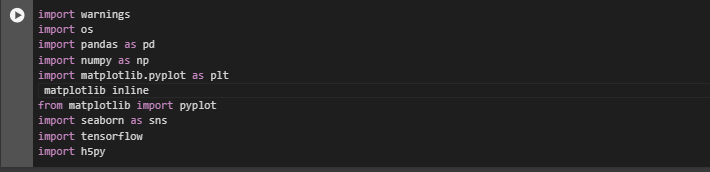
**Project Title: Train a CNN on the SVHN Dataset for Classification**

**Ashraf Hussain**

**ashrafmian2001@gmail.com**

**github.com/ashrafhussain044**

****

**summary :**

1. `**import warnings`:** This module manages warnings in your code. You might use it to filter out warnings or control their display, helping to focus on critical output without being overwhelmed by minor warnings.

2. `**import os`:** This module provides functions for interacting with the operating system, such as managing directories and files, setting paths, and handling environment variables.

3. `**import pandas as pd**`: The `pandas` library (imported as `pd`) is used for data manipulation and analysis, especially with structured data. It provides powerful data structures like DataFrames for easy handling and analysis.

4. `**import numpy as np`:** `numpy` (imported as `np`) is a fundamental package for scientific computing in Python. It’s especially useful for handling arrays and performing mathematical operations.

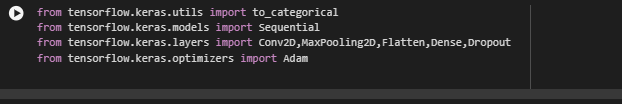
5. `**import matplotlib.pyplot as plt` and `matplotlib inline`:** `matplotlib.pyplot` (imported as `plt`) is a plotting library used for creating static, animated, and interactive visualizations. The `matplotlib inline` command is typically used in Jupyter notebooks to display plots inline within the notebook itself.

6. `**from matplotlib import pyplot`:** This imports the `pyplot` module specifically, which provides functions for plotting. Since you already imported it as `plt`, you might not need this line unless you plan to use it differently.

7. `**import seaborn as sns`:** `seaborn` (imported as `sns`) is a visualization library built on top of `matplotlib` that provides a high-level interface for creating attractive and informative statistical graphics.

8. `**import tensorflow`:** This imports TensorFlow, a deep learning library commonly used for building and training machine learning models, especially neural networks.

9. **`import h5py`:** `h5py` allows for reading and writing HDF5 files, a file format often used to store large datasets in scientific computing.



Here’s a breakdown of each of these imports:

**1. `from tensorflow.keras.utils import to\_categorical`:**

- This function converts labels (often integers) into a categorical format (one-hot encoded), which is essential for classification tasks in neural networks, especially when working with multi-class data.

2. **`from tensorflow.keras.models import Sequential`:**

- `Sequential` is a model type in Keras where layers are stacked sequentially, forming a linear stack of layers. It’s simple and ideal for building straightforward, layer-by-layer neural networks.

3**. `from tensorflow.keras.layers import Conv2D, MaxPooling2D, Flatten, Dense, Dropout`:**

- `**Conv2D`**: This layer performs a 2D convolution, which is often used in image processing for detecting features.

**- `MaxPooling2D**`: This layer performs max pooling, which reduces the spatial dimensions of the data, thereby helping with dimensionality reduction and computational efficiency.

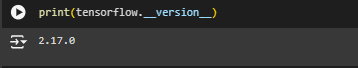
- **`Flatten`:** This layer flattens the input, converting 2D data into a 1D vector, which is necessary before feeding data into dense layers.

**- `Dense**`: A fully connected layer, which is the standard layer in neural networks, where each neuron is connected to all neurons in the previous layer.

**- `Dropout**`: This layer helps prevent overfitting by randomly setting a fraction of input units to zero at each update during training.

**4. `from tensorflow.keras.optimizers import Adam`**:

- `Adam` is an optimizer that combines the benefits of two other extensions of stochastic gradient descent, Adaptive Gradient Algorithm (AdaGrad) and Root Mean Square Propagation (RMSProp). It is efficient and commonly used for training deep learning models.



is a command used to check which version of TensorFlow is currently installed.



`**warnings.filterwarnings('ignore'**)` tells Python to ignore and hide all warning messages, keeping the output cleaner.



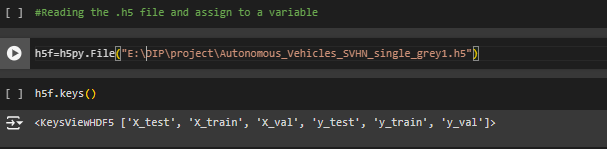
`**pd.options.display.max\_columns=None**` configures pandas to display all columns in the output, regardless of how many there are, making it easier to view large DataFrames.



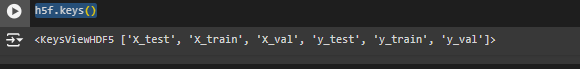
**`pd.options.display.float\_format='{:.7f}'**.format` sets the display format for floating-point numbers in pandas to show seven decimal places, ensuring numerical data is presented with consistent precision when printed.



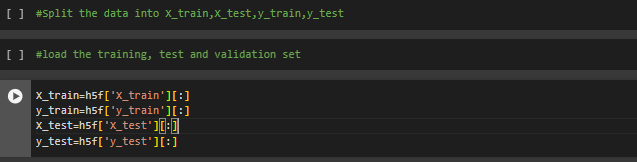
**`pd.options.display.max\_rows=None**` configures pandas to display all rows of a DataFrame, no matter how many there are, so you can see the full dataset without any rows being hidden.



**`h5f = h5py.File("E:\\DIP\\project\\Autonomous\_Vehicles\_SVHN\_single\_grey1.h5")`** opens the HDF5 file at the specified path, allowing you to read or interact with its data.



**`h5f.keys()`** retrieves a list of all the top-level groups or datasets in the opened HDF5 file, helping you see what data is available inside.



These lines of code extract data from the HDF5 file and store it in variables:

- **`X\_train = h5f['X\_train'][:]`:** Loads the training input data (features) into the variable `X\_train`.

**- `y\_train = h5f['y\_train'**][:]`: Loads the training output data (labels) into the variable `y\_train`.

- **`X\_test = h5f['X\_t**est'][:]`: Loads the testing input data into the variable `X\_test`.

- `y**\_test = h5f['y\_test']**[:]`: Loads the testing output data into the variable `y\_test`.

The `[:]` notation retrieves all the data from each specified dataset in the HDF5 file.



- `X\_train.shape`: Shows the dimensions of the training feature dataset, indicating the number of samples and features.

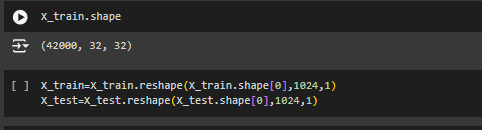
- `X\_test.shape`: Displays the dimensions of the testing feature dataset, also indicating the number of samples and features.

- `y\_train.shape`: Provides the shape of the training labels dataset, indicating the number of samples corresponding to `X\_train`.

- `y\_test.shape`: Displays the shape of the testing labels dataset, indicating the number of samples corresponding to `X\_test`.



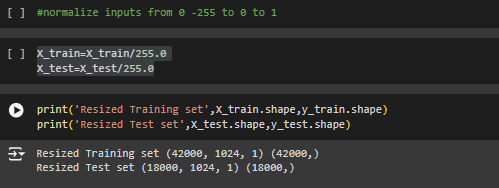
The function `img\_lab(n)` displays `n` grayscale images from the `X\_train` dataset in a single row and hides the axes. It also prints the corresponding labels from the `y\_train` dataset for each displayed image.



* The command `X\_train.shape` returns the dimensions of the training dataset, showing the number of images, their height, and width, typically in the format `(num\_samples, height, width)`, such as `(1000, 28, 28)` for 1000 images of 28x28 pixels each.
* The code snippets reshape the `X\_train` and `X\_test` datasets as follows:

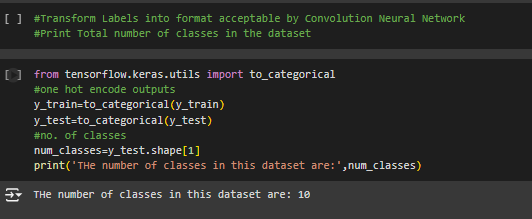
- `X\_train = X\_train.reshape(X\_train.shape[0], 1024, 1)`: Reshapes `X\_train` to have `num\_samples` (the first dimension), with each image flattened into a vector of size 1024 (the second dimension), and adds a third dimension of size 1.

- `X\_test = X\_test.reshape(X\_test.shape[0], 1024, 1)`: Similarly reshapes `X\_test` to have the same structure, converting each image into a vector of size 1024 with a third dimension of size 1.

* 
* `X\_train = X\_train / 255.0`: Divides each pixel value in the `X\_train` dataset by 255 to scale the values to a range of 0 to 1, which is common for image data to help improve model performance during training.
  + `X\_test = X\_test / 255.0`: Similarly scales the pixel values in the `X\_test` dataset to the range of 0 to 1.

 **print('Resized Training set', X\_train.shape, y\_train.shape)**: This prints the shape of the resized X\_train dataset (the number of samples and the new dimensions of each image) and the shape of the y\_train labels (the number of labels corresponding to the training images).

 **print('Resized Test set', X\_test.shape, y\_test.shape)**: This prints the shape of the resized X\_test dataset and the shape of the y\_test labels.



1. `from tensorflow.keras.utils import to\_categorical`:

- This imports the `to\_categorical` function from Keras, which is used for converting class labels to one-hot encoded format.

2. `y\_train = to\_categorical(y\_train)`:

- This applies one-hot encoding to the `y\_train` labels, transforming each label into a binary array that indicates the presence of a class. For example, if the original label is `2` in a dataset with 4 classes, it will be converted to `[0, 0, 1, 0]`.

3. `y\_test = to\_categorical(y\_test)`:

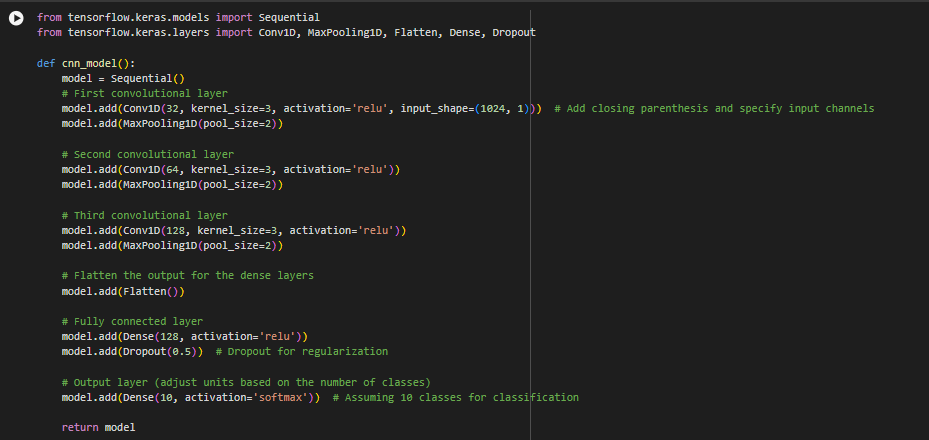
- Similarly, this applies one-hot encoding to the `y\_test` labels.

4. `num\_classes = y\_test.shape[1]`:

- This retrieves the number of classes from the shape of the one-hot encoded `y\_test` array, where the second dimension corresponds to the number of classes.

5. `print('The number of classes in this dataset are:', num\_classes)`:

- This prints the total number of classes in the dataset based on the one-hot encoding.



**1. Imports:**

**- `from tensorflow.keras.models import Sequential`:** Imports the `Sequential` model, which allows for building a model layer by layer.

**- `from tensorflow.keras.layers import Conv1D, MaxPooling1D, Flatten, Dense, Dropout`:** Imports necessary layers for the CNN, including convolutional, pooling, flattening, fully connected, and dropout layers.

**2. Function Definition:**

- `def cnn\_model():`: Defines a function named `cnn\_model` that builds and returns the CNN architecture.

**3. Model Initialization:**

- `model = Sequential()`: Initializes a sequential model.

**4. Convolutional Layers:**

**- First Layer:**

- `model.add(Conv1D(32, kernel\_size=3, activation='relu', input\_shape=(1024, 1)))`: Adds a 1D convolutional layer with 32 filters, a kernel size of 3, and ReLU activation. The `input\_shape` specifies that each input sample has 1024 timesteps and 1 feature.

**- Pooling:**

- `model.add(MaxPooling1D(pool\_size=2))`: Adds a max pooling layer that reduces the spatial size by taking the maximum value in each pool of size 2.

**- Second Layer:**

- `model.add(Conv1D(64, kernel\_size=3, activation='relu'))`: Adds another convolutional layer with 64 filters.

- `model.add(MaxPooling1D(pool\_size=2))`: Adds another max pooling layer.

**- Third Layer:**

- `model.add(Conv1D(128, kernel\_size=3, activation='relu'))`: Adds a convolutional layer with 128 filters.

- `model.add(MaxPooling1D(pool\_size=2))`: Adds another max pooling layer.

**5. Flattening:**

- `model.add(Flatten())`: Flattens the output from the convolutional layers into a 1D array to feed into the fully connected layers.

**6. Fully Connected Layers:**

- `model.add(Dense(128, activation='relu'))`: Adds a dense layer with 128 neurons and ReLU activation.

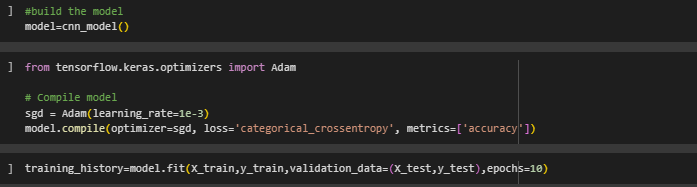
- `model.add(Dropout(0.5))`: Adds a dropout layer with a rate of 50% to help prevent overfitting during training.

**7. Output Layer:**

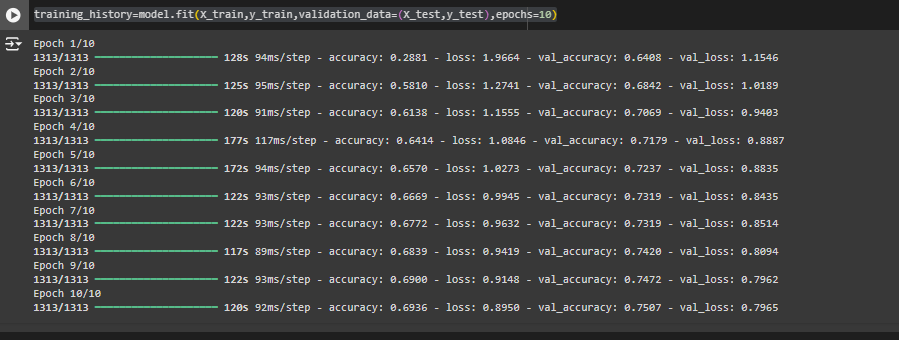
- `model.add(Dense(10, activation='softmax'))`: Adds the output layer with 10 neurons (for the 10 classes) and softmax activation, which outputs probabilities for each class.

**8. Return Statement:**

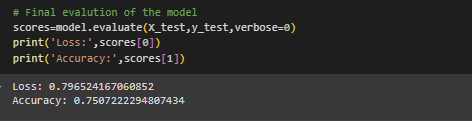
- `return model`: Returns the constructed CNN model.



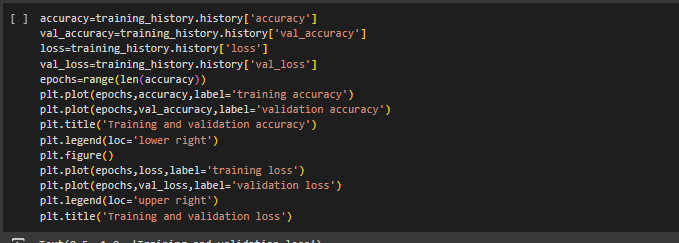
This code prepares the CNN model for training by configuring it with the Adam optimizer, specifying the loss function for multi-class classification, and setting accuracy as the evaluation metric. After this step, you can proceed to train the model using the fit method.



This code trains the CNN model on the training data while evaluating its performance on the validation data for **10 epochs**, allowing you to monitor the model's learning progress and generalization ability. The training history, which contains metrics like loss and accuracy for both training and validation sets, is stored in training\_history for later analysis.



This code evaluates the performance of the trained CNN model on the test dataset, providing the loss and accuracy metrics. This evaluation helps you understand how well the model generalizes to unseen data after training.



**1. Extracting Metrics:**

**- `accuracy = training\_history.history['accuracy']`:** Retrieves the training accuracy values recorded during the training process.

- **`val\_accuracy = training\_history.history['val\_accuracy']`:** Retrieves the validation accuracy values recorded during training.

- **`loss = training\_history.history['loss**']`: Retrieves the training loss values.

**- `val\_loss = training\_history.history['val\_loss']`:** Retrieves the validation loss values.

**2. Setting Epochs:**

- **`epochs = range(len(accuracy))`:** Creates a range object for the number of epochs based on the length of the accuracy list, which will be used as the x-axis for the plots.

**3. Plotting Accuracy:**

- **`plt.plot(epochs, accuracy, label='training accuracy')`:** Plots the training accuracy against the number of epochs.

**- `plt.plot(epochs, val\_accuracy, label='validation accuracy')`:** Plots the validation accuracy against the number of epochs.

**- `plt.title('Training and validation accuracy')`:** Sets the title for the accuracy plot.

**- `plt.legend(loc='lower right')`**: Displays the legend in the lower right corner of the plot.

**4. Creating a New Figure for Loss:**

- **`plt.figure()`:** Creates a new figure for the loss plot.

**5. Plotting Loss:**

**- `plt.plot(epochs, loss, label='training l**oss')`: Plots the training loss against the number of epochs.

- `**plt.plot(epochs, val\_loss, label='validation loss')`:** Plots the validation loss against the number of epochs.

- `**plt.legend(loc='upper right')`:** Displays the legend in the upper right corner of the plot.

**- `plt.title('Training and validation loss')`:** Sets the title for the loss plot.

