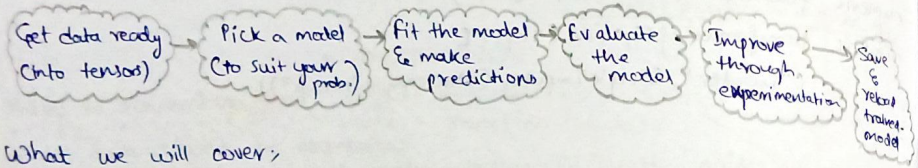
Why use 'Pipeline'? → reusability
→ consistent application
→ code organization
→ easy hyperparameter tuning.

Note: If Scikit-Learn version > 1.2 ; then "plot_roc_curve" will show error so, use "RocCurveDisplay" syntax!

Neural Networks : Deep Learning, Transfer Learning & Tensorflow 2

A tensorflow workflow:



What we will cover:

→ An end-to-end multi-class classification workflow with TensorFlow
→ Preprocessing image data (getting it into Tensors)
→ Choosing a deep learning model → fitting a model to the data. (learning patterns)
→ Making predictions with a model (using patterns)
→ Evaluating model predictions → Saving & loading models
→ Using a trained model to make predictions on custom data.

* Refer Scikit-learn cheatsheat i.e. "choosing the right estimator":
↳ https://scikit-learn.org/stable/machine_learning_map.html

* DATA ENGINEERING : {Key Terms & Concepts} (similar to "DMCT" subject in sem.5)

⇒ ETL (Extract, Transform, Load):

- ETL is a process used to extract data from various sources, transform it into a proper format, and load it into a data warehouse for analysis.
- Relevance: Data Scientists rely on ETL pipelines to prepare & clean data before modeling. Well structured data is the key for building accurate models.

⇒ Data Warehouse:

- A central repo. of structured & unstructured data integrated from multiple sources, optimized for analytical queries & reporting.
- Relevance: Data Warehouses support complex analytical queries used in DSML tasks.

⇒ Outlier:

- Outliers are data points significantly different from the rest of the dataset. Detecting & handling outliers is essential to ensure the quality & accuracy of models.
- Relevance: In ML, outliers can skew results & impact the performance of algo.'s, so they need to be addressed during data preprocessing.

⇒ Types of Databases:

- Relational Databases (RDBMS): These store data in structured tables with rows & columns (Ex: MySQL, PostgreSQL). They use SQL for querying data.
- NoSQL Databases: Non-relational databases, which handle unstructured, distributed data (Ex: MongoDB, Cassandra). Useful for big data scenarios.
- NewSQL Databases: Combine the scalability of NoSQL with ACID transactions of RDBMS. They offer high-performance for distributed workloads. (Ex: Google Spanner)
- Relevance: Choice of DB influences how you store, query & manage data in ML workflows, especially when handling large datasets.

⇒ Kafka & Stream Processing:

- Kafka: A distributed event streaming platform used for building real-time data pipelines and streaming applications
- Stream Processing: Processing data in real-time as it is being produced, often using tools like Kafka or Apache Flink
- Relevance: In ML, stream processing is crucial for real-time analytics and building models that need to handle continuous data, such as in fraud detection systems.

⇒ OLTP vs OLAP:

- OLTP (Online Transaction Processing): It's a type of DB system that handles day-to-day transactional data (ex: bank transactions). It is optimized for quick insertions & retrievals.
- OLAP (Online Analytical Processing): Designed for complex querying & data analysis, often used in data warehouses for aggregating large datasets.
- Relevance: OLAP supports ML tasks by providing pre-aggregated, analyzed data for model training.

Neural Networks :{CONTINUATION}

Before going deep into NN → we will explore Supervised Learning Projects:

HEART DISEASE PREDICTION:

Approach: 1) Prob. defn. 2) Data 3) Evaluation 4) Features 5) Modelling 6) Experimentation

- Prob. Defn.: Given clinical parameters about a patient, can we predict whether or not they have heart disease?

- Preparing the tools:

Used → pandas, numpy, matplotlib, seaborn { import matplotlib.pyplot as plt; %matplotlib inline }

→ sklearn → linear_model → LogisticRegression
→ neighbors → KNeighborsClassifier
→ ensemble → RandomForestClassifier

→ model_selection → train_test_split, cross_val_score, GridSearchCV, RandomizedSearchCV

→ metrics → confusion_matrix, classification_report, precision_score, recall_score, f1_score, RocCurveDisplay

- Workflow:

1) Importing Libraries:

⇒ Pandas and Numpy: These libraries are used for handling and processing your data. 'pandas' handles dataframes, while 'numpy' assists in numerical computation.

⇒ Matplotlib and Seaborn: Both are used for data visualization. Plots such as bar charts, histograms and correlation heatmaps are created to explore the relationships between different clinical parameters.

2) Data Loading & Exploration:

⇒ You load the dataset (~csv file) using 'pandas'. This dataset contains clinical parameters (ex: age, cholestrol, blood pressure) that will be used to predict heart disease.

⇒ You likely use functions like head(), info() and describe() to get an overview of the data, identify missing values, and understand the distribution of variables.

3) Data Preprocessing:

⇒ Handling missing values, normalizing data, or encoding categorical features (ex: gender) may be performed to ensure the data is clean and usable for modeling.

⇒ But here → our data was already normalized & we need not handle any missing data as there isn't any missing data.

4) Exploratory Data Analysis (EDA):

⇒ Correlation matrix / heatmap: using seaborn's heatmap, you visualize correlations between clinical features. Highly correlated features may be useful for prediction.

⇒ Histograms and Countplots: Help you understand the distributions of variables (ex: age, cholestrol levels) in patients with (or) without heart disease.

5) Model Selection:

⇒ Logistic Regression: A linear model to predict whether the patient has heart disease (binary classification).

⇒ KNeighborsClassifier: A KNN algorithm where the model predicts based on the proximity to other patients.

⇒ RandomForestClassifier: An ensemble method where multiple decision trees are used to make predictions.

6) Splitting the data:

⇒ train_test_split: The data is split into training and test sets. The training set is used to fit the model, and the test set evaluates performance on unseen data.

7) Model training & Cross-Validation:

⇒ Models are trained on the training set, and you use 'cross_val_score' for k-fold cross validation to ensure robust evaluation.

8) Hyperparameter Tuning:

⇒ RandomizedSearchCV & GridSearchCV: These techniques are used to search for the best hyperparameters to improve model performance.

9) Model Evaluation:

⇒ Confusion Matrix: used to visualize true positives, false (+)ves, true (-)ves and false (-)ves.

⇒ Classification Report: provides detailed performance metrics such as precision, recall, & F1-score.

⇒ Precision, Recall, F1-Score: These metrics evaluate how well your model distinguishes between patients with & without heart disease.

⇒ ROC Curve: A plot of the true (+)ve rate against the false (+)ve rate, showing how well the model distinguishes between classes at various thresholds.

10) Final Model Selection:

⇒ Based on the performance metrics from your evaluations, model selected is "LogisticRegression" with an accuracy of 81.96%.

Copilot Use:

- car_sales["Sale Date"] = pd.date_range("1/1/2023", periods = len(car_sales))
  - car_sales["Sale Date"]: (creates a new col)
  - pd.date_range: (generates a sequence of dates)
  - "1/1/2023": (start date for the date range)
  - periods = len(car_sales): (specifies the no. of dates to generate)

- clf = RandomForestClassifier(n_estimators = 10)
  - n_estimators = 10: (This specifies the no. of decision trees in the forest. For ex: above one will consist of 10 decision trees.)
  - → More trees generally improve the model's performance & stability, but also increase computational cost.
  - → Decision tree: it models decisions and their possible consequences in a tree-like structure, making it easy to understand & interpret.

- ROC Curve: → TPR = $\frac{\text{true (+)ve}}{\text{true +ve + false (-)ve}}$, FPR = $\frac{\text{False (+)ve's}}{\text{False (+)ve's + True (-)ves}}$
  - → Ideal ROC Curve — quickly rises towards top-left corner of the graph, indicating a high TPR and low FPR.
  - → High TPR = model correctly identifies a large proportion of actual (+)ves.
  - → Low FPR = model incorrectly identifies very few actual (-)ves as (+)ves.

- gs_model = GridSearchCV(model, pipe_grid, cv=5, verbose=2)
  gs_model.fit(x_train, y_train)
  gs_model.score(x_test, y_test) → The scoring happens on the best parameters found by GridSearchCV.

- clf = LogisticRegression(C=9.236, solver="liblinear")
  clf.fit(x_train, y_train)
  clf.coef_ → or array of (+)ve & (-)ve
    - specific to logistic regression model
    - indicates strength of the relationship btw each feature & the target variable in terms of log-odds.
    - (vs) Correlation Matrix → general statistic measure, which shows the linear relationship btw pairs of features (also possibly target variable). Values range: [-1, 1]

BLUE BOOK FOR BULLDOZERS:

Refer github → readme.md & linkedin → projects

- Classification → Accuracy, Precision, Recall, F1
  & Regression → R^2, MAE, MSE, RMSE
  (Default evaluation in Scikit-Learn (underlined))

1) Accuracy (vs) R^2: Both give an overall measure of how well the model performs, but accuracy is for classification (discrete) and R^2 is for regression (continuous). R^2 tells how well the model captures variance, while accuracy measures how many predictions are correct.

2) Precision (vs) MAE (Mean Absolute Error): While precision focuses on correct positive classifications, MAE focuses on how close regression predictions are to the actual continuous values. Precision emphasizes the quality of +ve predictions, whereas MAE emphasizes the overall prediction accuracy in terms of error size.

3) Recall (vs) MSE (Mean Squared Error): While recall focuses on capturing all true +ves in a classification report, MSE focuses on minimizing the squared error in continuous value prediction.

4) F1 Score vs RMSE (Root Mean Squared Error): While F1 balances 2 components (precision and recall) in a classification task, RMSE provides a measure of how large errors are in regression, while still balancing the influence of outliers. Both metrics offer a nuanced understanding of model performance by combining multiple error components.

Supervised Learning:

→ Model trained on labeled data. Each i/p comes with an associated o/p (or label), & the goal is to learn a mapping from i/p's to o/p's.

→ I/p: Labeled data (ex: images with labels like "cat" or "dog")

Goal: Predict the o/p (labels) for new, unseen data.

Common Tasks: → Classification: predict discrete labels (ex: spam vs not spam)
→ Regression: predict continuous values (ex: house prices)

Unsupervised Learning:

→ Model is trained on unlabeled data. The goal is to uncover hidden patterns, structures, or relationships in the data without any explicit o/p (or) label guidance.

→ I/p: Unlabeled data (ex: set of images without categories)

Goal: Find patterns or) grouping in the data

Common Tasks: → Clustering: group similar data points together (ex: market segmentation)
→ Dimensionality Reduction: reduce the no. of features while retaining essential info. (ex: PCA (Principal comp. analysis))

Which Regression metric should you use?

- R^2 is similar to accuracy. It gives you a quick indication of how well your model might be doing. Generally, the closer your R^2 value is to 1.0, the better the model. But it doesn't really tell exactly how wrong your model is in terms of how far off each prediction is.
- MAE gives a better indication of how far off each of your model's prediction are on average.
- As for MAE or MSE, because of the way MSE is calculated, squaring the differences between predicted values and actual values, it amplifies larger differences. Let's say we're predicting the value of houses:
  → Pay more attention to MAE: When being $10,000 off is twice as bad as being $5,000 off.
  → Pay more attention to MSE: When being $10,000 off is more than twice as bad as being $5,000 off.

* Precision = ratio of true +ves to total no. of +ve predictions
  & Recall = ratio of true +ves to the total no. of actual +ve samples

DOG VISION:

Refer github → readme.md & linkedin → projects

- `from matplotlib.pyplot import imread`
  `image = imread(filenames[97])`
  `image.shape` → o/p: (332, 500, 3)
  (332 → height of image in pixels → 332 rows; 500 → width of image in pixels → 500 columns; 3 → 3 color channels (RGB))
- Advantage of Tensors over Numpy arrays
  → GPU Acceleration
  → Automatic Differentiation
  → Integration with Deep Learning Frameworks

⇒ NumPy Arrays: Think of them as individual topics (or) chapters in a book. They're great for general numeric computations & data manipulation.

⇒ Tensors: When you want to do neural network stuff, you need to bring these chapters together in a specific format that the NN can understand & work with. This format is what we call tensors.

⇒ Tensorflow: Imagine Tensorflow as a specialized book on NN. The 1st page of this book asks for tensors because TF is designed to work with this specific format. Tensors are like the standardized chapters that TF can read, understand, and process efficiently.

- Comparing on no.'s btw [0,1) is more efficient than using the range [0,255] because: → Normalization: better numerical stability during computations.
  → Better performance of NN with [0,1)
  → Avoiding Underflow/Overflow
- data.map(get_image_label) → it applies 'get_image_label' function to each element in 'data' dataset.
- argmax → function used to find index of max. value in an array
  (ex: array.argmax())
- Transfer Learning: is a technique in ML where a model developed for 1 task is reused as the starting point for a model on a second, related task.
- Keras (it's a deep learning API) — Sequential API vs functional API: (2 diff ways to build Neural Network)
  - Sequential API:
    - Linear Stack: Creates model layer by layer.
    - Each layer has only 1 i/p tensor & 1 o/p tensor.
    - Straightforward & easy to use.
    - Doesn't supports model with multiple i/p's.
  - functional API:
    - Flexibility: It allows for creation of complex models with non-linear topology, shared layers, & multiple i/p's & o/p's.
    - DAG: models are defined as a directed acyclic graph (DAG) of layers, which can handle more sophisticated architectures.
- epoch: refers to one complete pass through the entire dataset (training). It also refers to the no. of times a training dataset passes through a learning algorithm.
- Ways to prevent model overfitting in deep learning NN:
  ⇒ Increase training data ⇒ Early stopping
  ⇒ Simplify the model ⇒ Reduce training time

- ⇒ Epoch Accuracy: Refers to the accuracy of the model on the training (or validation dataset after completing one epoch.
- ⇒ Epoch Loss: Refers to the loss value of the model on the training/validation dataset after completing 1 epoch.
- TensorFlow is a deep learning computing library used to gain insights out of unstructured data. Ex: photos, audio waves.
  - Why TensorFlow?
    - Write fast deep learning code in Python (able to run on a GPU)
    - Able to access many pre-built deep learning models
    - Whole stack: preprocess, model, deploy → we can do all these 3 things using TF
  - TF used in deep learning
  - TF Hub used in transfer learning.
- Keras: tf.keras is tensorflow's high-level API for building & training deep learning models. Its 3 key advantages are: → User-friendly, Easy to extend, Modular & composable

- Project Overview:

1) Get our workspace ready:
   ⇒ Import TF and TF Hub ⇒ Make sure we are using a GPU
2) Getting our data ready (turning into Tensors):
   ⇒ Accessing our data → 'labels_csv': contains labels of our data
   ⇒ Get images & their labels → 'filenames': contains path of training images in google drive
   'labels': stores breeds of dog in the form of an array (len. = 10222)
   'unique_breeds': stores unique label values → len. = 120
   'boolean_labels': contains T/F for each value in 'labels'.
   ie: 120 values for 1 'labels' in which 1 = true & 119 = false
   → len. = 10222 (same for all other labels)
3) Creating our own Validation Set:
   ⇒ 'X': 'filenames' → represents the location of every image in training set
   'y': 'boolean_labels' → represents the boolean value numpy representation of the labels.

⇒ X_train, X_val, y_train, y_val = train_test_split(X[:NUM_IMAGES], (NUM_IMAGES = 1000)
y[:NUM_IMAGES],
test_size = 0.2,
random_state = 42)
(len=800) (len=200) (len=800) (len=200)

4) Preprocessing Images (turning images into Tensors):

⇒ • Take an image filepath as input
• Use TensorFlow to read the file and save it to a variable, image
• Turn our image (a jpg) into Tensors
• Normalize our image (convert colour channel values from 0-255 to 0-1)
• Resize the image to be a shape of (224, 224)
• Return the modified image.

{refer last page of this chapter}

⇒ → IMG_SIZE = 224
→ function → "process_image(image_path, img_size=IMG_SIZE)"
↳ takes an image file path & turns the image into a Tensor.

5) Turning our data into batches:

⇒ → function → "get_image_label(image_path, label)"
↳ Takes an image file path name and the associated label, process the image (process_image) and returns a tuple of (image, label)

(demo)
⇒ Converting our 'boolean_labels': 'y' into Tensors → tf.constant(y[37])

⇒ → BATCH_SIZE = 32
→ function → "create_data_batches(X, y=None, batch_size=BATCH_SIZE, valid_data=False, test_data=False)"

Creates batches of data out of image(X) and label (y) pairs. Shuffles the data if it's training data but doesn't shuffle if it's validation data. It also accepts test data as input (no labels).

⇒ train_data = create_data_batches(x_train, y_train)
val_data = create_data_batches(x_val, y_val, valid_data=True)

6) Visualizing Data Batches:

⇒ → function → "show_25_images(images, labels)"
↳ (displays a plot of 25 images & their labels from a data batch.)

⇒ train_images, train_labels = next(train_data.as_numpy_iterator())

⇒ val_images, val_labels = next(val_data.as_numpy_iterator())

(as_numpy_iterator(): a method which returns iterator which converts all elements of the dataset to numpy i.e. train_images='x' & train_labels=y)

7) Building Model:

⇒ Before we build a model, there are few things we need to define:
↳ The i/p shape (our images shape, in the form of Tensors) to our model.
↳ INPUT_SHAPE = [None, IMG_SIZE, IMG_SIZE, 3]
(batch) (height) (width) (color channels)
↳ setting to 'None' represents a dynamic batch size that will be determined when feeding data to our model.
↳ The o/p shape (image labels, in the form of Tensors) to our model
↳ OUTPUT_SHAPE = len(unique_breeds)
↳ The URL of model we want to use from TensorFlow Hub.
↳ MODEL_URL = "https://.."

⇒ Now we've got our inputs, outputs and model ready to go. Let's put them together into a Keras deep learning model. Knowing this, let's create a func. which:
↳ • Takes the i/p shape, output shape & the model we've chosen as parameters.
• Define the layers in a Keras model in sequential fashion
• Compiles the model (says it should be evaluated & improved).
• Builds the model (tells the model the i/p shape it'll be getting)
• Returns the model.

⇒ → function → "create_model(input_shape=INPUT_SHAPE, output_shape=OUTPUT_SHAPE, model_url=MODEL_URL)"
(Builds & compiles a Keras sequential model using a pre-trained model from TensorFlow Hub.)
↳ model = create_model()

8) Creating Callbacks:

(helper func.s a model can use during training to do such things as save its progress, check its progress or stop training early if a model stops improving)

⇒ We will create 2 callbacks, one for TensorBoard which helps track our model progress & another for early stopping which prevents our model from training for too long.

↳ TensorBoard Callback: → Load the TensorBoard notebook extension
→ Create a TensorBoard callback which is able to save logs to a directory & pass it to our model.s 'fit()' function.
→ function → "create_tensorboard_callback()"
→ Visualize our models training logs with the '%tensorboard' magic func. (we'll do this after model training)

↳ Early Stopping Callback: early_stopping = tf.keras.callbacks.EarlyStopping(
monitor="val_accuracy",
patience=3
)

9) Training a model (on subset of data)

⇒ Let's create a func. which trains a model.
- Create a model using 'create_model()'
- Setup a TensorBoard callback using 'create_tensorboard_callback()'
- Call the fit() func on our model passing it the training data, validation data, no. of epochs to train for (NUM_EPOCHS) & the callbacks we'd like to use
- Return the model.

⇒ function → "train_model()"
↳ trains a given model & returns the trained model.
↳ model = train_model()

10) Making & evaluating predictions using a trained model.

⇒ predictions = model.predict(val_data, verbose=1) } Make predictions on the validation data
↳ .shape = (200, 120)
len(y_val) len(unique_breeds)

⇒ function → "get_pred_label(prediction_probabilities)"
↳ turns an array (size=120) of prediction probabilities into a label

⇒ Now since our validation set is still in a batch dataset, we'll have to unbatchify to make predictions on the validation images & then compare those predictions to the validation labels (truth labels)

⇒ function → "unbatchify(data)"
↳ takes a batched dataset of (image, label) tensors & return seperate arrays of images & labels.
↳ val_images, val_labels = unbatchify(val_data)

⇒ Now we've got ways to get → Prediction labels
→ Validation labels (truth labels)
→ Validation images

Let's make a func. to make these all a bit more visualize.
We'll create a function which: (= predictions)
- Takes an array of prediction probabilities, an array of truth labels & an array of images & an integer.
- Convert the prediction probabilities to a predicted model.
- Plot the predicted label, it's predicted probability, the truth label & the target image on a single plot.

⇒ function → "plot_pred(prediction_probabilities, labels, images, n=1)"
(val_labels) (val_images)
↳ View the prediction, ground truth & image for sample 'n'

⇒ Now we've got one func. to visualize our models top predictions, let's make another to view our models top 10 predictions. This func. will:
- Take an i/p of prediction probabilities array & a ground truth array & an integer
- Find the prediction using get_pred_label()
- Find top 10 → prediction possibilities indexes
→ prediction probabilities values
→ prediction labels
- Plot the top 10 predictions probability values & labels, coloring the true label green

⇒ function → "plot_pred_conf(prediction_probabilities, labels, n=1)"
(val_labels)
↳ Plots the top 10 highest prediction confidences along with the truth label for sample 'n'.
(Prediction probabilities are also known as confidence levels.)

11) Saving & Reloading a model:

⇒ function → "save_model(model, suffix=None)"
↳ Saves a given model in models directory & appends a suffix (string)

function → "load_model(model_path)"
↳ loads a save model from : {model_path}

12) Training on full data:

⇒ full_data = create_data_batches(x, y)
→ full_model = create_model()
→ full_model_tensorboard = create_tensorboard_callback()
→ full_model_early_stopping = tf.keras.callbacks.EarlyStopping(
monitor="accuracy",
patience=3
)
→ fitting, saving & loading → loaded_full_model = load_model('model_path')

13) Making predictions on test dataset:

⇒ Since our model has been trained on images in the form of Tensor batches, to make predictions on the test data, we'll have to get it into the same format. Also, we created 'create_data_batches()' earlier which can take a list of filenames as i/p & convert them into Tensor batches.

To make predictions on the test data, we'll:

- Get the test image filepaths.
- Convert the filenames into test data batches using `create_data_batches()` and setting the `test_data` parameter to `True` (since the test data doesn't have labels).
- Make a predictions array by passing the test batches to the `predict()` method called on our model.

⇒
- `test_path = "drive/MyDrive/Dog Vision/dog-breed-identification/test/"`
- `test_filenames = [test_path + fname for fname in os.listdir(test_path)]`
  - len. = 10357
- `test_data = create_data_batches(test_filenames, test_data=True)`
- `test_predictions = loaded_full_model.predict(test_data, verbose=1)`
- Saving predictions (Numpy array) to csv file
  - `np.savetxt("drive/MyDrive/Dog Vision/dog-breed-identification/preds_array.csv", test_predictions, delimiter=",")`
- Loading predictions (Numpy array) from csv file
  - `test_predictions = np.loadtxt("<above-file-path>", delimiter=",")`
    - `.shape = (10357, 120)` (rows) (cols)

14) Preparing test dataset predictions for kaggle:

⇒ On kaggle, we find that it wants our models prediction probability outputs in a DataFrame with an ID and a col. for each diff. dog breed. To get data in this format, we'll:

- Create a pandas DataFrame with an ID column as well as a column for each dog breed
- Add data to the ID column by extracting the test image IDs from their filepaths.
- Add data (the prediction probabilities) to each of the dog breed columns
- Export the DataFrame as a CSV to submit it to Kaggle.

⇒
- `preds_df = pd.DataFrame(columns=["id"] + list(unique_breeds))`
  - (0 x 121 cols.)
- `test_ids = [os.path.splitext(path)[0] for path in os.listdir(test_path)]`
- `preds_df["id"] = test_ids` → add test image IDs to predictions DataFrame
- `preds_df[list(unique_breeds)] = test_predictions`
  - adding prediction probabilities to each dog breed column

⇒ Save our predictions dataframe to CSV for submission to kaggle.
- `preds_df.to_csv("drive/MyDrive/Dog Vision/dog-breed-identification/full_model_predictions_submission_mobilenetv2.csv", index=False)`

15) Making predictions on custom images:

⇒ To make predictions on custom images, we'll:

- Get the filepaths of our own images
- Turn the filepaths into data batches using `create_data_batches()`. And since our custom images won't have labels, we set the `test_data` parameter to `True`.
- Pass the custom image data batch to our model's `predict()` method.
- Convert the prediction o/p probabilities to predictions labels.
- Compare the predicted labels to the custom images.

⇒ Refer github → `readme.md` file for code.

- Log Loss: is a metric used to measure the difference between the predicted probabilities and the actual labels in machine learning.

- Difference btw ↪ Resizing our image & then normalizing it ↪ Normalizing it & then resizing our image

→ Key Differences:

⇒ Interpolation Accuracy: When resizing first, the interpolation algo. has access to the full range of pixel values [0, 255]. This allows for better accuracy in the resizing process. If normalization is applied before resizing, the interpolation is working with smaller, fractional pixels (btw 0 & 1), which could result in minor loss of precision.

⇒ Normalization Impact: If you normalize first & then resize, the normalization operation has already scaled the values down. The resizing might introduce subtle inaccuracies, as the precision is slightly reduced.

→ Which is better:

⇒ Resizing first & then normalizing is generally preferred. Resizing is typically more accurate when performed on the original image values. Normalization can be applied afterward without affecting the resizing quality.