Data Science and Generative AI

by SATISH @

https://sathyatech.com

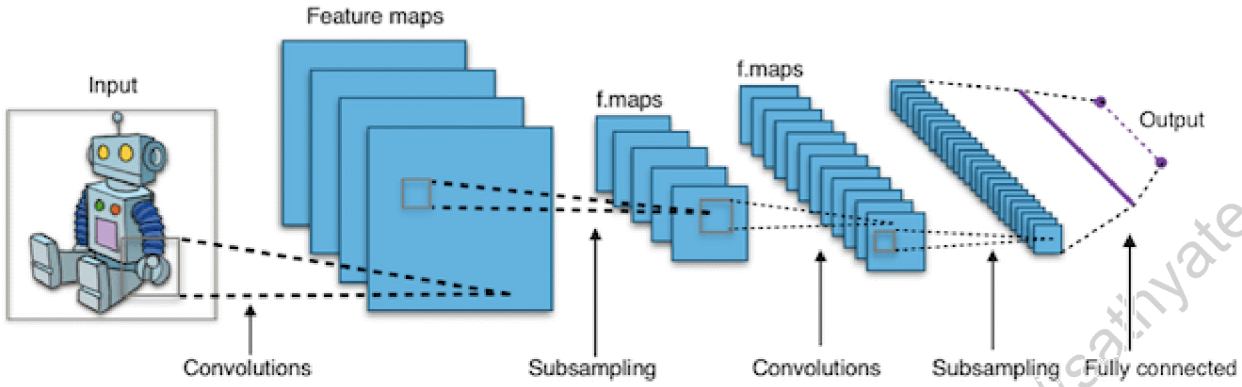
Deep Learning (DL)

KERAS

Keras is our recommended library for deep learning in Python, especially for beginners. Its minimalistic, modular approach makes it a breeze to get deep neural networks up and running. You can read more about it here:

WTF are Convolutional Neural Networks?

In a nutshell, Convolutional Neural Networks (CNN's) are multi-layer neural networks (sometimes up to 17 or more layers) that assume the input data to be images.



By making this requirement, CNN's can drastically reduce the number of parameters that need to be tuned. Therefore, CNN's can efficiently handle the high dimensionality of raw images.

Step 1: Set up your environment for Anaconda

First, make sure you have the following installed on your computer:

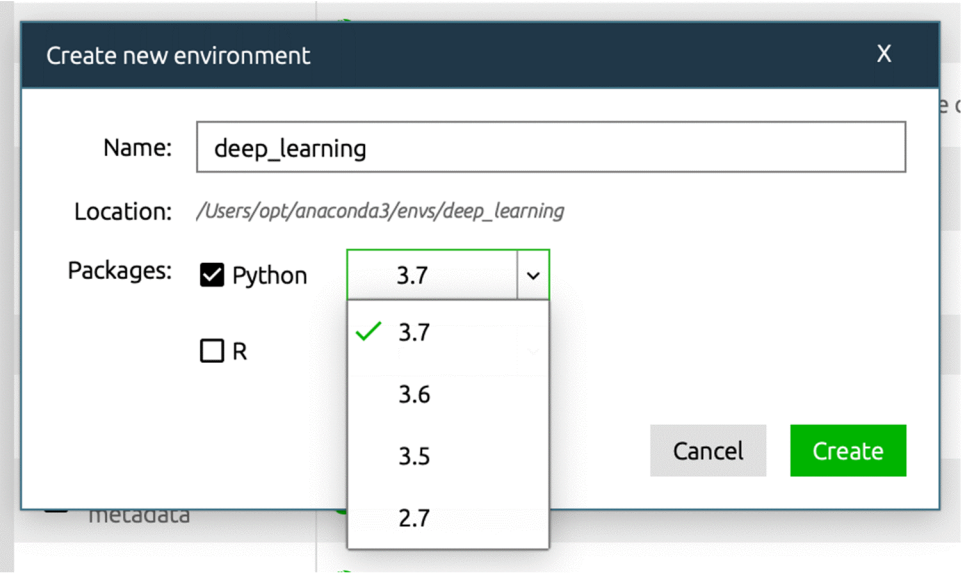
- Python 3+
- SciPy with NumPy
- Matplotlib (Optional, recommended for exploratory analysis)

We strongly recommend installing Python, NumPy, SciPy, and matplotlib through the Anaconda Distribution. It comes with all of those packages. Plus, it makes the next step much easier.

Once you've installed Anaconda, you can open up the **Anaconda Navigator** application that came with it. From here, you can manage your environments.

As a best practice, the first thing we recommend is creating a new environment specifically for deep learning with Keras and Tensorflow. This helps keep your packages and dependencies organized, without spilling into other things you're working on.

To do so, click on the **Environments** tab on the left and then click **Create** at the bottom of the list. Select **Python 3.7** and give it a name:



You can check to see if you've installed everything correctly:

Go to your command line program (Terminal on a Mac) and type in:

```
1$ conda activate deep_learning
```

This will switch over to the new environment you just installed. Then, type in:

```
1$ python
```

You'll see the Python interpreter:

```
1Python 3.7.13 (default, Mar 28 2022, 07:24:34)
2[Clang 12.0.0 ] :: Anaconda, Inc. on darwin
```

Next, you can import your libraries and print their versions:

```
1>>> import numpy
2>>> import matplotlib
3>>>print( numpy.__version__ )
41.21.5
5>>>print( matplotlib.__version__ )
63.5.1
```

```
7>>>quit()
```

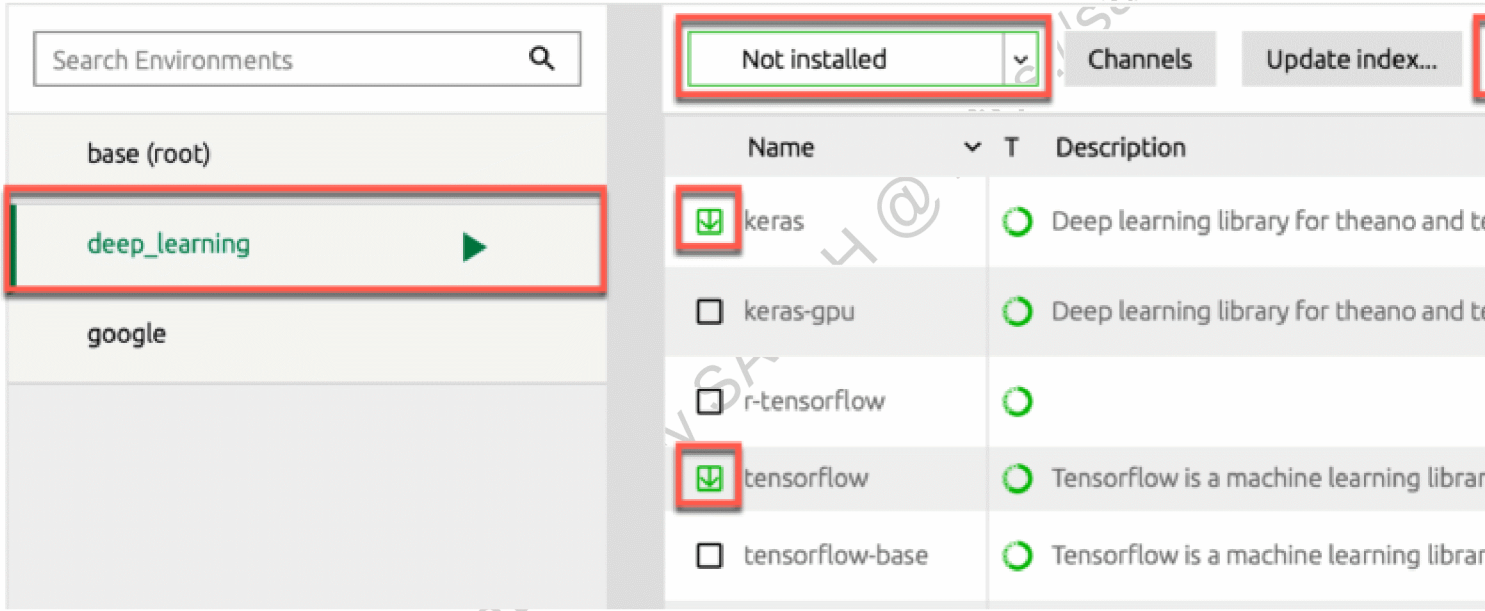
Note: If either or both of these libraries were not found, don't worry (sometimes Anaconda changes which libraries come installed by default). You can install them along with the next two libraries in Step 2:

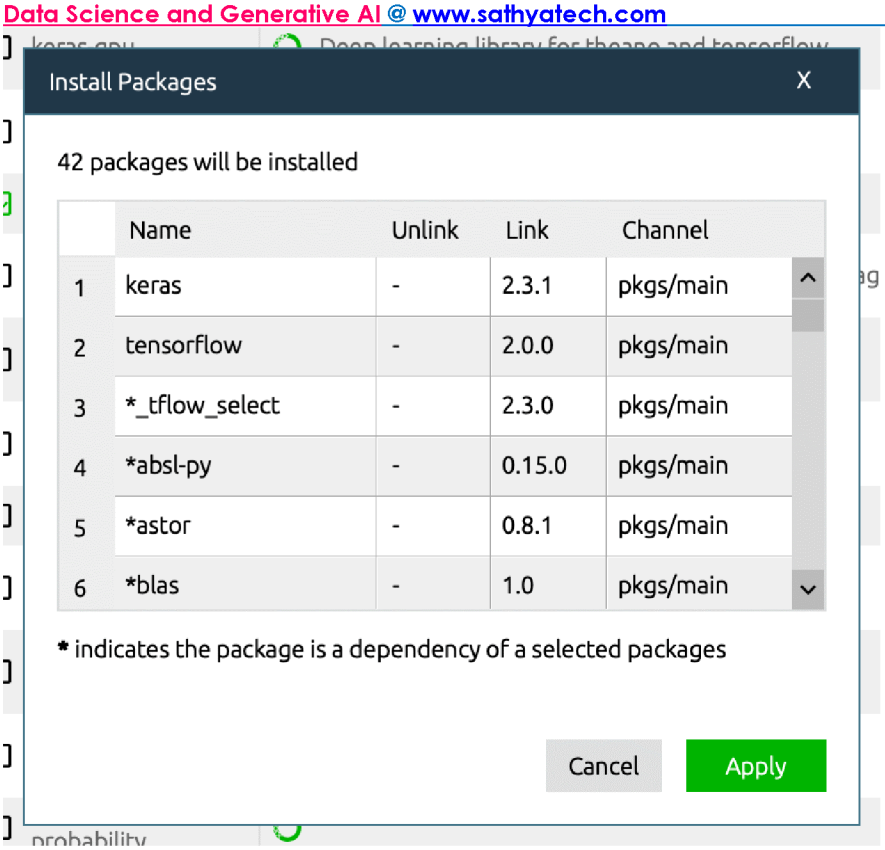
Step 2: Install Keras and Tensorflow for Anaconda

It wouldn't be a Keras tutorial if we didn't cover how to install Keras (and TensorFlow). TensorFlow is a free and open source machine learning library originally developed by Google Brain. These two libraries go hand in hand to make Python deep learning a breeze.

The good news is that if you used Anaconda, then you can install them directly through Anaconda Navigator.

1. Simply navigate to the **Environments** tab.
2. Select **Not Installed** from the package manager.
3. Search for "tensorflow" in the search bar.
4. Then check the following packages: *keras* and *tensorflow*.
5. Click **Apply** at the bottom, then **Apply** in the pop-up prompt.





Note: if you run into issues at this step, make sure your Anaconda installation is up-to-date. You can update it from the command line using the command conda update anaconda .

You can confirm it's installed correctly, you can run this in the command line:

```
1$ python -c "import keras; print( keras.__version__ )"

2Using TensorFlow backend.

32.3.1
```

Step 3: Import libraries and modules.

Let's start by importing numpy and setting a seed for the computer's pseudorandom number generator. This allows us to reproduce the results from our script:

Python

```
1import numpy as np

2np.random.seed(123)  # for reproducibility
```

Next, we'll import the Sequential model type from Keras. This is simply a linear stack of neural network layers, and it's perfect for the type of feed-forward CNN we're building in this tutorial.

Python

```
1from keras.models import Sequential
```

Next, let's import the "core" layers from Keras. These are the layers that are used in almost any neural network:

Python

```
1from keras.layers import Dense, Dropout, Activation, Flatten
```

Then, we'll import the CNN layers from Keras. These are the convolutional layers that will help us efficiently train on image data:

Python

```
1from keras.layers import Convolution2D, MaxPooling2D
```

Finally, we'll import some utilities. This will help us transform our data later:

Python

```
1from keras.utils import np_utils
```

Now we have everything we need to build our neural network architecture.

Step 4: Load image data from MNIST.

MNIST is a great dataset for getting started with deep learning and computer vision. It's a big enough challenge to warrant neural networks, but it's manageable on a single computer. That makes it perfect for this Keras tutorial.

The Keras library conveniently includes it already. We can load it like so:

Python

```
1from keras.datasets import mnist
2
3# Load pre-shuffled MNIST data into train and test sets
4(X_train, y_train), (X_test, y_test) = mnist.load_data()
```

We can look at the shape of the dataset:

Python

```
print( X_train.shape )

# (60000, 28, 28)
```

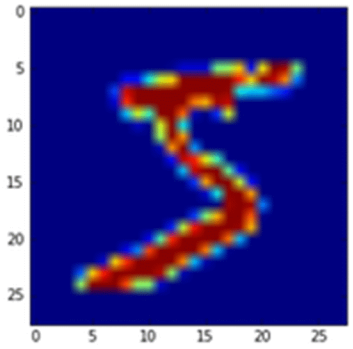
Great, so it appears that we have 60,000 samples in our training set, and the images are 28 pixels x 28 pixels each. We can confirm this by plotting the first sample in matplotlib:

Python

```
from matplotlib import pyplot as plt

plt.imshow(X_train[0])
```

And here's the image output:



In general, when working with computer vision, it's helpful to visually plot the data before doing any algorithm work. It's a quick sanity check that can prevent easily avoidable mistakes (such as misinterpreting the data dimensions).

Step 5: Preprocess input data for Keras.

When using the TensorFlow backend, you must explicitly declare a dimension for the number of *channels* in the input images. For example, a full-color image with all 3 **RGB channels** will have a channel value of 3.

Our MNIST images only have 1 channel, but we must explicitly declare that.

In other words, we want to transform our dataset from having shape (n, width, height) to (n, width, height, channels).

Here's how we can do that easily:

Python

```
X_train = X_train.reshape(X_train.shape[0], 28, 28, 1)

X_test = X_test.reshape(X_test.shape[0], 28, 28, 1)
```

To confirm, we can print X_train's dimensions again:

Python

```
1print( X_train.shape )

2# (60000, 28, 28, 1)
```

The final preprocessing step for the input data is to convert our data type to **float32** and normalize our data values to the range [0, 1].

Python

```
1X_train = X_train.astype('float32')

2X_test = X_test.astype('float32')

3X_train /= 255

4X_test /= 255
```

Now, our input data are ready for model training.

Step 6: Preprocess class labels for Keras.

Next, let's take a look at the shape of our class label data:

Python

```
1print( y_train.shape )

2# (60000,)
```

Hmm… that may be problematic. We should have 10 different classes, one for each digit, but it looks like we only have a 1-dimensional array. Let's take a look at the labels for the first 10 training samples:

Python

```
1print( y_train[:10] )

2# [5 0 4 1 9 2 1 3 1 4]
```

And there's the problem. The y_train and y_test data are not split into 10 distinct class labels, but rather are represented as a single array with the class values.

We can fix this easily:

Python

```
1# Convert 1-dimensional class arrays to 10-dimensional class matrices

2Y_train = np_utils.to_categorical(y_train, 10)

3Y_test = np_utils.to_categorical(y_test, 10)
```

Now we can take another look:

Python

```
1print( Y_train.shape )

2# (60000, 10)
```

There we go… much better!

Step 7: Define model architecture.

Now we're ready to define our model architecture. In actual R&D work, researchers will spend a considerable amount of time studying model architectures.

To keep this tutorial moving along, we're not going to discuss the theory or math here. This alone is a rich and meaty field, and we recommend the CS231n class mentioned earlier for those who want to learn more.

Plus, when you're just starting out, you can just replicate proven architectures from academic papers or use existing examples.

Let's start by declaring a sequential model format:

Python

```
1model = Sequential()
```

Next, we declare the input layer:

Python

```
1model.add(Convolution2D(32, (3,3), activation='relu', input_shape=(28,28,1)))
```

The input shape parameter should be the shape of 1 sample. In this case, it's the same (28, 28, 1) that corresponds to the (width, height, channels) of each digit image.

But what do the first two parameters represent? They correspond to the number of convolution filters to use (32) and the number of rows and columns (3, 3) in each convolution kernel.

**Note: The step size is (1,1) by default, and it can be tuned using the 'strides' parameter.*

We can confirm this by printing the shape of the current model output:

Python

```
1print( model.output_shape )

2# (None, 26, 26, 32)
```

That output corresponds to (samples, new_rows, new_cols, filters). In other words, the current model will output all of the samples, convoluted into a 26×26 array using 32 filters.

Next, we can simply add more layers to our model like we're building legos:

Python

```
1model.add(Convolution2D(32, (3,3), activation='relu'))

2model.add(MaxPooling2D(pool_size=(2,2)))

3model.add(Dropout(0.25))
```

Again, we won't go into the theory too much, but it's important to highlight the **Dropout** layer we just added. This is a method for regularizing our model in order to prevent overfitting.

MaxPooling2D is a way to reduce the number of parameters in our model by sliding a 2×2 pooling filter across the previous layer and taking the max of the 4 values in the 2×2 filter.

So far, for model parameters, we've added two Convolution layers. To complete our model architecture, let's add a fully connected layer and then the output layer:

Python

```
1model.add(Flatten())

2model.add(Dense(128, activation='relu'))

3model.add(Dropout(0.5))

4model.add(Dense(10, activation='softmax'))
```

For Dense layers, the first parameter is the output size of the layer. Keras automatically handles the connections between layers.

Note that the final layer has an output size of 10, corresponding to the 10 classes of digits.

Also note that the weights from the Convolution layers must be flattened (made 1-dimensional) before passing them to the fully connected Dense layer.

Here's how the entire model architecture looks together:

Python

```
model = Sequential()

model.add(Convolution2D(32, (3,3), activation='relu', input_shape=(28,28,1)))
model.add(Convolution2D(32, (3,3), activation='relu'))
model.add(MaxPooling2D(pool_size=(2,2)))
model.add(Dropout(0.25))

model.add(Flatten())
model.add(Dense(128, activation='relu'))
model.add(Dropout(0.5))
model.add(Dense(10, activation='softmax'))
```

Now all we need to do is define the loss function and the optimizer, and then we'll be ready to train it.

Step 8: Compile model.

Now we're in the home stretch! The hard part of the Keras tutorial is already over.

We just need to compile the model and we'll be ready to train it. When we compile the model, we declare the loss function and the optimizer (SGD, Adam, etc.).

Python

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

Step 9: Fit model on training data.

To fit the model, all we have to do is declare the batch size and number of epochs to train for, then pass in our training data.

Python

```
model.fit(X_train, Y_train,
      batch_size=32, epochs=10, verbose=1)
# Epoch 1/10
# 7744/60000 [==>.....] - ETA: 96s - loss: 0.5806 - acc: 0.8164
```

Easy, huh?

This might take a few minutes, but you'll be able to track the progress as it goes.

Step 10: Evaluate model on test data.

Finally, we can evaluate our model on the test data:

Python

```
score = model.evaluate(X_test, Y_test, verbose=0)
```

Congratulations… you've made it to the end of this Keras tutorial!

We've just completed a whirlwind tour of Keras's core functionality, but we've only really scratched the surface. Hopefully you've gained the foundation to further explore all that Keras has to offer.

The complete code, from start to finish.

Here's all the code in one place, in a single script.

Python

```
# 3. Import libraries and modules
import numpy as np
np.random.seed(123)  # for reproducibility

from keras.models import Sequential
from keras.layers import Dense, Dropout, Activation, Flatten
from keras.layers import Convolution2D, MaxPooling2D
from keras.utils import np_utils
from keras.datasets import mnist

# 4. Load pre-shuffled MNIST data into train and test sets
(X_train, y_train), (X_test, y_test) = mnist.load_data()

# 5. Preprocess input data
X_train = X_train.reshape(X_train.shape[0], 28, 28, 1)
X_test = X_test.reshape(X_test.shape[0], 28, 28, 1)
X_train = X_train.astype('float32')
X_test = X_test.astype('float32')
X_train /= 255
X_test /= 255

# 6. Preprocess class labels
Y_train = np_utils.to_categorical(y_train, 10)
```

```
Y_test = np_utils.to_categorical(y_test, 10)

# 7. Define model architecture
model = Sequential()

model.add(Convolution2D(32, (3,3), activation='relu', input_shape=(28,28,1)))
model.add(Convolution2D(32, (3,3), activation='relu'))
model.add(MaxPooling2D(pool_size=(2,2)))
model.add(Dropout(0.25))

model.add(Flatten())
model.add(Dense(128, activation='relu'))
model.add(Dropout(0.5))
model.add(Dense(10, activation='softmax'))

# 8. Compile model
model.compile(loss='categorical_crossentropy',
              optimizer='adam',
              metrics=['accuracy'])

# 9. Fit model on training data
model.fit(X_train, Y_train,
          batch_size=32, epochs=10, verbose=1)
```

```
47

48# 10. Evaluate model on test data

49score = model.evaluate(X_test, Y_test, verbose=0)
```

Problem Statement

Chest X-ray exam is one of the most frequent and cost-effective medical imaging examinations. However, clinical diagnosis of a chest X-ray can be challenging, and, sometimes, believed to be harder than diagnosis via chest CT imaging. To achieve clinically relevant computer-aided detection and diagnosis (CAD) in real-world medical sites on all data settings of chest X-rays is still very difficult unless several thousands of images are employed for study.

As part of a research study to explore deep learning techniques, National Institute of Health Clinical Center (NIH) has recently open-sourced its dataset of frontal chest X-ray images of patients. The task was to identify the class of thorax diseases from the given chest x-ray images.

Data Exploration

We were given two separate datasets, one containing images and the other containing CSV files. The train data has information for 18,577 patients and test data has information for 12,386 patients. The target variable has 14 types of thorax diseases. Some part of the data was anonymised to restrict fraudulent submissions.

Variable	Description
row_id	Unique patient id
age	Patient age
gender	Patient gender
view position	Position of Image (binary)
Image_name	X-ray image corresponding to patient

Variable	Description
detected	Target variable

Table 1. Description of data in the csv files.

#We import the required libraries for data exploration and visualization.

```
import pandas as pd
import matplotlib.pyplot as plt
import os
import numpy as np
import seaborn as sns
%matplotlib inline
```

The DL#2 consists of three CSV files (train, test and sample_submission) and two folders which consists of the train and the test images. The train and the test CSV files consist of the meta-data related to each patient's X-ray images stored in the train and test image folders, respectively. Table 2 shows the metadata stored in train.csv for the first five patients.

#Next we read the datasets.

```
train = pd.read_csv('train.csv')
test = pd.read_csv('test.csv')
train.head()
```

	row_id	age	gender	view_position	Image_name	detected
0	id_0	45	M	0	scan_0000.png	class_3
1	id_1	57	F	0	scan_0001.png	class_3
2	id_10	58	M	0	scan_00010.png	class_3
3	id_1000	64	M	0	scan_0001000.png	class_6
4	id_10000	33	M	1	scan_00010000.png	class_3

Table 2. Information of the first five patients stored in train.csv

The target variable or the detected variable (as seen in Table 2) has 14 unique classes, each denoting a class of thorax disease.

Exploratory Visualization

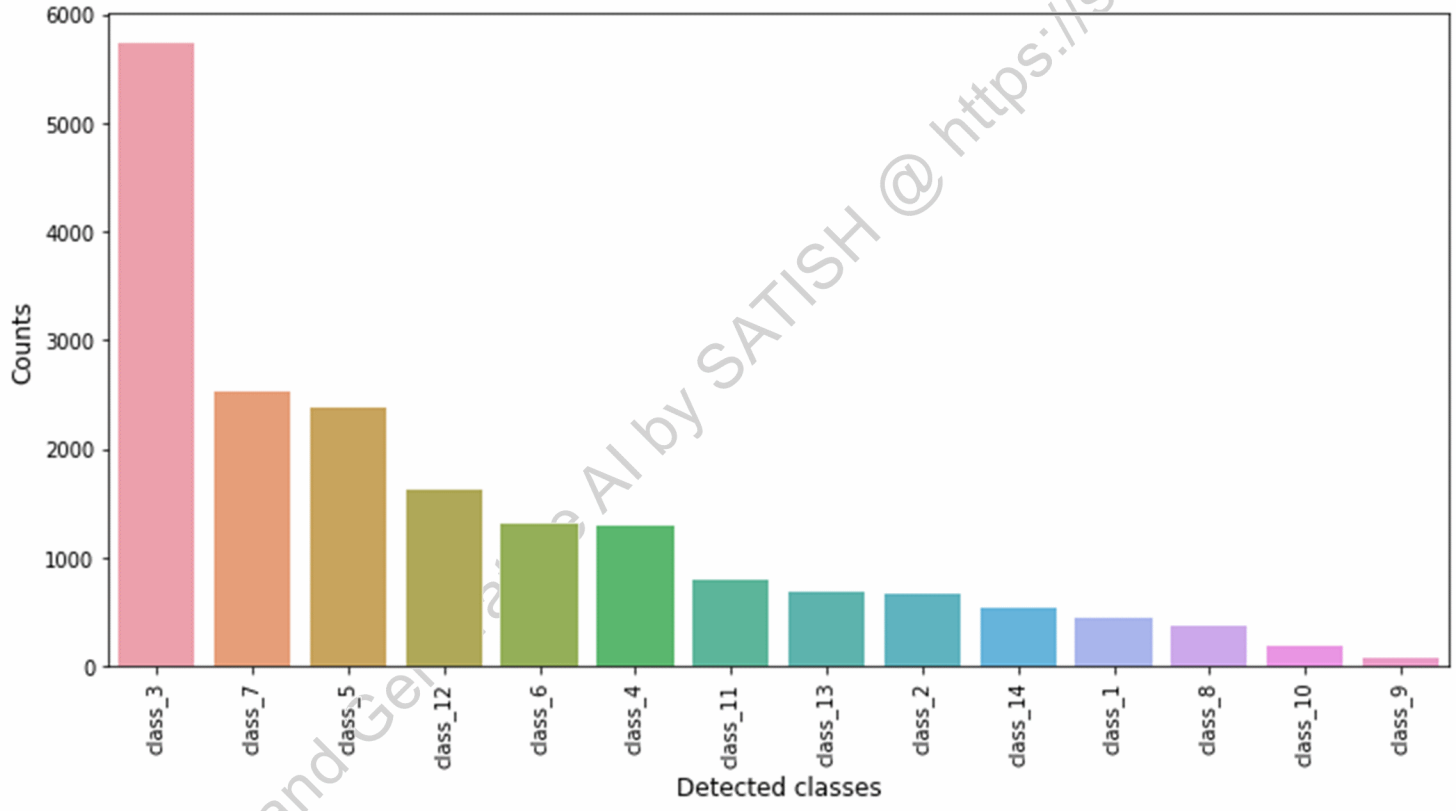


Fig 1. Frequency of each target class in the training dataset.

We now explore our dataset with the help of visual graphs and plots to get a better understanding of it. We use seaborn and matplotlib libraries to produce our visualisations.

Fig 1. shows the occurrence/ frequency of each class in the target variable of the training dataset, sorted in the descending order. From the figure we can see that class_3 type thorax disease is more prominent in patients, while class_9 type thorax disease is quite rare.

#Distribution of the target variable.

```
detected_counts = train.detected.value_counts()

plt.figure(figsize = (12,6))

sns.barplot(detected_counts.index, detected_counts.values, alpha = 0.9)

plt.xticks(rotation = 'vertical')

plt.xlabel('Detected classes', fontsize =12)

plt.ylabel('Counts', fontsize = 12)

plt.show()
```

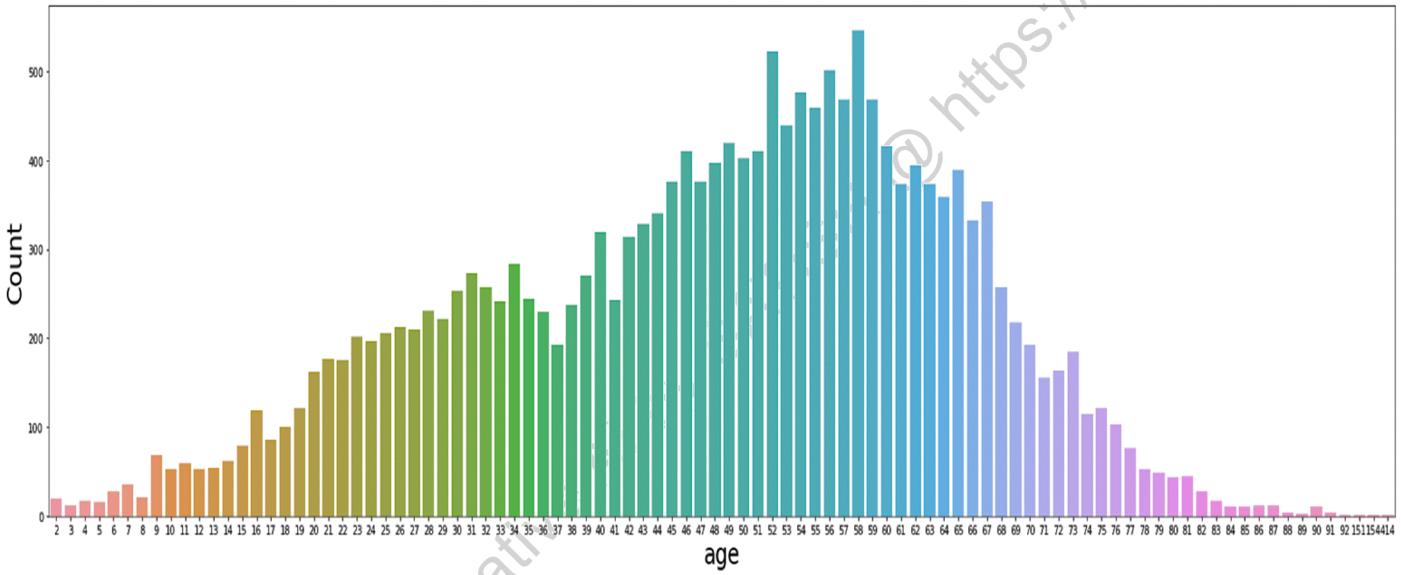


Fig 2. Countplot showing the distribution of the varaible 'age' in the training dataset.

Next, we analyse the distribution of the variable ‘age’ in the training dataset. From Fig 2. we see that the ‘age’ of the patients is normally distributed. Also, it contains outliers, e.g. some patients in the training dataset have age in the range of 150–500 years, which seems to be incorrect and may misguide our training model.

#Distribution of the variable 'age'.

```
ax = plt.figure(figsize=(30, 8))

sns.countplot(train.age)

axis_font = {'fontname':'Arial', 'size':'24'}

plt.xlabel('age',  **axis_font)

plt.ylabel('Count',  **axis_font)
```

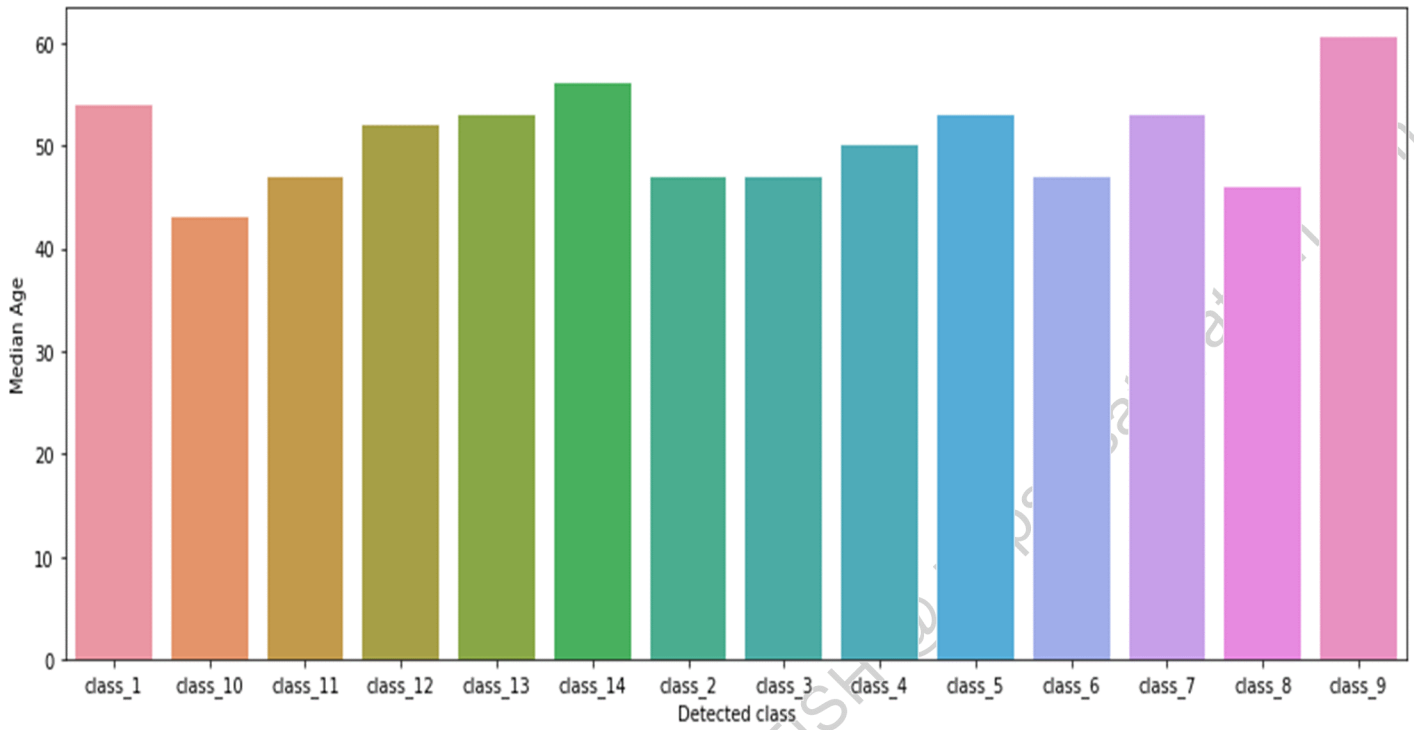


Fig 3. Median age for each class in the target variable

Fig. 3 shows the median age of patients for each type of class in the 'detected' (target) variable. We can see that class_9 type thorax disease is common in patients over the age of 60. We calculate the median of the variable 'age' for each target class, instead of the mean because of outliers in the data.

#Distribution of the variable 'age' over the target variable.

```
temp = train.groupby(['detected']).median()

ax = plt.figure(figsize=(15, 6))

sns.barplot(temp.index, temp.age)
```

```
plt.xlabel('Detected class')

plt.ylabel('Median Age')
```

Finally, we use the OpenCV(cv2) library for image manipulation. Other libraries such as PILLOW, PIL, or skimage can also be used. Unlike humans, a computer cannot recognise an image as it is. The computer cannot see shapes or colors. To a computer, any image is read as an array of numbers.

```
#Load the image data and visualise it USING Open CV.


TRAIN_PATH = 'train_/'

TEST_PATH = 'test_/'


import cv2

img = cv2.imread(TRAIN + 'scan_0000.png')

print(img.shape)

plt.imshow(img)
```

Each image in the training dataset is of the shape (1024, 1024, 3), where the first two numbers represent the number of rows of pixels and the number of columns of pixels. Number 3 represents the RGB color spectrum. Each number in the array represents a single pixel.

Data Preprocessing

Since each image in our dataset is an array of size 1024*1024*3, processing 18,577 images of size 1024*1024*3 requires enormous computation power. Therefore, we resize them to more appropriate sizes. Here, we resize each image in our dataset to 128*128*3 dimensions.

```
#We create a function, which reads an image, resizes it to 128 x128 dimensions and returns it.


def read_img(img_path):
```

```
img = cv2.imread(img_path)

img = cv2.resize(img, (128, 128))

    return img


from tqdm import tqdm

train_img = []

for img_path in tqdm(train['image_name'].values):

train_img.append(read_img(TRAIN_PATH + img_path))
```

Next, we rescale our image data. Rescale is a value by which we will multiply the data before any other preprocessing. Our original images are made of RGB coefficients lying in the range 0–255, but such values will be too high for our models to process (given a typical learning rate), so we target values between 0 and 1 instead by scaling with a 1/255 factor.

```
#Rescaling the images


x_train = np.array(train_img, np.float32) / 255.
```

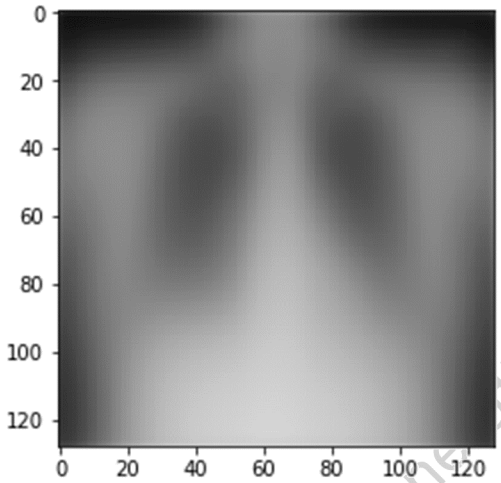


Fig 4. Mean of all the training images

Next, we calculate the mean of all the training images. It is a good average representation of the entire training dataset. The mean image is calculated by taking the mean values for each pixel across all training examples. The image roughly represents the thorax (see Fig 4.) . This image lets us conclude that all the thoraxes are somewhat aligned to the center and are of comparable sizes.

#Mean of the images

```
mean_img = np.mean(x_train, axis=0)

plt.imshow(mean_img)
```

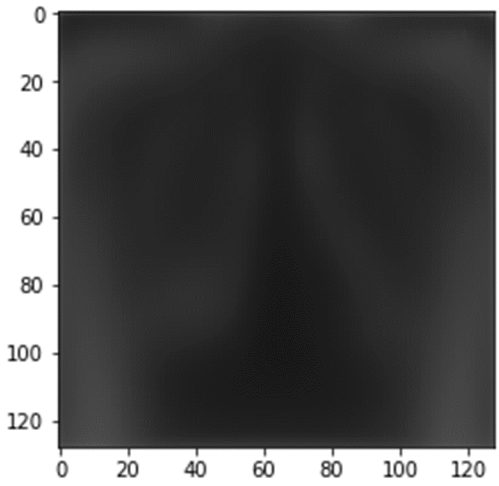


Fig 5. Standard deviation of all the trainig images.

Following this, we calculate the standard deviation of the all images. High variance shows up whiter (see Fig. 5), so we can see that the pictures vary a lot at the lungs compared to the rest of the image.

#Standard deviation of the images

```
std_img = np.std(x_train, axis=0)

plt.imshow(std_img)
```

The reason we calculate the mean and standard deviation of the images is that in the process of training our network, we are going to be multiplying (weights) and adding (biases) to these initial inputs to cause activations, which get back-propagated with the gradients to train the model. We will like each feature to have a similar range so that our gradients don't go out of control (and we only need one global learning rate multiplier).

Thus, we normalize both our train and test datasets. This is done using the following formula:

$X = {(x-μ)}/{σ}$

where,

$x = Input array$

$μ = Mean value$

$σ = Standard deviation$

$X = Output/Resultant array$

#Normalization

```
x_train_norm = (x_train - mean_img) / std_img

x_train_norm.shape
```

Finally, we encode our target variables as the model needs an array of numbers to train. But for categorical variables where no ordinal relationship exists, 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. Therefore, we apply one-hot encoding to the integer representation of the categories. Here, the integer encoded variable is removed and a new binary variable is added for each unique integer value, for example, a one-hot encoded representation of the target variable can be:

Class_1 : 10000000000000

Class_2 : 01000000000000

Class_3 : 00100000000000, and so on.

#Encoding the target variable as integers

```
class_list = train['detected'].tolist()

Y_train = {k:v+1 for v,k in enumerate(set(class_list))}

y_train = [Y_train[k] for k in class_list]
```

#One-hot encoding the target variable.

```
from keras.utils import to_categorical

y_train = to_categorical(y_train)
```

Modelling

The most important part of a deep learning problem is to choose the appropriate training model. While choosing a model, we have two options: - Transfer learning: Where you transfer a pre-trained model and weights and fit it to your model - Create your own model: Here you create your own model from scratch and train its weights; this gives you better control over your model

We use the Keras framework to build our model. Keras is a high-level API written in Python. It runs on top Tensorflow, CNTK, or Theano. It is relatively easy to use and understand and is very fast.

#Importing the required libraries

```
from keras.models import Sequential

from keras.layers import Dense, Dropout, Flatten, Convolution2D, MaxPooling2D

from keras.callbacks import EarlyStopping
```

We create a Sequential model to solve this problem. A Deep learning model consists of a number of layers connects to each other. The input data is passed through each layer, one-by-one. We use a Sequential model which is a linear stack of layers.

#We create a Sequential model using 'categorical cross-entropy' as our loss function and 'adam' as the optimizer.

```
model = Sequential()

model.add(Convolution2D(32, (3,3), activation='relu', padding='same',input_shape = (128,128,3)))

#if you resize the image above, change the input shape

model.add(Convolution2D(32, (3,3), activation='relu'))

model.add(MaxPooling2D(pool_size=(2,2)))


model.add(Convolution2D(64, (3,3), activation='relu', padding='same'))
```

```
model.add(Convolution2D(64, (3,3), activation='relu'))

model.add(MaxPooling2D(pool_size=(2,2)))

model.add(Dropout(0.25))


model.add(Convolution2D(128, (3,3), activation='relu', padding='same'))

model.add(Convolution2D(128, (3,3), activation='relu'))

model.add(MaxPooling2D(pool_size=(2,2)))

model.add(Dropout(0.25))


model.add(Flatten())

model.add(Dense(128, activation='relu'))

model.add(Dense(256, activation='relu'))

model.add(Dropout(0.25))

model.add(Dense(y_train.shape[1], activation='softmax'))

model.compile(loss = 'categorical_crossentropy', optimizer = 'adam', metrics = ['accuracy'])

model.summary()
```

Layer (type)	Output Shape	Param #
conv2d_13 (Conv2D)	(None, 128, 128, 32)	896
conv2d_14 (Conv2D)	(None, 126, 126, 32)	9248
max_pooling2d_7 (MaxPooling2	(None, 63, 63, 32)	0
conv2d_15 (Conv2D)	(None, 63, 63, 64)	18496
conv2d_16 (Conv2D)	(None, 61, 61, 64)	36928
max_pooling2d_8 (MaxPooling2	(None, 30, 30, 64)	0
dropout_7 (Dropout)	(None, 30, 30, 64)	0
conv2d_17 (Conv2D)	(None, 30, 30, 128)	73856
conv2d_18 (Conv2D)	(None, 28, 28, 128)	147584
max_pooling2d_9 (MaxPooling2	(None, 14, 14, 128)	0
dropout_8 (Dropout)	(None, 14, 14, 128)	0
flatten_3 (Flatten)	(None, 25088)	0
dense_5 (Dense)	(None, 128)	3211392
dense_6 (Dense)	(None, 256)	33024
dropout_9 (Dropout)	(None, 256)	0
dense_7 (Dense)	(None, 15)	3855

Total params: 3,535,279.0
Trainable params: 3,535,279.0
Non-trainable params: 0.0

Fig 6. Summary representation of the model.

Fig. 6, gives a summary of our model architecture. The input shape (128*128) is passed at the very top of the model. This input is run through all the layers sequentially. We can see from the figure that our model has approximately 3.5 million parameters (weights) to train. We use ‘categorical cross entropy’ as the loss function and ‘adam’ as the optimizer.

#We define an early stopping condition for the model. If the val_acc is the same three times, the model stops.

```
early_stops = EarlyStopping(patience=3, monitor='val_acc')
```

We now train this model using the input data. An epoch is when the model runs through the entire data once. batch_size refers to the number of training examples utilised in one iteration. validation_split splits our data into 70% training and 30% validation. The following code splits the input data into batches of 100 images and runs them through the model 10 times.

#Training the model for 10 epochs.

model.fit(x_train_norm, y_train, batch_size=100, epochs=10, validation_split=0.3, callbacks=[early_stops])

Lastly, we use our trained model to predict the labels on the test dataset.

#Now that we have built and trained our model, it is time to predict the test data.

test_img = []

for img in tqdm(test['image_name'].values):

test_img.append(read_img(TEST_PATH + img))

#Applying the same data-preprocessing steps on the test data.

x_test = np.array(test_img, np.float32) / 255.

x_test_norm = (test_img, np.float32) / 255.

#The test data is normalised

predictions = model.predict(x_test_norm)

predictions = np.argmax(predictions, axis= 1)

y_maps = dict()

y_maps = {v:k for k, v in Y_train.items()}

pred_labels = [y_maps[k] for k in predictions]

#Creating the submission file.

sub = pd.DataFrame({'row_id':test.row_id, 'detected':pred_labels})

sub.to_csv('submission.csv', index=False)