2). And hence, the calculations above checked how sensitive the weight, bias and even the feature was towards the cost, but only within phase 1. The similar can be done with phase 2 by swapping the weight and bias with respect to the second neuron, and F_n by a_{21}. What we've been doing so far is going from phase 1 to 2, but the network might as well do the same backwards after each complete evaluation by the network. Say, if a_l is the final layer in the network, the sensitivity of the factors towards the cost will be monitored (by doing the previously mentioned calculations) from a_{l-1} to a_l (phase n), then to a_{l-2} to a_{l-1} (phase n-1) and so on. Hence the name, Backpropagation.

// Different Layers and their Tasks

>> Dense layer

A layer formed with neurons outputting $\mathbf{activation\_func}((\sum \mathbf{F_n W_n}) + \mathbf{b})$ (example activation function could be the sigmoid function), is called a Dense layer. We've already discussed deeply about it's working.

>> Convolutional Layer

This layer extracts features, from features. Therefore, it gives out advance features that are better and more informative. Layers which work in this manner contribute to deep learning. A neural network is said to be a class of deep (learning) neural network, when it consists of any layer(s) which analyse the given feature set in an immersive/"in depth" manner and generate further, better features. The deep dive into convolutional layers, and some pre-requisites:

Gradients, in an image:

Features of an image are the pixel values (RGB intensities of each pixel) of that image. The same can be fed to a neural network but instead, we dig out more information from these features, and create advance features. Differences between neighbouring pixels can be very useful in this case. In fact, the difference in value between neighbouring pixels is called an *image gradient.* Pixel values usually differ in significance at boundary of objects, when there is a shadow, within a pattern, or on a textured surface in that image.

To compute an image gradient, we separately calculate the differences along the x-axis and the y-axis of the image and then suffice them back into a 2D vector. Calculating differences (separately) along the axes are 1D operations that can be conducted by using a vector filter, also called a kernel or a mask. For instance, difference between the left neighbour and the right neighbour or the up neighbour and the down neighbour (depends on which axis you're applying the filter on) can be calculated by using the filter $[1, 0, -1]$, a 1D gradient filter. In order to apply a filter to an image, we perform a *convolution.*

Convolution:

The convolutional operator (denoted by $*$) takes 2 functions as input and outputs a new function. It flips (reverses) one of the input functions, moves

it across the other function, and outputs the total area under the multiplied curves at each point:

$$(\boldsymbol{f} * \boldsymbol{g})(\boldsymbol{t}) = \boldsymbol{f}(\boldsymbol{t}) * \boldsymbol{g}(\boldsymbol{t}) = \int_{-\infty}^{\infty} \boldsymbol{f}(\boldsymbol{\tau})\boldsymbol{g}(\boldsymbol{t}-\boldsymbol{\tau})\boldsymbol{d\tau} = \int_{-\infty}^{\infty} \boldsymbol{g}(\boldsymbol{\tau})\boldsymbol{f}(\boldsymbol{t}-\boldsymbol{\tau})\boldsymbol{d\tau}$$

It does not matter which input function is flipped (reversed), the output will be the same because the operator is symmetric in the inputs.

Now, before you rage-quit and close the book, I just want you to know that the convolution used in computing image gradients from a 1D filter is much simpler. Which is, flipping the filter and taking the inner product with a small patch of the image, then moving to the next patch. Example:

$$example\ filter = [a\ b\ c];\ small\ patch\ of\ the\ image = [1\ 2\ 3]$$

$$[\boldsymbol{a}\ \boldsymbol{b}\ \boldsymbol{c}] * [\mathbf{1}\ \mathbf{2}\ \mathbf{3}] = \boldsymbol{c} * \mathbf{1} + \boldsymbol{b} * \mathbf{2} + \boldsymbol{a} * \mathbf{3}$$

Therefore, the **x** and **y** gradients at pixel (**i**, **j**) are (using $[1, 0, -1]$ as the filter):

$$\boldsymbol{g_x}(\boldsymbol{i},\boldsymbol{j}) = [\mathbf{1}\ \ \mathbf{0}\ \ -\mathbf{1}] * [\boldsymbol{I}(\boldsymbol{i}-\mathbf{1},\boldsymbol{j})\ \boldsymbol{I}(\boldsymbol{i},\boldsymbol{j})\ \boldsymbol{I}(\boldsymbol{i}+\mathbf{1},\boldsymbol{j})] = -\mathbf{1} * \boldsymbol{I}(\boldsymbol{i}-\mathbf{1},\boldsymbol{j}) + \mathbf{1} * \boldsymbol{I}(\boldsymbol{i}+\mathbf{1},\boldsymbol{j})$$

$$\boldsymbol{g_y}(\boldsymbol{i},\boldsymbol{j}) = [\mathbf{1}\ \ \mathbf{0}\ \ -\mathbf{1}] * [\boldsymbol{I}(\boldsymbol{i},\boldsymbol{j}-\mathbf{1})\ \boldsymbol{I}(\boldsymbol{i},\boldsymbol{j})\ \boldsymbol{I}(\boldsymbol{i},\boldsymbol{j}+\mathbf{1})] = -\mathbf{1} * \boldsymbol{I}(\boldsymbol{i},\boldsymbol{j}-\mathbf{1}) + \mathbf{1} * \boldsymbol{I}(\boldsymbol{i},\boldsymbol{j}+\mathbf{1})$$

And therefore,

$$\boldsymbol{\nabla I}(\boldsymbol{i},\boldsymbol{j}) = \begin{bmatrix} \boldsymbol{g_x}(\boldsymbol{i},\boldsymbol{j}) \\ \boldsymbol{g_y}(\boldsymbol{i},\boldsymbol{j}) \end{bmatrix}$$

Euclidean norm (magnitude) of this function, which is $\sqrt{g_x^2 + g_y^2}$, indicates how much the pixel value changes around that pixel.

The orientation or direction of the gradient is given by (α):

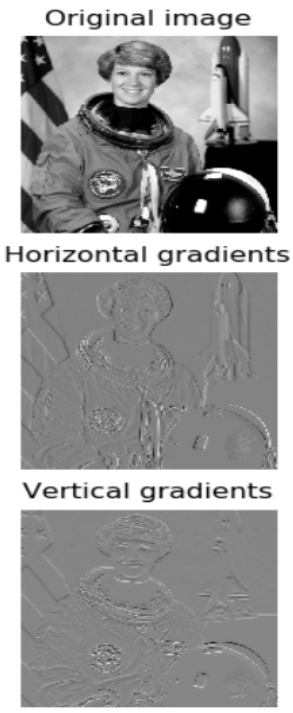
$$\alpha = tan^{-1}\left(\frac{g_y}{g_x}\right)$$

Calculating image gradients using 1D gradient filter in python:

```
### installation for pre-requisite modules ###
#                                            #
#  Python 3.7.7                              #
#  pip install matplotlib==3.1.1             #
#  pip install numpy==1.18.5                 #
#  pip install skimage==0.17.2               #
#                                            #
##############################################

import matplotlib.pyplot as plt
import numpy as np
from skimage import data, color

## Loading example image and converting it into grayscale.
## “data” module from skimage contains image features for certain images.
## In this example, an image of an astronaut.
image = color.rgb2gray(data.astronaut())

## Computing the horizontal gradient (gx):
## The leftmost and the rightmost edges' gradients are set to 0.
## Replacing each non-border pixel with difference between its right
## and left neighbours.
gx  = np.empty(image.shape, dtype=np.double)
gx[:, 0] = 0
gx[:, -1] = 0
gx[:, 1:-1] = image[:, :-2] - image[:, 2:]

## Same with the vertical gradient (gy).
gy  = np.empty(image.shape, dtype=np.double)
gy[0, :] = 0
gy[-1, :] = 0
gy[1:-1, :] = image[:-2, :] - image[2:, :]

## Matplotlib incantations:
fig, (ax1, ax2, ax3) = plt.subplots(3, 1,
                                    figsize=(5,9),
                                    sharex=True,
                                    sharey=True)

ax1.axis('off')
ax1.imshow(image, cmap=plt.cm.gray)
ax1.set_title('Original image')
ax1.set_adjustable('box')

ax2.axis('off')
ax2.imshow(gx, cmap=plt.cm.gray)
```

```
ax2.set_title('Horizontal gradients')
ax2.set_adjustable('box')

ax3.axis('off')
ax3.imshow(gy, cmap=plt.cm.gray)
ax3.set_title('Vertical gradients')
ax3.set_adjustable('box')

plt.show()
```

Shows:



But what if I wanted a 2D gradient filter?

Then we may have to use the integral equation of a convolution. Although, digital images have discrete (non-continuous) pixels, which makes the convolutional integral a discrete sum:

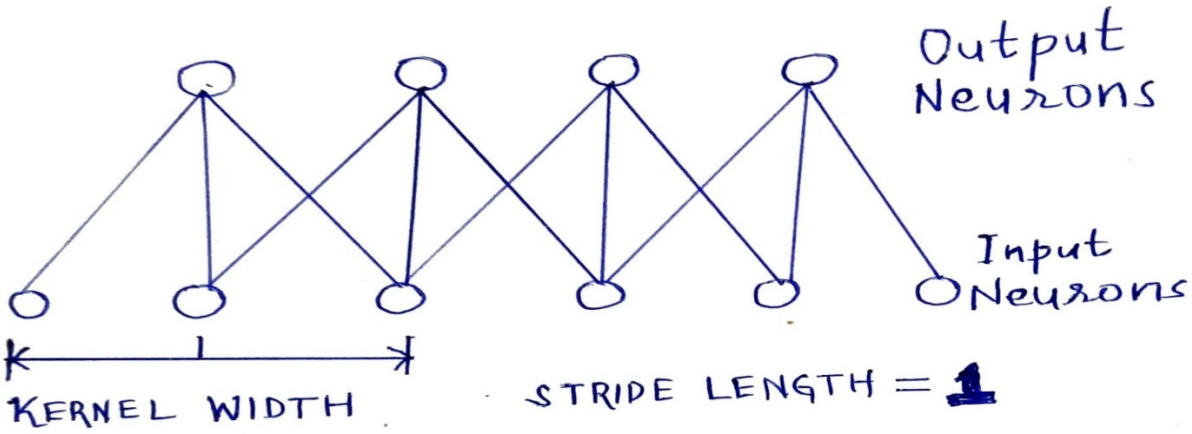
$$(\boldsymbol{f} * \boldsymbol{g})[\boldsymbol{i},\boldsymbol{j}] = \sum_{\boldsymbol{u}=\boldsymbol{0}}^{\boldsymbol{m}} \sum_{\boldsymbol{v}=\boldsymbol{0}}^{\boldsymbol{n}} \boldsymbol{f}[\boldsymbol{u},\boldsymbol{v}]\boldsymbol{g}[\boldsymbol{i}-\boldsymbol{u},\boldsymbol{j}-\boldsymbol{v}]$$

An example of 2d convolutional filter would be the *Gaussian* filter, which produces a weighted average of nearby values. On an image, it has the effect of blurring nearby (varying) pixel values. The 2D Gaussian filter ($\boldsymbol{G}$):

$$G(x, y) = \frac{1}{2\pi\sigma} e^{-\frac{x^2+y^2}{2\sigma^2}}$$

Where, σ is the standard deviation of the Gaussian function.

While applying a filter to an image, you don't necessarily cover the whole image in one go. You take pieces of the image, apply the filter to each piece, and carry on the same process until the whole image is filtered. It's a less clumsy and far cleaner way to filter an image (which is, obtaining advance features). It also makes a clever use of the vast numbers of neurons present in the layer for making the filter move across an image. *Illustration:*



Note: A stride is the number of convoluted pixels the pooling layer slides with, after examining the previous filter block.

A convolutional layer only makes use of a small subset of the inputs for each output and therefore, it is not a fully connected layer. Whereas, a dense layer makes use of the whole set of inputs for each output and hence, is a fully connected layer. The vector of weights and biases for this layer is the filter itself. Instead of each neuron having different weights and biases, a single filter is shared among the neurons. Which again, differs this layer a lot from a dense layer.

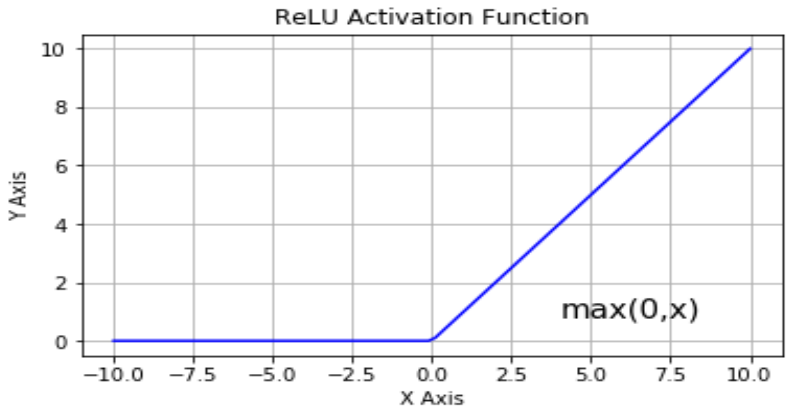
To remember: A convolutional layer does not give a particular pattern

identifying the entirety of an image, but various local patterns computed from small *batches* of an image.

Rectified Linear Unit (ReLU) Activation:

ReLU is another activation function like the sigmoid function, but linear. It is the simplest variation of a linear function where the negative part is simply zeroed out and the positive part is left unbonded.

$$\boldsymbol{ReLU(x) = \max(0, x)}$$



ReLU is a commonly used activation function in convolutional neural networks due its simple characteristics.

>> Normalization and Response Normalization Layers

Normalization, on a set of features, is the process of arranging the features on a simple and similar scale. The easiest bit of normalization would be dividing a set of pixel values by 255 (since a pixel value can only fall between 0 and 255). Which is a very common way of normalization while working with images for getting the work done quick. Although, normalization can be applied to any set of features and suggestively, should be. It improves performance and training stability.
Common normalization techniques ($\boldsymbol{x}$ being the feature set, $\boldsymbol{x'}$ being its normalized result):

Scaling to a Range / Min-Max Scaling:

When you know the approximate upper and lower bounds on your data (feature values), and the data is uniformly distributed across that range, consider scaling to range. Keep in mind, the uniform distribution of the data must exist or else a large portion of the data distribution would be squeezed in one part of the scale. Your normalized result will have values ranging within $[0, 1]$. Formula:

$$x' = \frac{x - x_{min}}{x_{max} - x_{min}}$$

Feature Clipping:

Clipping is, setting all features above (or below) a certain value, to a fixed value (which is, avoiding outliers). Formula:

$$if\ x > max, then\ x' = max. if\ x < min, then\ x' = min$$

Log Scaling:

Also known as the *power law distribution*, log scaling computes the log for each of your feature values and narrows down the range. Formula:

$$x' = \log(x)$$

Z-score:

Represents the number of standard deviations away from the mean. It also ensures that your feature distribution have mean=0 and std=1. Formula:

$$x' = \frac{x - \mu}{\sigma}$$

Note: μ is the mean and σ is the standard deviation.

l^2 *Normalization*:

Divides the feature array by its Euclidean norm. Formula:

$$x' = \frac{x}{\left|\left|x\right|\right|_2}$$

Note: $\left|\left|x\right|\right|_2 = \sqrt{x_1^2 + x_2^2 + \cdots + x_n^2}$

A **Response Normalization Layer**, as the name suggests, works on responses (outputs) in such a way that the strength of each output is measured relative to its neighbours. These responses are obtained from a filter, like a convolutional filter.

Filter Response Normalization with Thresholded Linear Unit:

Note: Feed-forward neural networks are neural networks which display acyclic response flow throughout all layers. We've been working with the same kind so far and will continue to.

Assume a feed-forward convolution neural network produces a 4D tensor **X** of all filter (kernel) responses with shape $[\boldsymbol{B}, \boldsymbol{W}, \boldsymbol{H}, \boldsymbol{C}]$, where $\boldsymbol{B}$ is the batch size, $\boldsymbol{W}$, $\boldsymbol{H}$ define spatial extents of the activation map (filter response) and $\boldsymbol{C}$ is the number of filters (divided over all smalls sets or batches of the training entity) used in the convolution. Let $\boldsymbol{x} = \boldsymbol{X}_{\boldsymbol{b},:,:,\boldsymbol{c}} \, \boldsymbol{\epsilon} \, \boldsymbol{R}^{\boldsymbol{N}}$, where $\boldsymbol{N} = \boldsymbol{W} \, \boldsymbol{x} \, \boldsymbol{H}$, be the vector of filter responses for the c^{th} filter for the b^{th} batch point. Let $\boldsymbol{v}^{\boldsymbol{2}} = \sum_{\boldsymbol{i}} \boldsymbol{x}_{\boldsymbol{i}}^{\boldsymbol{2}} / \boldsymbol{N}$, be the mean squared norm of $\boldsymbol{x}$. Putting it all together, a Filter Response Normalization (FRN) equation can be formed, which is as follows:

$$\hat{\boldsymbol{x}} = \frac{\boldsymbol{x}}{\sqrt{\boldsymbol{v}^{\boldsymbol{2}} + \boldsymbol{\epsilon}}}$$

$\boldsymbol{\epsilon}$ being a small positive constant to prevent division by zero. An affine transformation (or affine mapping) is further applied to this equation so that the network can undo the effects of the normalization:

$$\boldsymbol{y} = \Upsilon \hat{\boldsymbol{x}} + \boldsymbol{\beta}$$

Υ and $\boldsymbol{\beta}$ being learned parameters. Last step to complete the FRN process is to add an activation function. We'll be using ReLU for the same, although due to deficiency of mean centering (sum of particular coordinates for a feature over total number of features), activations can lead to arbitrary bias away from zero. And such a bias conjunction with ReLU can prejudicious effect on learning and lead to poor performance. This is dealt by augmenting ReLU with a learned threshold $\boldsymbol{\tau}$ in the following way:

$$\boldsymbol{z = max(y, \tau)}$$

This is the Thresholded Linear Unit (TLU), which is our final addition to the FRN layer. Since:

$$max(y, \tau) = max(y - \tau, 0) + \tau = ReLU(y - \tau) + \tau,$$

the effect of TLU activation is the same as having a shared bias before and after ReLU. TLU significantly improves the performance of models using FRN.

Local Response Normalization:

Local Response Normalization (LRN) divides a value by a combination of its neighbors or the output of the neighboring filters. Let $x_{i,j}^{k}$ be the output of the kth filter applied at position i, j. Also, let $neighborhood$ be the set of all positions and $y_{i,j}^{k}$ be the normalized result relative to other filters in the neighborhood. Therefore, forming the following equation:

$$\boldsymbol{y_{i,j}^{k} = x_{i,j}^{k} \,/\, \left(c + \alpha \sum_{l \,\epsilon\, neighboorhood\ of\ k} x_{l}^{2} \right)^{\beta}}$$

The size of kernel $neighborhood$, c, α and β being the hyperparameters (in control of the learning process) that are tuned via a validation set of images.

While differentiating over LRN and FRN, one thing you might notice is that LRN does normalization over adjacent output channels at the same spatial location, while FRN is global normalization over the spatial extent.

>> Pooling Layers

Multiple outputs get converted into single output by using a pooling layer. Or, you can say it helps diminish the shape of the output matrix. A pooling layer become useful in models (neural networks) consisting convolutional layers. This is because of the nature of a convolutional layer, which is to filter across an image and generate output for each neighborhood or batch or position. Therefore, a pooling layer boils down the many outputs from their respective positions (normalized or not) to one.

Due to this downsize of outputs, the probability of overfitting the network to training data is efficaciously reduced. Also, pooling layers help connect layers like a convolutional layer which produces a multidimensional output, to layers like a dense layer which are limited to accept 1D inputs.

Multiple ways to pool inputs are: averaging, summing (calculating a generalized norm), or taking the maximum value.

Max Pooling:

Assume that you're given a 4x4 input on which you ran a filter in blocks of 2x2 or with a stride of two convoluted pixels (outputs). A max pooling layer would now move across these 2x2 blocks (having 4 outputs) and pick the output with the maximum value. Hence summarizing what initially was a 4x4 input to 2x2 pool. This example didn't comprise overlapping, but running a 3x3 filter with a stride of one convoluted pixel on the same input would result in overlapping.

// Bibliography

Alice Zheng, Amanda Casari. "Automating the Featurizer: Image Feature Extraction and Deep Learning." *Feature Engineering for Machine Learning: Principles and techniques for data scientists* (O'Reilly Media, Inc., 2019): 133-153.

Saurabh Singh, Shankar Krishnan. "Filter Response Normalization Layer: Eliminating Batch Dependence in the Training of Deep Neural Networks." (Google Research, provided by the Computer Vision Foundation) Source: *https://openaccess.thecvf.com/content_CVPR_2020/papers/Singh_Filter_Response_Normalization_Layer_Eliminating_Batch_Dependence_in_the_Training_CVPR_2020_paper.pdf*

"Normalization." *Data Preparation and Feature Engineering for Machine Learning* Retrieved from: *https://developers.google.com/machine-learning/data-prep/transform/normalization*

Knowing me, Knowing you

we'll just have to face it this time

- ABBA ("Arrival", 1976)

We're now progressing into the hands-on approach of what we've learnt so far. And we'll be using python for the same. More specifically, we'll be using Google's Tensorflow library for python. The entirety of the process will be divided into a number of steps, in order to get a better grasp of what's what and its importance.

// Tensorflow

Released in 2015 by Google Brain Team, Tensorflow is an open-source library which helps in developing Machine Learning models. Tensorflow is capable of doing pretty much everything we've learnt about in machine learning so far, and much more. In brief, it helps in constructing and concatenating neural layers and therefore, creating a neural network which can be further trained and tested. Although, we will be using a few other libraries as well for manual feature extraction and etc.

>> Installation and requirements for Tensorflow

Requirements:

Architecture (for all systems): 64-bit

Python: 3.5-3.8

Operating Systems:

- ✓ Ubuntu 16.04 or later
- ✓ macOS 10.12.6 (Sierra) or later (no GPU support)
- ✓ Windows 7 or later (with C++ redistributable)
- ✓ Raspbian 9.0 or later

Installation:

```
# Current stable release
pip install tensorflow
```

```
# OR, the version I am using
pip install tensorflow==2.3.1
#! Ensure latest pip (if required)
pip install --upgrade pip
```

Find further help on: https://www.tensorflow.org/install

// Goal of the Chapter

Essentially, we'll be building, training and testing a neural network for image classification. The images that the neural network would encounter in the training process would be of human faces, with each face showing a different emotion. The neural network then, should learn to classify the emotions shown by different images. The whole process of building, testing, training and etc. will be broken down into steps. Each step will have its own significance and importance.

// STEP 1: *manual "training and testing" data collection*

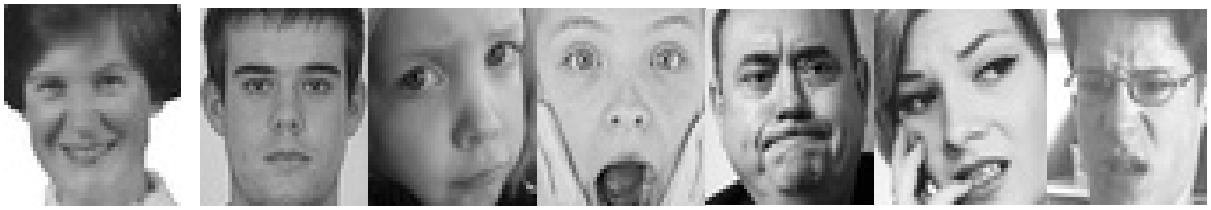
We'll be using Kaggle (in brief, a subsidiary of Google LCC as a dataset repository) to gather data for training and testing purposes.

```
link: https://www.kaggle.com/msambare/fer2013
```

Download the zip file from the link given above (or, the book's GitHub). On extraction, you'll find the following directory tree:

```
├───test
│   ├───angry
│   ├───disgust
│   ├───fear
│   ├───happy
│   ├───neutral
│   ├───sad
│   └───surprise
└───train
    ├───angry
    ├───disgust
    ├───fear
    ├───happy
    ├───neutral
    ├───sad
    └───surprise
```

Each sub-directory in `test` and `train` is named after the emotion expressed by the images that it consists. Also, on an average, each folder consists about 1,200 images. Example image from each sub-directory in `test`:



Sub-directories in `train` would consist similar sort of images. Although, they won't be the exact ones from the `test` sub-directories.

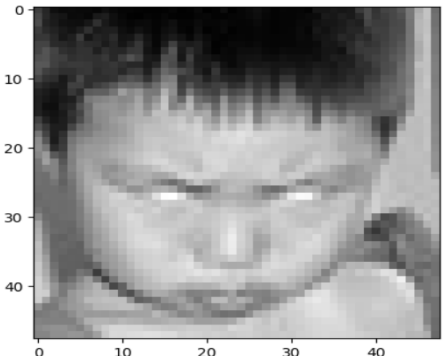
// STEP 2: *manual feature extraction*

We'll now start extracting features from the images that we've just collected. Remember, neural networks work with features, which in the case of image classification are the RGB values of each image. Or, BGR values, as interpreted by opencv.

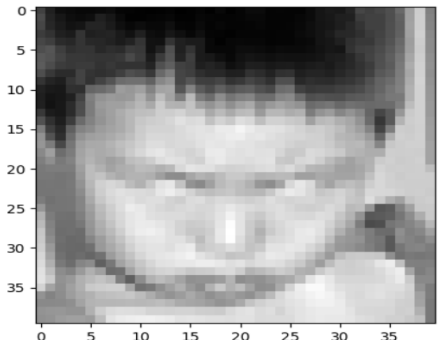
In general, once we've collected the BGR values, we would reshape the feature set (themselves) to grayscale (2D) if the images don't really need to be in full colour. Example:

```
# Before reducing to grayscale:
(48, 48, 3)
# After reducing to grayscale:
(48, 48, 1)
```

Next, we resize all the images in order to eliminate size variation in different images which also minimizes the length of the feature set. This reduces the load that will be given to the neural network. It might as well enhance the overall performance if, the magnitude for resizing is chosen correctly. That 'correctness' can be adjusted by examining how recognizable the image remains after resizing. The more recognizable yet squished the image is, the easier it gets for the neural network to perform with good results. Resizing an example image:




```
(48, 48)
```




```
(40, 40)
```

This resizing isn't that significant, though it will normalize all images to `(40, 40)` and eliminate size variation. Also, the image remains very recognizable.

We also take note of the desired output or $\boldsymbol{d}$ (as denoted in the previous chapter). In general, while extracting features for a model serving the purpose of classification, we come across *"desired output classes"* which are the literal justifications (like, "happy" or "sad"). The same is stored in a list, from which we get can get another list of their indexes. All in all, the model must end up pointing to the index for the literal justification, and therefore that index number acts as the value for our desired output.

Now, putting it all together and storing the features and the desired outputs:

```
### installation for pre-requisite modules ###
#                                            #
#  pip install numpy==1.18.5                 #
#  pip install opencv-python==4.1.0.25       #
#                                            #
##############################################
```

```
import os                    # -> available in standard lib
from random import shuffle   # -> available in standard lib
import numpy as np
import cv2

# Sub-directories which, in this case, also act
# as our literal justifications.
desired_output_classes = ['angry', 'disgust', 'fear', 'happy', 'neutral', 'sad
', 'surprise']
# This will be the temporary features and desired outputs
# 2D list, calling it "training_data".
training_data = []
# These will be the final features and desired outputs.
FEATURES = []
D = []

# This function will be used later on
def extract_features(image, subdir):
    try:
        # Adding "cv2.IMREAD_GRAYSCALE" flag to reduce the image to grayscale:
        feature_list = cv2.imread(image, cv2.IMREAD_GRAYSCALE)
        # Resizing to 40x40:
        feature_list = cv2.resize(feature_list, (40, 40))
        # To training_data, a list consisting the features and the
        # desired output (which is, the index of the subdir/literal
        # justification) is appended:
        training_data.append([feature_list, desired_output_classes.index(subdi
r)])
    except Exception as e:
        print(f'@{image}: {e}')


# Going through each sub-directory and
# extracting features from found images.
for subdir in desired_output_classes:
    for image in os.listdir('train/'+subdir):
        extract_features(f'train/{subdir}/{image}', subdir)

# IMPORTANT: shuffling the training data is crucial for
# better performance of the neural network. Because, it allows the
# neural network to learn from a varying class of data rather than
# a continuous stream of the same class of data.
shuffle(training_data)

# Appending the shuffled and final features to the FEATURES list
# and desired outputs to the D list.
for features, desired_output in training_data:
    FEATURES.append(features)
    D.append(desired_output)
```

```
# Final reshaping:
# -1 indicates that we are using all
# the entries in the FEATURES array.
# 40, 40, 1 indicates the image size
# (1 because the image is in grayscale).
FEATURES = np.array(FEATURES).reshape(-1, 40, 40, 1)

# Saving the FEATURES and D lists:
# This should create a FEATURES.npy and a
# D.npy in your current working directory.
np.save('features',FEATURES)
np.save('desired_outputs', D)
```

// STEP 3: *scripting and training the neural network*

But first, a few pre-requisites that need to be covered:

>> Loss Functions (Cost):

A loss function tells you about the performance of your network as a whole. In the previous chapter, I used the term "cost function" for an example loss function. Their purpose is exactly the same. On a given prediction, the loss function analyses its correctness, or, rubbishness. Commonly used loss functions:

Binary Cross-Entropy / Logistic Loss:

While working with neural networks, you'll come across a couple of distributions. 1st, which contains the predictions/scores shared by the neural network on given feature sets ($\boldsymbol{p}$). And 2nd, which contains the true values/answers relating to those feature sets ($\boldsymbol{q}$). Particularly for this loss function, $\boldsymbol{q}$ should have binary elements (either 0 or 1). *Entropy* between 2 values tell how disordered/dissimilar they are with each other. Therefore, high entropy between values in $\boldsymbol{p}$ and $\boldsymbol{q}$ means the network is performing badly, and vice-versa.

Before implementing a loss function, you are given the option to scale/model the prediction distribution ($\boldsymbol{p}$). This is achieved by using the *logit function*:

$$\boldsymbol{logit}(\boldsymbol{p_i}) = \mathbf{log}(\frac{\boldsymbol{p_i}}{\mathbf{1} - \boldsymbol{p_i}})$$
$$for\ p_i\ \in\ (0,1)$$

Therefore, $\boldsymbol{p_i}$ (an element in $\boldsymbol{p}$) is scaled between -∞ and ∞. The scaled $\boldsymbol{p}$ should now contain all the *logits* (values given by the logit function) for each element in the original $\boldsymbol{p}$. This isn't compulsory, you can use the original value of $\boldsymbol{p_i}$ which ranges between 0 and 1, and therefore the original $\boldsymbol{p}$. But it's recommended to use logit.

Note: do not confuse the "logit" function with the "logistic" function. The logistic function is the inverse of the logit function.

Finally, we can move on to the formula for Binary Cross-Entropy:

$$-\frac{\mathbf{1}}{\boldsymbol{N}}\sum_{\boldsymbol{i=1}}^{\boldsymbol{N}} \boldsymbol{q_i}\mathbf{log}(\boldsymbol{p_i}) + (\mathbf{1} - \boldsymbol{q_i})\mathbf{log}(\mathbf{1} - \boldsymbol{p_i})$$

Where, $\boldsymbol{N}$ is the number of samples in both the distributions. The equation can be broken down into 2 parts. The first one being negative times the sum over all logistic cross-entropies (entropy between each value in $\boldsymbol{p}$ and $\boldsymbol{q}$). And, the second one being the averaging ($\frac{\mathbf{1}}{\boldsymbol{N}}$).

Categorical Cross-Entropy:

This time, values of $\boldsymbol{q}$ (distribution of true values/answers) are represented in *one-hot* encoding. A list is called to be one-hot encoded when all the elements are represented as either a high or low bit (1 or 0) with a constraint to contain only a single high bit, example:

```
[0, 0, 1, 0] or [0, 1, 0]
```

Note: *One-cold* encoding follows the same format, but with a constraint to contain only a single low bit.

Here's an example which will show how $\boldsymbol{p}$ and $\boldsymbol{q}$ are formatted for this loss function:

```
# One-hot encoded desired outputs (true values) for
# N (samples/feature sets evaluated) = 3:
q = [[1, 0, 0], [0, 0, 1], [0, 1, 0]]

# Predictions of the neural network, with
# number of samples/feature sets evaluated (N) = 3,
# where each sample contains the probability of
# itself corresponding to either of the desired
# outputs (true values). Number of probabilities
# in each sample can be called K, which will simply be
# equal to the number of desired outputs (in the is
# case, K is 3):
p = [[0.1, 0.8, 0.1], [0.05, 0.05, 0.9], [0.9, 0.1, 0]]
```

Now, using the prediction from the neural network ($\boldsymbol{p}$) and the one-hot encoded true values ($\boldsymbol{q}$), the Categorical Cross-Entropy function forms up as:

$$-\frac{1}{N}\sum_{i=1}^{N}\sum_{j=1}^{K} q_{ij}\log(p_{ij})$$

Where, $\boldsymbol{N}$ is the number of samples in both the distributions, and $\boldsymbol{K}$ is the number of probabilities in each sample from $\boldsymbol{p}$. Remember: high values of the loss reflect on high entropy, and vice-versa. This formula can be broken down in the same way as the formula of the previous loss function. 1st part should be negative times the sum of all entropies and 2nd part should be the averaging.

Sparse Categorical Cross-Entropy:

This loss function expects $\boldsymbol{q}$ to contain any integer as an element, unlike the previous loss functions. In fact, it's just a variation of Categorical Cross-Entropy, but without the one-hot constraint on $\boldsymbol{q}$. This is useful when we have a large enough number of true values.

>> Optimizers:

An optimizer (sometimes called *updater*) is responsible for improving the

performance of a neural network by reducing the value of a cost/loss function (we've seen about this goal in detail in the previous chapter). It is the heart of the "learning part" of a neural network. More specifically, it changes the weights on the basis of a minimizing function. Examples:

Stochastic Gradient Descent (SGD):

$$\boldsymbol{SGD}(\boldsymbol{w_i}) = \boldsymbol{w_i} - \Upsilon\, \boldsymbol{g_i}$$

Where, $\boldsymbol{w_i}$ is the $\boldsymbol{i^{th}}$ weight in the distribution of weights taken as an example, Υ is the *learning rate* (step size) and $\boldsymbol{g_i}$ is the gradient descent (recall from the last chapter) for the $\boldsymbol{i^{th}}$ instance of the loss function used.

Finally:

$$\boldsymbol{w_i'} = \boldsymbol{SGD}(\boldsymbol{w_i})$$

Where $\boldsymbol{w_i'}$ is the optimized weight with respect to $\boldsymbol{w_i}$.

Adaptive Gradient Algorithm (AdaGrad):

AdaGrad is an "experience" based optimizer which is rather more optimal than SGD. Here's the formula:

$$\boldsymbol{AdaGrad}(\boldsymbol{w}_{t+1,i}) = \boldsymbol{w}_{t,i} - \frac{\Upsilon}{\sqrt{\sum_{\tau=1}^{t} \boldsymbol{g}_{t,i}^2}}\, \boldsymbol{g}_{t,i}$$

Where, $\boldsymbol{w_i}$ is the $\boldsymbol{i^{th}}$ weight in the distribution of weights taken as an example, Υ is the learning rate, $\boldsymbol{t}$ is the time step (encounter instance) and $\boldsymbol{g_{t,i}}$ is the gradient descent for the $\boldsymbol{i^{th}}$ instance of the loss function used at time stamp $\boldsymbol{t}$. The learning rate differs for each weight according to its progression. Respective weights are changed/updated on the basis of the same function.

Adam:

The Adam optimizer works piece-by-piece on all its components before

summarizing an output, and it's the most optimal of all. It doesn't require remembering (unlike AdaGrad) and therefore, it uses a little portion of the memory. It can handle occurrences inheriting large data/parameters efficiently. Piece-by-piece working of Adam (this procedure is also called the *update rule*):

1. Compute gradient $\boldsymbol{g_t}$ for the current time ($\boldsymbol{t}$).
2. Get Exponential decay rates for the *moment** estimates:
$$\boldsymbol{\beta_1, \beta_2} \in [0,1)$$
3. Update biased first moment estimate:
$$\boldsymbol{m_t} = \beta_1 m_{t-1} + (1-\beta_1) g_t$$
4. Update biased Second raw moment estimate:
$$\boldsymbol{v_t} = \beta_2 v_{t-1} + (1-\beta_2) g_t^2$$
5. Compute bias-corrected first moment estimate: $\widehat{\boldsymbol{m_t}} = \frac{m_t}{1-\beta_1^t}$
6. Compute bias-corrected second raw moment estimate: $\widehat{\boldsymbol{v_t}} = \frac{v_t}{1-\beta_2^t}$
7. Update parameters:

$$\boldsymbol{w_t} = \boldsymbol{w_{t-1}} - \boldsymbol{\alpha} \frac{\widehat{\boldsymbol{m_t}}}{\sqrt{\widehat{\boldsymbol{v_t}}} + \boldsymbol{\epsilon}}$$

Where, epsilon ($\boldsymbol{\epsilon}$) is a small constant responsible for numeric stability (defaults to $1e-7$) and alpha ($\boldsymbol{\alpha}$) is the step size. *Moment is a quantity that determines the shape of a function. The first moment figures the function's mean and the second moment figures the function's variance.

Now, we can finally move onto scripting the neural network! Here's the rundown:

```
### installation for pre-requisite modules ###
#                                            #
#  pip install numpy==1.18.5                 #
#  tensorflow ~ follow previously mentioned  #
#  installation.                             #
#                                            #
##############################################
```

```
import tensorflow as tf
from tensorflow.keras import layers, models
from numpy import load
# Loading the features and desired_outputs:
FEATURES = load('features.npy')
DESIRED_OUTPUTS = load('desired_outputs.npy')

FEATURES = FEATURES/255.0  # Normalizing the features list

# Creating a placeholder for all the layers,
# this will constitute as a model. Pattern
# for addition of layers will be Sequential
# (stacked one after another).
model = models.Sequential([
        # number of neurons = 32; kernel length = 3x3;
        layers.Conv2D(32, (3,3), activation='relu', input_shape=(40, 40, 1)),
        # stride length = 2x2;
        layers.MaxPooling2D((2, 2)),
        # Increased the number of neurons to 64.
        layers.Conv2D(64, (3, 3), activation='relu'),
        layers.MaxPooling2D((2, 2)),
        layers.Conv2D(64, (3, 3), activation='relu'),

        # Completing the model with a dense layer to generate
        # classifications. The "Flatten" layer is used to
        # transform the multi-dimensional data used and
        # produced by previous layers into 1D (remember,
        # Dense layers work with 1D data or feature set).
        layers.Flatten(),
        layers.Dense(64, activation='relu'),
        # Output layer: consits 7 outputs (or, 7 classifications
        # for the 7 emotions we extracted the training data for).
        layers.Dense(7, activation='relu'),
])
model.summary()

model.compile(optimizer='adam',
              # Setting from_logits to true in order to apply the logit
              # function on the network's predictions.

            loss=tf.keras.losses.SparseCategoricalCrossentropy(from_logits=1),
             metrics=['accuracy'])

model.fit(FEATURES, DESIRED_OUTPUTS,
          # portion of the dataset to taken at a time = 64 entries.
          # This quantity can be changed depanding on how much data
          # you have and your machine's capabilities.
          batch_size=64,
```

```
            # iterations over the network (epochs) = 8. Recommended
            # to use more epochs!
            epochs=10,
            # Alloting 40% of the data for validation/checking purpose.
            # This quantity can be changed depanding on how much data you
            # have. Although, this value can directly affect your network's
            # overall performace. Keeping it too high may lead to poor
            # results as you're trimming down your training data.
            # Keeping it too low may also lead to poor results as the
            # validating entities settle to be fewer.
            validation_split=0.4)

# This command will save the model in your current working directory.
model.save('KMKY_Image_Classifier')
```

Model summary:

```
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
conv2d (Conv2D)              (None, 38, 38, 32)        320
_________________________________________________________________
max_pooling2d (MaxPooling2D) (None, 19, 19, 32)        0
_________________________________________________________________
conv2d_1 (Conv2D)            (None, 17, 17, 64)        18496
_________________________________________________________________
max_pooling2d_1 (MaxPooling2 (None, 8, 8, 64)          0
_________________________________________________________________
conv2d_2 (Conv2D)            (None, 6, 6, 64)          36928
_________________________________________________________________
flatten (Flatten)            (None, 2304)              0
_________________________________________________________________
dense (Dense)                (None, 64)                147520
_________________________________________________________________
dense_1 (Dense)              (None, 7)                 455
=================================================================
```

// STEP 4: *testing the network*

This is the final step! And in this very step you'll get to witness your network/model in action. But before launching off to the script and getting a prediction, I would like to help you understand and dodge some errors that a developer could make throughout the whole procedure of building a neural network. Firstly, know your machine and your dataset in order to properly input arguments to the `fit` function (mentioned in the previous

step). Each argument can varyingly affect your network's performance. More `epochs` result in a better overall accuracy of the network, but take more time. The rest of the arguments must be adequate in order to get the same accuracy in fewer `epochs`. Secondly, try to print all things printable to get a better grasp of what's going on. This will, later on, help you build your own projects with a breeze and it'll make bug fixing way faster and easier. And lastly, know the structure of the directories you're working with and the position of the script file that you're currently editing. This will help you work with resources better and errorfree.

Finally, here's the script and the outcomes:

```
### installation for pre-requisite modules ###
#                                            #
#  pip install numpy==1.18.5                 #
#  pip install opencv-python==4.1.0.25       #
#  tensorflow ~ follow previously mentioned  #
#  installation.                             #
#                                            #
##############################################

import tensorflow as tf
import numpy as np
import cv2

desired_output_classes = ['angry', 'disgust', 'fear', 'happy', 'neutral', 'sad
', 'surprise']

# Using the same feature extractor shown in step 2 in order to
# collect features of our test images
def extract_features(image):
    try:
        # Adding "cv2.IMREAD_GRAYSCALE" flag to reduce the image to grayscale:
        feature_list = cv2.imread(image, cv2.IMREAD_GRAYSCALE)
        # Resizing to 40x40:
        feature_list = cv2.resize(feature_list, (40, 40))
        # Final reshaping (remember from step 2):
        return feature_list.reshape(-1, 40, 40, 1)
    except Exception as e:
        print(e)
        exit(1)

# Example test image and its features (we know it to fall under
# the "happy" class):
path_to_test_image = 'test/happy/PrivateTest_218533.jpg'
```

```
test_image_features = extract_features(path_to_test_image)

# Loading in the model:
model = tf.keras.models.load_model('KMKY_Image_Classifier')
# Getting a prediction on our test image. This would output the
# score of each class at being the true value/classification
# for the given test image. Index of the score having the highest
# value will correspond to the index of the literal classification
# as mentioned in a list above (desired_output_classes).
prediction = model.predict(test_image_features)
print(prediction)
# ^prints: [[ 78.833336  0.  0.  303.71823  0.  0.  55.082634 ]] (comes
# through the output layer that we defined while scripting the network)

associated_index = np.where(prediction == np.amax(prediction))[1][0]
# ^ = 3 (verify from the list printed above, value at index 3 is
# the highest value)
literal_classification = desired_output_classes[associated_index]
print(literal_classification)
# ^prints: happy
```

// Bibliography

Sham Kakade. "Adaptive Gradient Methods, Adam/AdaGrad" (Machine Learning for Big Data, CSE547/STAT548, University of Washington) Source:*https://courses.cs.washington.edu/courses/cse547/17sp/slides/adaptive_gradient_methods.pdf*

Diederik P. Kingma, Jimmy Lei Ba. "Adam: A Method for Stochastic Optimization" Source: *https://arxiv.org/pdf/1412.6980.pdf*

Hey, Good Lookin'

what you got cooking?

- Hank Williams (1964)

We're now going live! Instead of inputting features extracted from saved images to the model we built in the previous chapter, we'll be inputting features extracted from *frames* captured through a live camera feed. Each frame shares a certain array of RGB (or BGR) values, which is our targeted feature set. In order to capture the same, we'll use OpenCV with python. Here's the rundown:

```
### installation for pre-requisite modules ###
#                                            #
#  pip install numpy==1.18.5                 #
#  pip install opencv-python==4.1.0.25       #
#  tensorflow ~ follow previously mentioned  #
#  installation.                             #
#                                            #
##############################################

from time import ctime  # -> available in standard lib
import tensorflow as tf
import numpy as np
import cv2

desired_output_classes = ['angry', 'disgust', 'fear', 'happy', 'neutral', 'sad
', 'surprise']

# Loading in the model:
model = tf.keras.models.load_model('KMKY_Image_Classifier')

# Initializing a video capture, 0 => webcam:
cap = cv2.VideoCapture(0)
# While setting the frame height and width, make sure
# that there isn't a black padding on the top and
# bottom of the frame because they'll fill your feature
# set with lots of zeros at the head and the tail. If
# you find these black paddings, shrink your frame.
# Also, keep the dimensions adequate for the resizing.
cap.set(cv2.CAP_PROP_FRAME_HEIGHT, 80)
cap.set(cv2.CAP_PROP_FRAME_WIDTH, 80)

# This function will be used later on
```

```
def get_prediction(feature_set):
    # Remember all these commands from the previous chapter (STEP 4):
    prediction = model.predict(feature_set)
    # print(prediction)
    associated_index = np.where(prediction == np.amax(prediction))[1][0]
    literal_classification = desired_output_classes[associated_index]
    print(literal_classification)

while True:
    # Reading from the capture:
    # (ret is a bool, True if teh function can read successfully
    # frame is a list, containing pixel data)
    ret, frame = cap.read()
    # Setting up an output window named "* live":
    cv2.imshow('* live', frame)

    # Converting the current pixel data to gray scale (gray scale
    # reduction):
    feature_set = cv2.cvtColor(frame, cv2.COLOR_BGR2GRAY)
    # Remember these commands from the "extract_features" function
    # used in the previous chapter (STEP 2):
    feature_set = cv2.resize(feature_set, (40, 40))
    feature_set = feature_set.reshape(-1, 40, 40, 1)
    print(f"{'-'*10} PREDICTION FOR FRAME EVALUATED AT {ctime()} {'-'*10}")
    get_prediction(feature_set)

    # The waitKey function returns the key code of a key when
    # pressed (255 if none of the keys are pressed). "&" is
    # the bitwise AND operator, pairing it with 0xff (8 high bits)
    # ensures that the key code is limited to a single byte.
    KC = cv2.waitKey(1) & 0xff
    if KC == 0x71:   # 0x71 or 113 is the key code for 'q'.
        # The loop breaks if q is pressed.
        break

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

And that's it! Moving forward with the same, you can improve the current script or add a new functionality, like sockets, in order to predict on a stream shared by a foreign device. You can also make it so that instead of a live feed, the script is compatible with recorded videos. But that's an exercise for you!