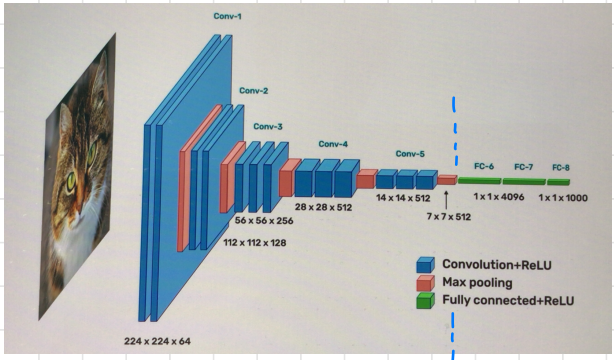
CONVOLUTIONAL NEURAL NETWORKS (CNN) FUNDAMENTALS

Im the context of an IMAGE CLASSIFICATION PROBLEM
⇒ CNN are developed to process image data

Ex. Net Architecture VGG-16 (developed IM 2014)



Architecture composed by a series of convolutional blocks followed by Fully connected layers

convolutional blocks → extract meaning full features from the INPUT image

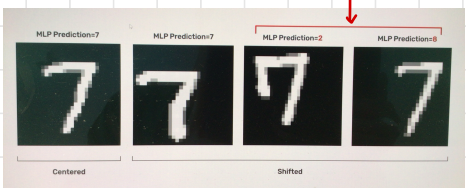
Fully connected layers → than passed to the NN layers for the classification task

⇑

CNN are more suited to process image data with respect to MLP

↳ NOT best choice because

- MLP are NOT TRANSLATION INVARIANT!
  network react differently depending on where the digit pixels is located in the image frame!
  When off-centered image ... predictions are less Reliable!



- MLP have excessive number of parameters

↓

it makes MLP prone to OVERFITTING! because image data has a large number of inputs coupled with fully connected (dense) MLP layers

ex.

224×224×3 (RGB image) → flattened im 224·224·3 1D vector of pixels → overall |W| ≈ 300 Billion parameters!

[size used by VGG-16 CNN]

requires lots of data AND is prone to OVERFITTING on training dataset

↓

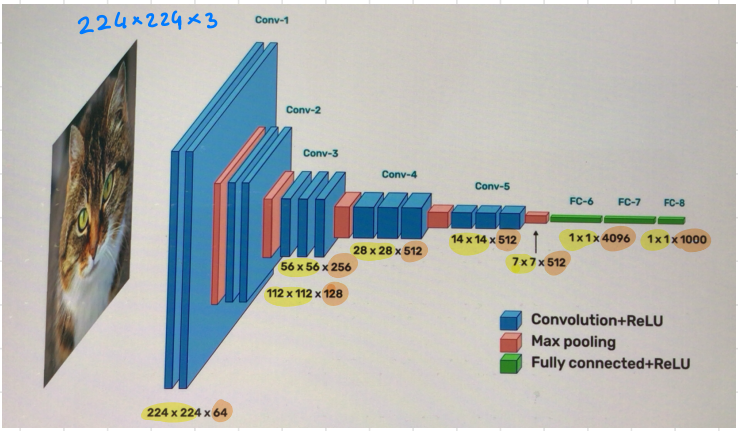
CNN are better to process image data

Developed for effectively and efficiently process image data

VGG-16 (example, VGG-16 to study its components and operations for each individual layer)

feature extractor (upstream) ≃ "Backbone" / "Body" of the Network

classifier (downstream) ≃ "Head" of the Network



5 convolutional blocks constitute the model feature extraction segment → 3 Fully connected layers which transform the extracted feature in class prediction at the final output layer

this is a sequence of layers used in feature extraction

Each layer has a spatial dimension and a depth

W × H × D

↑

this represents the shape of the data as it flows through the network

VGG-16 take as input RGB images of 224 × 224 × 3

↓

as it pass through the CNN its shape is transformed!
its spatial dimension decrease while the depth increase

↑

depth ≈ Number of channels

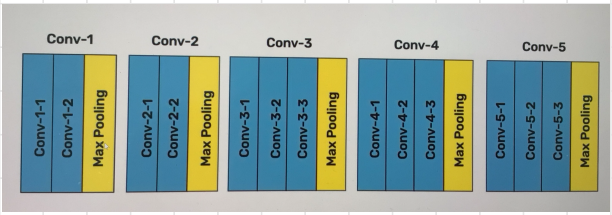
↓

the data is transformed with CONVOLUTIONAL and POOLING layers

↑

foundamental layers of CNN

- **FEATURE EXTRACTOR** the specific layers in a convolutional block vary depending on the Architecture

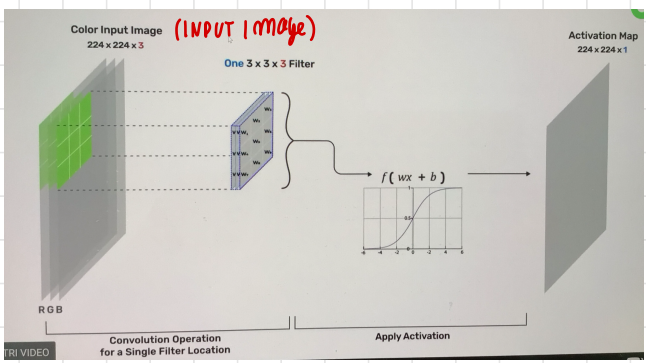


↓

at the most basic level, a conv block contains at least one 2D conv layer followed by pooling layer

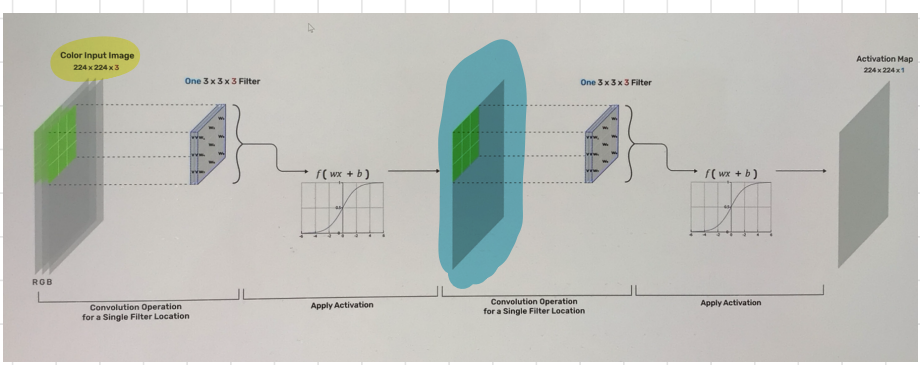
(sometimes additional layers are incorporated)

Convolutional layers (general concept) ≃ "EYES of the CNN"



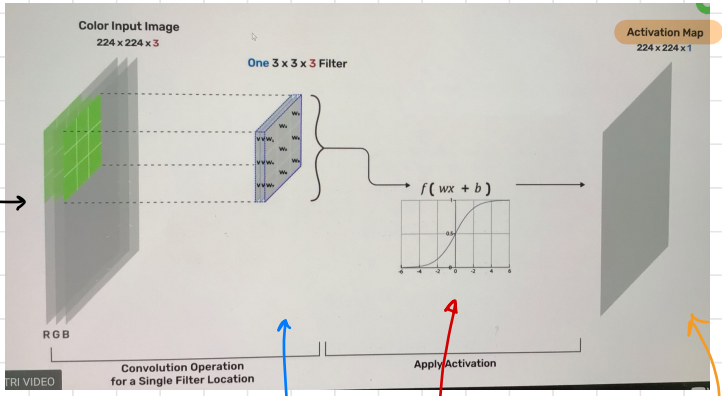
↓ Neurons in a conv. layer looks for specific features

At basic layer, it has INPUT an RGB 2D array OR an output from a previous layer



same weights are used to process different part of the image, because feature patterns are translation invariants as kernel moves!

FIRST comv. layer



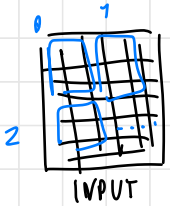
(VGG-16 take an image 224x224x3)

Filters are used to process input data

the filter output is passed to an ACTIVATION FUNCTION

than the output of the activation function populates the corresponding entry in the output (ACTIVATION MAP)

filters "move across the input"



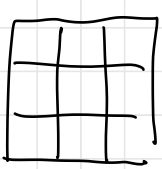
and at each filter location CONVOLUTION operation is performed → producing a single number x

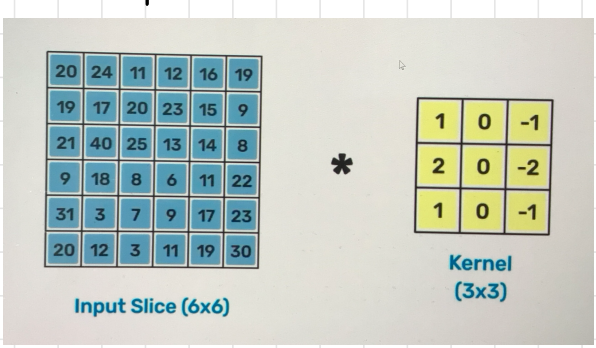
the ACTIVATION MAP contains a summary of the meaningful features from the input via
1. conv. process
2. transform by Activation

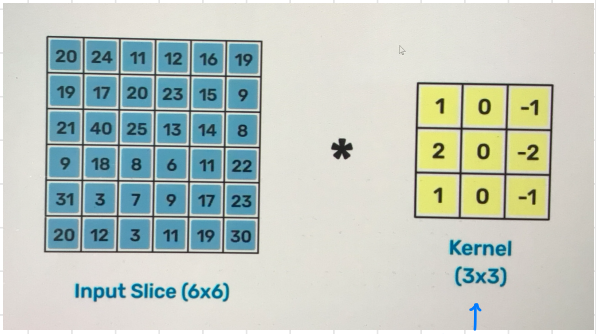
⇓

BUT how CONVOLUTION OPERATION is performed?

IN this operation, a small filter (KERNEL) ~ usually 3×3 or 5×5 is used to process INPUT DATA





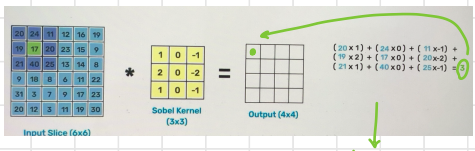


→ CONVOLUTION

the filter (KERNEL) pass over the input slice and a element wise operation is done (DOT PRODUCT)

$$\sum I_{ij} \cdot K_{ij} = K$$

well know kernel often used SOBEL KERNEL (used to detect VERTICAL edges)



the output is a number for a given filter location

this KERNEL has fixed elements that are weights learned by the Network during training

$$K = \begin{bmatrix} w_1 & w_2 & w_3 \\ w_4 & w_5 & w_6 \\ w_7 & w_8 & w_9 \end{bmatrix}$$

the filter is slide across the input slice by N pixels at a time

"STRIDE" represent the sliding step of the filter

the region of the input where the KERNEL act is called RECEPTIVE FIELD

the output location correspond to the center of the Receptive Field in the input slice

then OUTPUT has a smaller spatial size than the INPUT slice...

this depends on how the FILTER is placed over input data (IF NOT extending over the boundary)

↓

- padding techniques can be used to PAD the INPUT image so that in/out has same size
- stride larger than one reduces the output size (less convolution operation)

specific KERNELS can detect Basic structure in an image

↓ IMAGE KERNELS (https://setosa.io/ev/image-kernels/)

When Kernel moves, it is able to detect a specific feature (structure) in the INPUT image → as an higher intensity output

different Filters can be used to detect different edges in the input image

↑

CNN generalize this concept of feature extraction because Kernel weights are learnt during the TRAINING process ← CNN can detect many different types of features automatically

↓

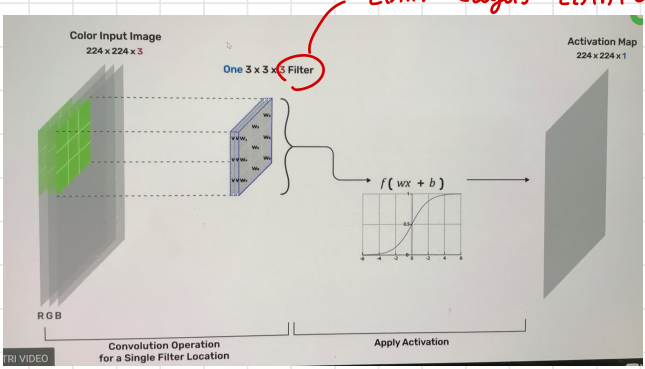
the Network is incentivite to find Filters that extract meaningfull features

- After the CONVOLUTION operation, the output is passed through an ACTIVATION FUNCTION to produce an Activation map

conv layers contains many filters typically

↓

each conv. layer produces multiple activation maps

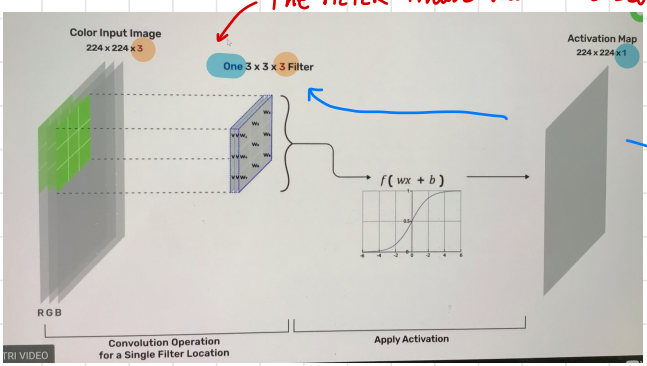


Convolutional Layer Keypoints

* The depth of a filter (channels) must match the depth of the input data (i.e, the number of channels in the input).
* The spatial size of the filter is a design choice, but 3x3 is very common (or sometimes 5x5).
* The number of filters is also a design choice and dictates the number of activation maps produced in the output.
* Multiple activations maps are sometimes collectively referred to as "an activation map containing multiple channels." But we often refer to each channel as an activation map.
* Each channel in a filter is referred to as a kernel, so you can think of a filter as a container for kernels.
* A single-channel filter has just one kernel. Therefore, in this case, the filter and the kernel are one and the same.
* The weights in a filter kernel are initialized to small random values and are learned by the network during training process.
* The number of trainable parameters in a convolutional layer depends on the number of filters, the filter size, and the number of channels in the input.

EXAMPLE:

the FILTER must have the same depth of the INPUT (no specific requirement on spacial size WxH)



here we use JUST one filter F_1 of three kernels (K_1, K_2, K_3) containing overall $3 \times 3 \times 3 \times 1 = 27$ weights + 1 bias → 28 parameters to tune

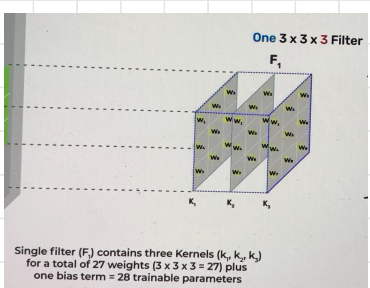
because we use ONE FILTER, the output depth is ONE

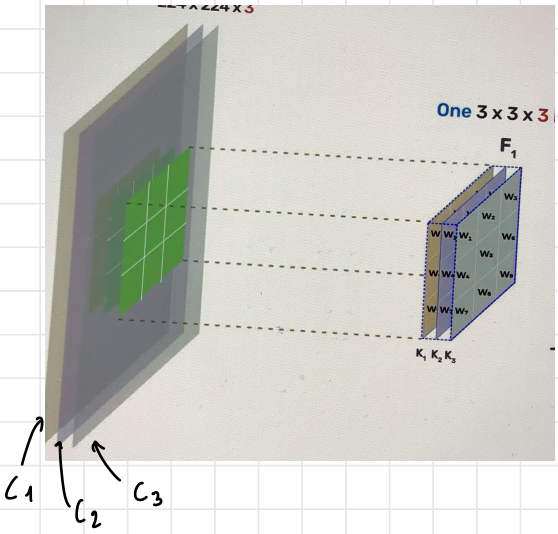
↓

we produce JUST one channel Activation map as output convolving one 3 channels filter with 3 channel input

⇓

the convolutional operation is performed for each channel separately



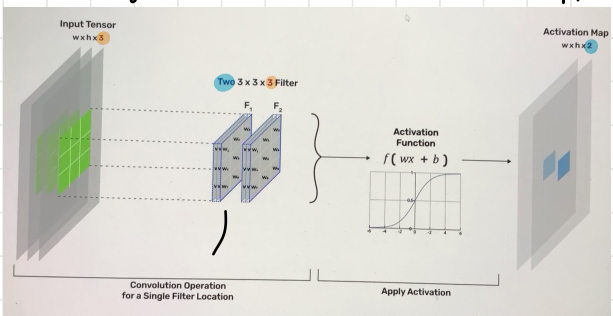


K_1 on C_1, K_2 on C_2, K_3 on C_3
each kernel on each channel
↓
then weighted sum of each channels + bias
$f(Wx + b)$
↑ goes through an activation function
↓
each Filter convolution step produces a single number in the output activation map, overall generated by sliding the FILTER by its STRIDE

to summarize

- # input channels = # kernels
- # of Filters = # of Activation maps (output channels) design choice

In a general case with two filters applied on a $W \times h$ input image



the "INPUT" is the generic conv. layer input which can be the input img for first layer or the output from a previous conv. layer (Activ. Map)

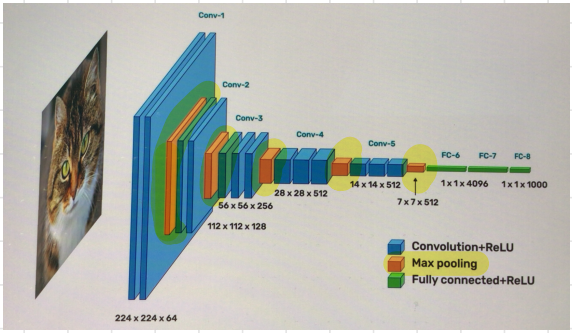
the depth 3 does NOT necessary corresponds to RGB channels

each filter has $3 \times 3 \times 3 = 27$ weights $\times 2 = 54 +$ (2 bias) $= 56$ tunable params
↑ one bias ∀ filter

Intuition on Filters usage ?...

CNN uses features to characterize image content

- what are POOLING LAYERS?



In most CNN Architecture, typically spatial dimensions of the data is reduced periodically via pooling layers

↓

used typically after a series of conv. layers to reduce spatial size of the activation map of preceding conv. layer

↓

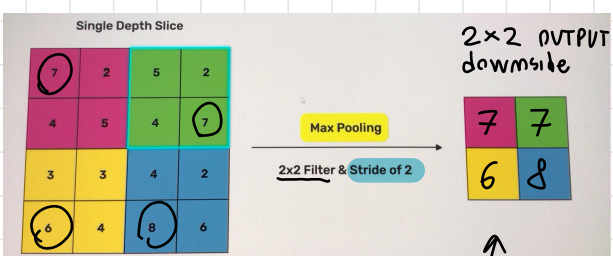
pooling layer reduce the # of parameters in the network by reducing input size of following layers ← It is used to summarize the features of an activation map

In this way the TRAINING computation requirements are reduced + mitigate OVERFITTING

Pooling is a form of DOWNSIZING that uses 2D sliding filters passing over input slice according to the configurable STRIDE parameter

↑

number of pixels that filter moves from one position to the next between pooling



4x4 INPUT SLICE

max value over the Filter slide (Receptive Field) is written in the output

We have two types of pooling

↙ ↘

MAX POOLING (most common)

AVERAGE POOLING

while conv. filters has trainable parameters

Max Pooling Filters has NO trainable parameters

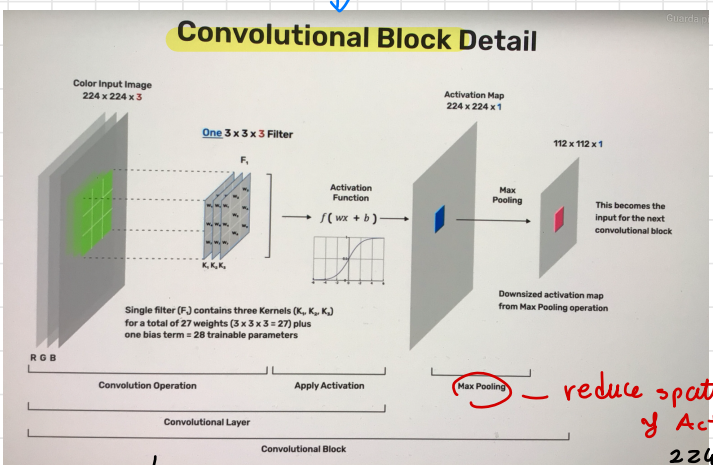
↓

the filter specify the size of the window to use for max operation (does NOT involve weights)

- With all the elements explained

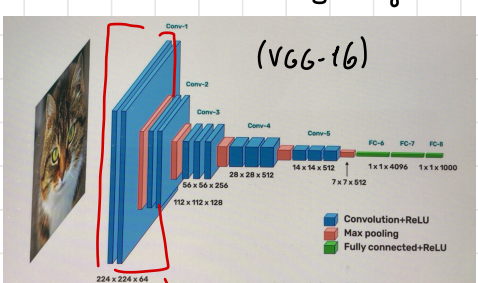
↓

example:
one conv. layer with one Filter



reduce spatial dimension of Activation map
224x224 → 112x112

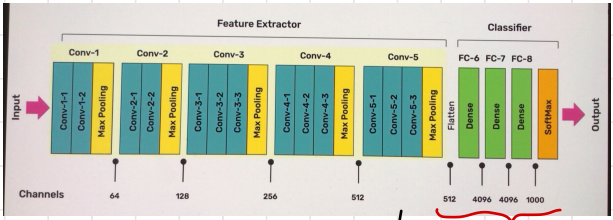
↓ in reality, before a Max Pooling layer there are two / three conv. layers



conv. block

With conv. layers containing at least 32 filters

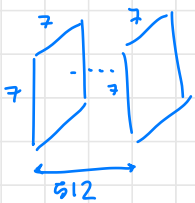
so what is the overall transition from the INPUT to the output of a CNN?



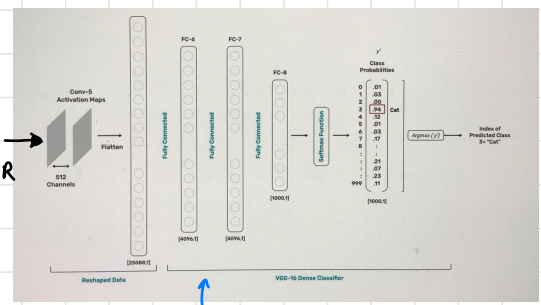
the output from the east conv. block contains a series of Activation maps

↑ in the CLASSIFIER, the fully connected dense layers transform extracted features into class probabilities

im VGG-16, it is 7x7x512 channels



FEATURE EXTRACTOR →



before data from feature extractor are processed by classifier, they are Flatten 7x7x512 1D vector

↑ than flattened data is processed by fully connected layers

Fully connected layers are needed in classification tasks!

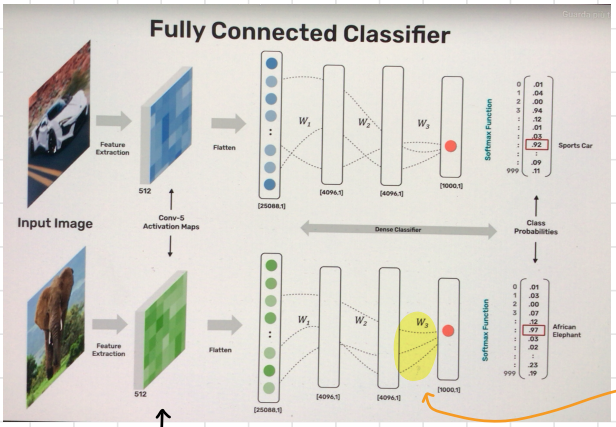
because the Activation maps from a final conv. layer of a trained CNN represent meaningfull information about an image contents

Fully connected classifier layers allow it to consider informations from the entire image!

each spatial location in a feature map is related to original image

the flattening doesn't alter the spatial interpetation of the data! (just repack data for processing purpose)

Example



the weights are fixed at the end of the training

FEATURE EXTRACTOR: has learnt to detect lots of features in the training dataset

↓

CLASSIFIER then learns the final activation

this dotted lines visualize pathways leading to highest activation in output neurons (high weights with high activation)

↑ the resulting feature map of last layer will be different

⇓

for larger problems, it is hard to visualize...

the high level concept is the usual weights training of network to minimize loss function in order to learn meaning full features + fire appropriate neuron in output layers

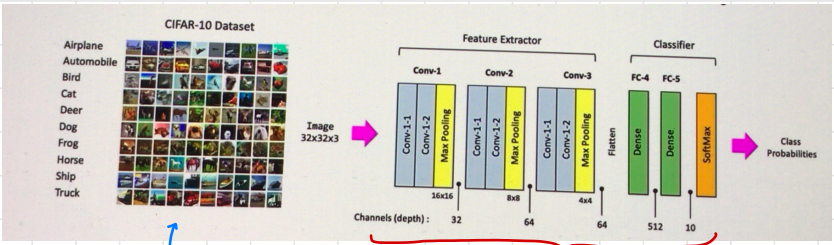
Summary of Keypoints

* CNNs designed for a classification task contain an upstream feature extractor and a downstream classifier.
* The feature extractor comprises convolutional blocks with a similar structure composed of one or more convolutional layers followed by a max pooling layer.
* The convolutional layers extract features from the previous layer and store the results in activation maps.
* The number of filters in a convolutional layer is a design choice in the architecture of a model, but the number of kernels within a filter is dictated by the depth of the input tensor.
* The depth of the output from a convolutional layer (the number of activation maps) is dictated by the number of filters in the layer.
* Pooling layers are often used at the end of a convolutional block to downsize the activation maps. This reduces the total number of trainable parameters in the network and, therefore, the training time required. Additionally, this also helps mitigate overfitting.
* The classifier portion of the network transforms the extracted features into class probabilities using one or more densely connected layers.
* When the number of classes is more than two, a SoftMax layer is used to normalize the raw output from the classifier to the range [0,1]. These normalized values can be interpreted as the probability that the input image corresponds to the class label for each output neuron.

CIFAR-10 CLASSIFICATION USING CNN

FROM SCRATCH !

IMPLEMENTING A CNN IN TENSORFLOW AND KERAS



Training Dataset
CIFAR-10

Simple CNN Architecture we well implement from scratch

+ we well introduce the DROPOUT layer often used to mitigate overfitting

1) Load CIFAR-10 dataset + display some samples
2) PRE-PROCESS the dataset
3) set-up Data / Training configuration parameters
4) Model structure implementation (CNN)

because the model overfit training data

5) Add DROPOUT layers to mitigate overfitting → better generalize new data + improving accuracy on validation

Training again the model

6) HOW to SAVE the model trained to file system + HOW to load back in memory to use it

7) How to evaluate model accuracy on test dataset
- use individual samples to make prediction
- confusion matrix

1) cifar10.load_data() → contains 10 classes small colored images of size 32×32×3 (RGB) overall 60000 data
↑
included in Tensorflow distribution

↳ Inspect the images (32×32) small, NOT lot of details BUT still enough information to support classification task

2) Normalize image data on 0÷1 range → it helps training the model + convert labeled images in One-Hot Encoded Labels to_categorical

3) We create simple classes to create training and Data configuration parameters

```
@dataclass(frozen=True)
class DatasetConfig:
    NUM_CLASSES: int = 10
    IMG_HEIGHT: int = 32
    .....
```

← this allow to organize parameters

this constrains instances to NOT being changed! NOT intended to being modified

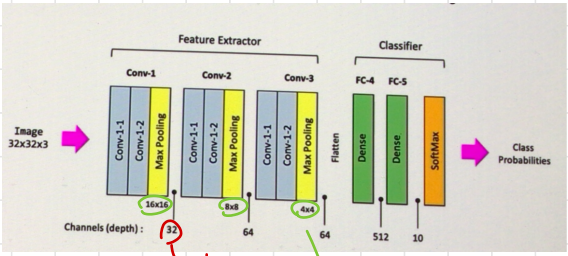
dataclass provides decorator to generate user define classes ← @dataclass is a decorator used to add special methods to classes (automatically create required methods)

this automatically implement __init__, __repr__ and __eq__ methods

4) to model the CNN, as done in the MLP NN
1. build the network with pre defined Keras layers
2. compile the model .compile(...)
3. train the model .fit(...)

We consider a small CNN

with



channels

spatial size of Activation map

↓

We define the network model in a function for convenience

1. Instantiate NN model `model = sequential()` ← this allow to create the model sequentially adding one layer at a time

2. Add the layers `model.add(...)`

`(Conv2D(filters, kernel_size, padding, activation, input_shape))`

convolutional layer

`'same'` ↓ this pad input tensor so that output has same size of input (if NOT defined, smaller output spatial size ... NO padding by default)

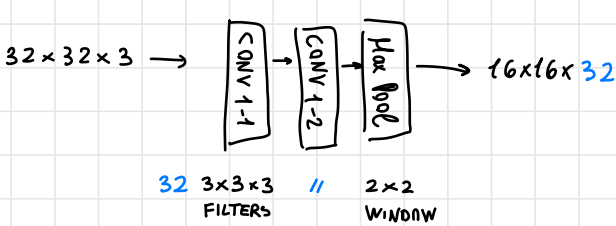
`'relu'` ↑ used in all except for last layer

to specify only for ↓ first layer! in the following layers is automatically computed

...

`(MaxPooling2D(pool_size = (2,2)))`

↑ window size of pooling



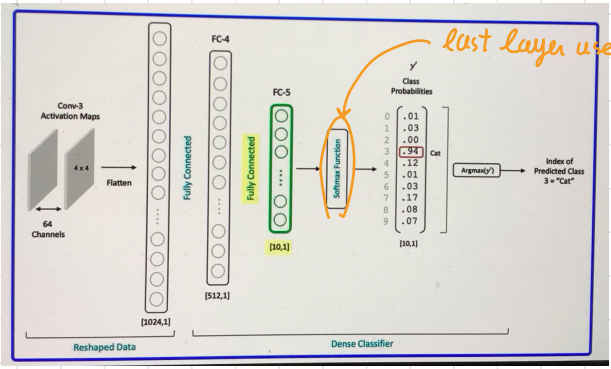
... in the same way you define the other conv. blocks

3. before defining the Fully connected (Dense) layers of classifier ...

↓

the output of conv part has to be flattened 4x4x64 → flatten → 1D 1024 vector

this 1D vector is computed in 2 hidden layers + output layer



last layer uses softmax as activation function to generate a probability distribution → it normalize the data in [0,1]

it is an ACTIVATION function but act differently than ReLU (which apply on every neurons)

4. create the model

↓ than visualize

model.summary() → visualize size/shape ∀ layer in the CNN Architecture

+ total # parameters to train ~ 669 354

small numbers... VGG-16 has $138 \cdot 10^6$ params

5. compile model

model.compile(optimizer, loss, metrics)

'rsm prop' gradient descent

'categorical_crossentropy' CROSSENTROPY as standard for classification

6. train model

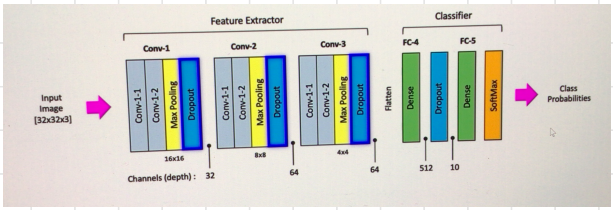
model.train(...)

↓

X_train, y_train, batch_size, epochs, verbose, validation_split (0.3 = use 30%)

↓

it takes last 30%! IF the dataset is ORDERED, we need to randomize first...



model.add (Dropout (0.25))

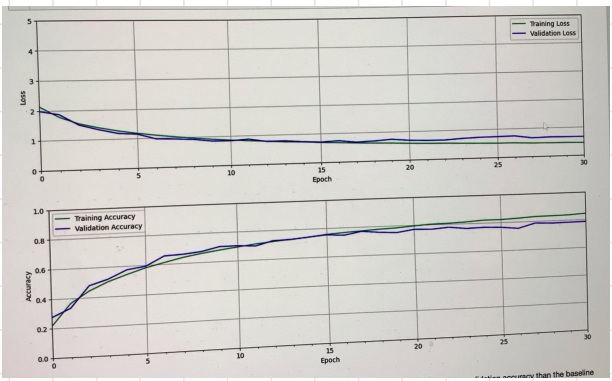
↑

percentage of neurons to randomly drop from previous layer during training

With more Dropout layers and fraction parameter, the more the effect is pronounced... we need a good trade-off

this choice is based on best-practice + your experience

... the remaining steps are equal



but NOW the Validation is NO more getting WORSE! NO more overfitting

↓

NOW Validation Accuracy is satisfactory ~ 80%

6) SAVE/LOAD models → how to save a model and use it later on

model.save (<name>) ← save in filesystem, creating a folder with the proper architecture + training configuration are saved in a .pb (proto buff format)

+ a checkpoint training file including model weights

models.load_model (<model>)

↑ this can now be used

7) EVALUATE

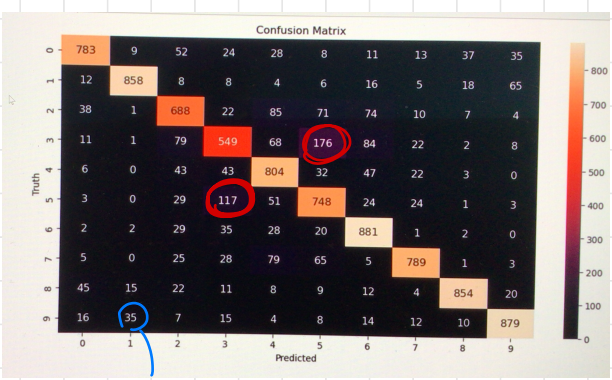
1. we can compute accuracy on test dataset
2. visually inspect results on a subset
3. plot confusion matrix

1. `model.evaluate(X_test, y_test)`
   ↳ return loss and accuracy on test dataset

2. use `model.predict(...)` to make prediction on a subset of the test data
   ↑
   visual inspection of the classification abilities ... reveal dataset problems, possible corruption OR outliers

3. use `confusion_matrix(...)` on the test dataset,
   common metric to access classification accuracy
   very revealing table representation on ground truth - prediction

   `tf.math.confusion_matrix(labels, predictions)`



{ 3 := CAT
{ 5 := DOG

↵ often confused!
this 2 classes are often misclassified...

When y = CAT, 176 times DOG
y = DOG, 117 times CAT
↑
MISSCLASSIFICATION

TRUCKS (9) often confused with AUTOMOBILES (1)

↓
→ this make sense due to similarities of the classes!

TRICKY. QUESTION... study better how convolutional operations and padding are performed!

Question 2

1/1 point (graded)

```
model = tf.keras.Sequential()
model.add(Conv2D(64, kernel_size=5, padding='valid', activation='relu'))
model.add(Conv2D(32, kernel_size=3, padding='same', activation='relu'))
model.add(MaxPooling2D(pool_size=(2, 2)))
```

What is the shape of the output tensor when an tensor of shape of (2, 15, 15, 3) is passed as input to the model above?

- (2, 5, 5, 32) ✔
- (2, 4, 4, 32)
- (2, 6, 6, 32)
- (2, 5, 5, 64)

IMAGE CLASSIFICATION USING PRE-TRAINED ImageNet MODELS

Pre-trained models to perform image classification

↓

What to do when a big number of object types are involved in the classification problem? many classes → requires large networks with millions of parameters

↓

thanks to ImageNet project pre-trained CNN models are available in Keras... trained to detect objects from 1000 classes!

⇒ how to use this models without training requirements?

- ImageNet and ILSVRC

↓

large visual database designed to support research in object recognition ⇒ this project give birth to ImageNet Large Scale Visual Recognition Challenge annual competition where researcher evaluate theirs models for recognition tasks

thanks to this contribution KERAS has built in pre-trained ImageNet models ≃ 19 different pre-trained model available ←

↓

how to use those pre trained models: (4 steps)

1. load pre-trained model
2. pre-process input images
3. use the predict method for inference
4. de-code predictions by post-process

⇓

Let's look at the Keras API to use those models

ResNet50() as example model

many parameters guarantee flexibility of usage

1) INSTANTIATE the model

```
model_resnet50 = tf.keras.applications.resnet50.ResNet50(include_top=True,
                                                         weights='imagenet',
                                                         input_tensor=None,
                                                         input_shape=None,
                                                         pooling=None,
                                                         classes=1000,
                                                         classifier_activation='softmax',
                                                        )
```

When the classes you are interested in are part of the ImageNet dataset you can use this model directly in your application

include_top := tells if using the model trained classifier

weights := which weight to use 'imagenet' ← use the weight you obtain on imageNet dataset

...

↑

It is possible to leverage pre-trained models in specialized classifier with different settings

2) PRE-PROCESS

- resizing images → to be conform to the expected network input

↓

Keras models provide dedicated preprocess_input() function

3) PREDICTION ← implementation detail: usually input are in form [b, h, w, c] ↑ batch dimension

in DL framework, image data are usually already packed properly, we need a batch dimension to ← 4 rank tensor our images to be processed by the model ⇝ reshape our data to conform batch

ResNet50 expect RGB 224×224×3, with a batch dimension in the first axis! (you need it to process it) [batch, 224, 224, 3]

expected when processing image

the final prediction contain array of class prob

4) DE-CODE with an available function model specific or general (same API) → return the top 5 predictions by default, easy to use and parse results

FIRST we visualize our classes to validate

We test:
- VGG-16 (B, 224, 224, 3)
- ResNet50 (B, 224, 224, 3)
- InceptionV3 (B, 229, 229, 3)

← all take RGB image, with first axis "None" representing a placeholder for batch dimension

We define a convenience function to process our images → to prepare data and predict

the CNN model used

paths to the images in filesys

special size of img required

```
def process_images(model, image_paths, size, preprocess_input,
display_top_k=False, top_k=2):

    plt.figure(figsize=(20,7))
    for idx, image_path in enumerate(image_paths):

        # Read the image using TensorFlow.
        tf_image = tf.io.read_file(image_path)

        # Decode the above `tf_image` from a Bytes string to a numeric Tensor.
        decoded_image = tf.image.decode_image(tf_image)

        # Resize the image to the spatial size required by the model.
        image_resized = tf.image.resize(decoded_image, size)

        # Add a batch dimension to the first axis (required).
        image_batch = tf.expand_dims(image_resized, axis=0)

        # Pre-process the input image.
        image_batch = preprocess_input(image_batch)

        # Forward pass through the model to make predictions.
        preds = model.predict(image_batch)

        # Decode (and rank the top-k) predictions.
        # Returns a list of tuples: (class ID, class description, probability)
        decoded_preds = tf.keras.applications.imagenet_utils.decode_predictions(
            preds=preds,
            top=5
        )
```

↑ reference to preprocessing function to that model

read contents byte

decode into numeric

additional optional argument to flexible output

↑ resize image to the expected size for the model input

↑ add batch dimension as first axis!

retrieve list of top k predictions

... then display

VGG-16 make some error and low confidence on some classes with bad accuracy on some class and missclassification

ResNet50 have better predictions on many classes with 99% confidence

InceptionV3 correctly classify also the images!

↑ because 1000 classes are part of the model, it is reasonable that a 99% confidence is NOT always possible... there are very similar classes!

⇒ it is possible to customize this PRE-TRAINED models for your application!

IF in your project you require classes NOT in ImageNet you can start from pre-trained ImageNet model and use your own dataset to re-purpose the model for NEW application

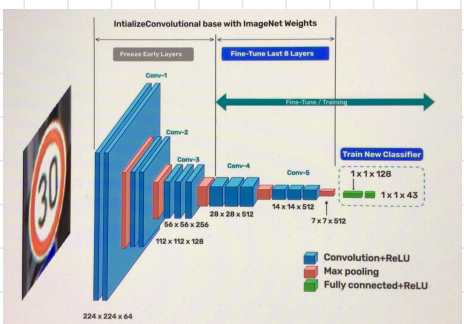
↓

Leveraging pre-trained models... using techniques called TRANSFER LEARNING and FINE TUNING

from a pre-trained model, a portion of it is re-trained with new image classes!

⇓

TRANSFER LEARNING and FINE-TUNING



here we focus on the Fine-tuning technique (most flexible for transfer learning) but discuss other techniques too

↓

leveraging pre-trained models is extremely helpfull when you don't have a big dataset to train your model OR you have limited resources...

↓

Repurpose pre-trained models for new applications!

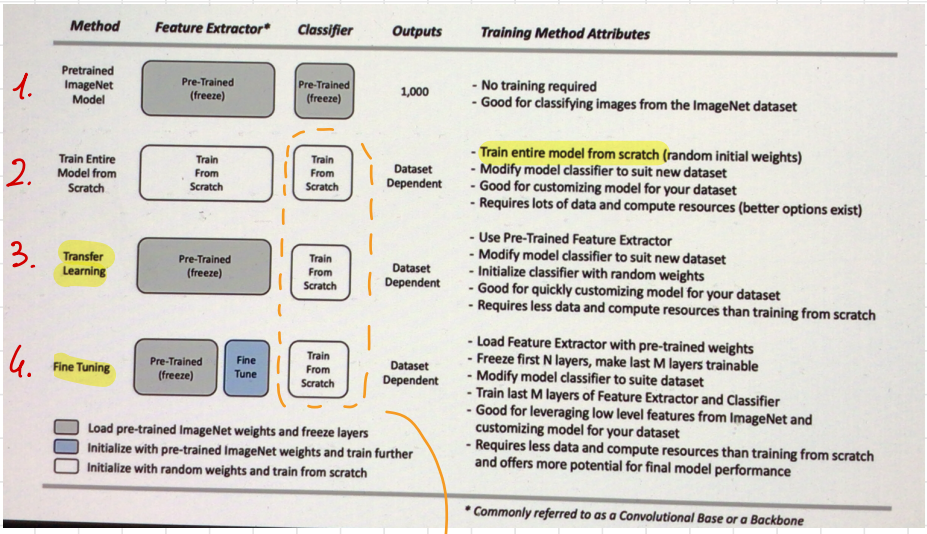
here we see how to apply Fine tuning on VGG-16 to repurpose on new classes

- we use the German traffic sign recognition benchmark dataset of 43 classes, with 50k images
  - ↳ we will reduce to a portion of 40·43 images (further split in (28·43 for training + 12·43 for validation)

to show the potential of transfer learning for small datasets ←

Then the evaluation will be done on the original 12k test dataset

Options when working with pre-trained models...



1. Just use the model as it is (out-of the box) ⇑

+ three additional options (cases) when you have custom dataset you want to repurpose the model for 2, 3, 4

↓

in each of this scenarios, a new classifier MUST be trained / defined consistent with custom dataset

the classifier pre-trained is tailored for ImageNet... it cannot be used as it is for new dataset → this must be redefined for new task

2. train-from scratch → load the model and train again, all weights are re-initialized with new random weights ← this requires many data (big dataset) and computational resources

+ consider including dropout layer on classifier to avoid expected OVERFITTING due to the small dimension of the dataset

still

validation loss >> training loss ← the model is OVERFITTING

for the model its challenging to acquire sufficient knowledge on such a small dataset!

NOTICE that initial loss ~ close to random chance

IF we have 43 class Prob = 1/43 ⇒ cross-entropy loss = $-\log\left(\frac{1}{43}\right) \simeq 3.76$

In fact initially we have ~ 3.7 initial training loss

3. TRANSFER LEARNING

We load pre-trained VGG-16 model feature extractor

+ re define the classifier for the dataset

↓

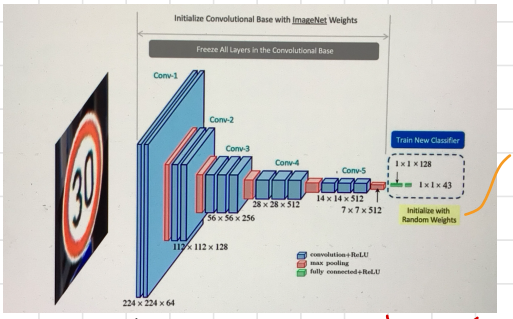
We need to load from Keras API only the model CONVOLUTIONAL Base pre-loaded with its weight

+ define our own classifier consistent with the dataset ... so having #outputs = # classes

for example use only one FC-1 + output layer

(128 neurons) (43 neurons)

↑ design choice, NOT related to original classifier



loaded with random weights ...

during training we allow only to the classifier weights to be updated

↓ this make sense because pre-trained ImageNet feature extractor has learned valuable features to detect various usefull object types... we presume those features are general enough to be applied to other dataset

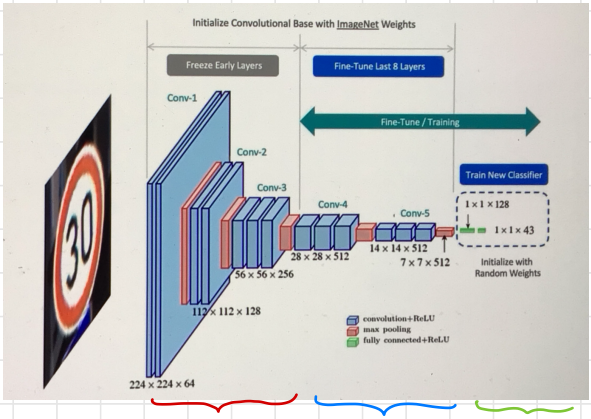
↓

therefore we nee to retrain only the classifier portion... to learn a new mapping from features to classes ← FASTER and MORE EFFICIENT training process

But sometimes, Retraining only the classifier is NOT enough ☹

and Fine Tuning may be a better option: more flexible alternatives to transfer learning

4. FINE TUNING



similar to transfer learning ...
A new classifier is defined BUT NOT all the feature extractor convolutional blocks are locked!

we freeze only initial layers of convolutional base AND we allow last layers to be further trained ...

TRAINABLE convolutional base portion (last layers)

NEW classifier to TRAIN

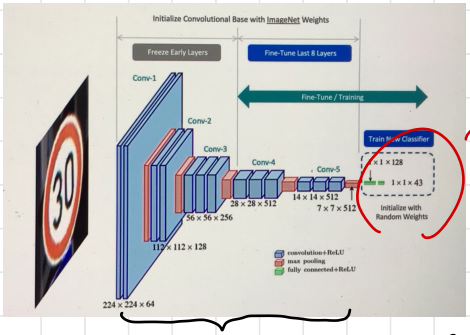
Fine-tune / train

the underline idea is that initial layers of feature extractor capture generic low level features (edges / corners / arcs) which are building blocks in any classifier. task

While the last layers (deeper) ← of feature extractor build upon low-level features to learn more complex representations related to the specific custom dataset

↓

Powerfull technique! Allow to adjust the pre-trained model to be more relevant to our specific use case!



While classifier uses small initial random values ... as learned from scratch features → classes mapping

when TRAINING, the convolutional layers are initialized with the pre-trained ImageNet weights. than the last layers are FINE-TUNED for our application (refined!) to learn specific feature

because the feature-extraction is fine tuned to improve model performance!

NOTICE that Option 2 (SCRATCH RE-TRAIN) with small dataset will clearly limit the performances! → While by Fine-tuning we leverage many features from ImageNet, starting from initial good state

Now we focus on FINE TUNING IMPLEMENTATION

While include transfer-learning + convolutional base layers selection to fine-tune

+ pre-processing testing dataset is important, because it is in a different format from training and validation dataset

NEW techniques to manage image data differently organized in the Filesystem...

1. Download custom Dataset → Notice that the images comes from video sequence... many images are adjacent frame in video... Therefore to resize the dataset (smaller) we need to sample with a certain step size to maintain diversity

2. Initialize usefull parameters such as paths strings and shapes of dataset + training parameters

taken similarly to the scratch training exampee.. by experience design

Train, Valid, Test folders are separated (with Train, valid having subfolders ∀ class)

and the labels for Test Images are located in a separate csv file (further load / parse of information is required)

-/ Train
- /class 1
- /class 2
- /class N

-/Valid
- /class 1
- /class 2
- /class N

-/Test
Test.csv

3. Create train an validation dataset objects

the load_data() method is used in keras to load images and labels into numpy array

↑ which is fine for small dataset ...

BUT in general it is more effecent to create a dataset object much more effecent when handle data in large dataset

- KERAS method convenient to create the dataset object from directory

```
train_dataset = image_dataset_from_directory(directory=DatasetConfig.DATA_ROOT_TRAIN,
                                             batch_size=TrainingConfig.BATCH_SIZE,
                                             shuffle=True,
                                             seed=SEED_VALUE,
                                             label_mode='int', # Use integer encoding
                                             image_size=(DatasetConfig.IMG_HEIGHT, DatasetConfig.IMG_WIDTH),
                                            )

valid_dataset = image_dataset_from_directory(directory=DatasetConfig.DATA_ROOT_VALID,
                                             batch_size=TrainingConfig.BATCH_SIZE,
                                             shuffle=True,
                                             seed=SEED_VALUE,
                                             label_mode='int', # Use integer encoding
                                             image_size=(DatasetConfig.IMG_HEIGHT, DatasetConfig.IMG_WIDTH),
                                            )
```

this expect file structure as

main_directory/
- class_a/
  - a_image_1.png
  - a_image_2.png
  - ...
- class_b/
  - b_image_1.png
  - ...

By default the class names in dataset will be inferred by the folders names

this function require the top-level-folders + optional arguments for further configure dataset

```
train_dataset = image_dataset_from_directory(directory=DatasetConfig.DATA_ROOT_TRAIN,
                                             batch_size=TrainingConfig.BATCH_SIZE,
                                             shuffle=True,
                                             seed=SEED_VALUE,
                                             label_mode='int', # Use integer encoding
                                             image_size=(DatasetConfig.IMG_HEIGHT, DatasetConfig.IMG_WIDTH),
                                            )
```

default image_size is 256x256 so you need to reset for your CNN Architecture

class names are then accessible, initialized as digits

this is NOT a problem for TRAIN / VALIDATION dataset but it influence the TEST dataset creation to have consistent labels... this influence when we create test dataset later on...

It is usefull to print some samples...

We can access image, label batch using the .take (1) method ↑ FIRST batch of data

(the dataset is very diverse!)

4. Create TEST dataset, which is organized with all images 512K in a folder Test and labels in a Test.csv tabular file...

↓

to assemble all in a custom Test dataset we need to

1. Retrieve class labels from csv, save in a list
2. build list of image file paths
3. combine image path and label to create a tf.data.Dataset object
4. use .map() method to load and pre-process image

4.1) We use pandas dataframe to read class ID and store on a list

BUT, in the train and validation Dataset, the class names have been ordered alphabetically rather than numerically!

↓

So to have consistent test dataset labels, we need to create a mapping to New ID

(an alternative solution was to use the class_names optional argument in STEP 3 when calling image_dataset_from_directory(...))
If we had supplied numerical order class list there would be no need for this mapping

for example the mapping is created with dictionary, then remap all IDs

+ Now we have a list of IDs of labels

↓

4.2) How to create images paths list ⇒ by glob python utility

4.3) Finally we can combine (paths, IDs) in a dataset object using from_tensor_slices method

4.4) define functions to load images into test dataset + pre processing

↳ we use decode_png to convert in uint8 and resize in expected size image

↓

applied to the dataset using the .map() method, so the dataset contain pre-processed images + ground truth

- we can finally display some Image in TEST dataset

5. modelmy VGG-16 for fine-tunimy

```
tf.keras.applications.vgg16.VGG16(include_top=True,
                                  weights='imagenet',
                                  input_tensor=None,
                                  input_shape=None,
                                  pooling=None,
                                  classes=1000,
                                  classifier_activation='softmax',
                                 )
```

we will set it to False, to instantiate ONLY the convolutional base

```
# Specify the model input shape.
input_shape = (DatasetConfig.IMG_HEIGHT, DatasetConfig.IMG_WIDTH, DatasetConfig.CHANNELS)

print('Loading model with ImageNet weights...')
vgg16_conv_base = tf.keras.applications.vgg16.VGG16(input_shape=input_shape,
                                                    include_top=False, # We will supply our own top.
                                                    weights='imagenet',
                                                   )

print(vgg16_conv_base.summary())
```

We will NOT include the classifier! only conv base

+ we need to further configure the model for fine tunimy

(freeze initial layers of conv base)

here we will fine tune the last 4 layers, keepimy the others fixed

we set all layers trainable

+ fix only the first to False as trainable attribute...

⇓

we see that the trainable parameters are reduced

6. construct the model

```
inputs = tf.keras.Input(shape=input_shape)

x = tf.keras.applications.vgg16.preprocess_input(inputs)

x = vgg16_conv_base(x)

# Flatten the output from the convolutional base.
x = layers.Flatten()(x)

# Add the classifier.
x = layers.Dense(128, activation='relu')(x)
x = layers.Dropout(TrainingConfig.DROPOUT)(x)

# Output layer.
outputs = layers.Dense(DatasetConfig.NUM_CLASSES, activation='softmax')(x)

# The final model.
model_vgg16_finetune = keras.Model(inputs, outputs)

print(model_vgg16_finetune.summary())
```

1ST layer as pre_process layer! — do this to use that weights initially

add the first layers above (conv)

flatten layer

add Dense + dropout layer

assemble entire model with IN/OUT layers...

We build the model using KERAS API differently from Sequential API!

7. compile + Train

- remember to chose the appropriate loss function { sparse: integer encoding; categorical: one-hot encoding }
- the from_logits attribute specify to False when softmax activation in the output ... When NO softmax, setting from_logits = True, keras add softmax automatically

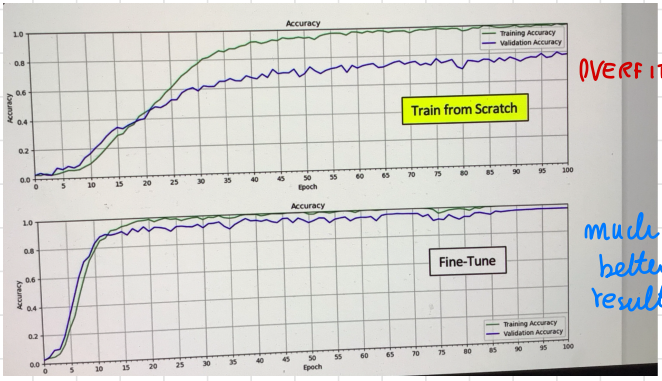
finally
Use the train() method with usual parameters

Clearly the results are very good! With train accuracy ~ 100% and also validation ~ 90%

⇓

Very good results in such small dataset

techniques such as DATA AUGMENTATION might be used to improve further accuracy



OVERFIT

much better results! → train more efficiently also using Data Augmentation

8. EVALUATE on Testing dataset obtaining ~ 97% Accuracy → GOOD generalization!

we can display some prediction compared to ground truth which are very good! With only some missclassification

moreover, again the CONFUSION matrix can be used to visualize results!

SEMANTIC SEGMENTATION (INTRODUCTION)

IMAGE SEGMENTATION USING TENSORFLOW HUB ← it contains many models!

by pre-trained semantic segmentation models

⇓

using pre-trained segmentation model, trained on CamVid dataset, which contains driving scene videos

In TensorFlow Hub: there are tons of problem and model solutions

We use

HRNet/camvid model → model specific page linked to example use case + access to the model

1. Download sample images
   ↳ build list of image path

   + read image in numpy array + expand dimension with BATCH + normalize the image

   ↓

   because the documentation explain to do so.. it depends on the model you are using

   ↳ load each image in memory and display... images from a camera on car with interesting part to segment!

+ create mapping (dictionary) from classID to names + RGB color value ∀class (help to visualize with a segmentation mask)

2. Use the model

using the model URL (taken from Tensor Flow Hub)

2.1) than use hub.load(·) to load the model from URL

2.2) we can directly call the predict() method passing the image which return predicted segmentation mask with dimension (batch, (special dim), classes + background)

regions NOT in any semantic class

each channel in the mask is a specific semantic class, with each of it representing prob that a given pixel is in that class!

⇓

it is a RAW prediction data that we need to post-process

2.3) post-process (1, 720, 960, 33)

- convert to numpy array
- remove background class when NOT needed (we may use it to divide back)
- remove batch dimension, which we don't need for post-process

(720, 960, 32)

each channel has prob ∀pixel

- we can print the mask channels for each semantic class
  each pixel has 0÷1 values that we can map as GRAYSCALE (scaled by matplotlib automatically)

⇓

than every pixel will have an assigned class as the max probability collapsing all channels in a single one with final segm.

→ it is possible to do it with a single command!

We get the argmax, obtaining index with max prob channel, we get single channel containing ID with highest prob
MASK 1-32 containing all the classes

- we can overimpose mask with image

Now we want to turn in RGB color image to determine class
↓
converting gray scale segmentation mask into color segment mask using a support function which uses our map dictionary

We can use it to segment with RGB color the scene!

3. we can formalize all to process in many image in simple way

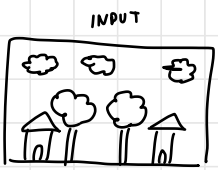
3.1. image_overlay() to overlay color map on top + transparency factor to better visualize original image

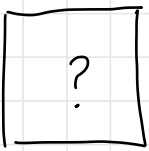
3.2. run_inference() logic to predict and post-process
↑
convenience function + define color legent to better interpret the result

- there are metrics to evaluate how well a segmentation model is behaving

Comments on SEGMENTATION:

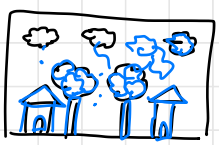
there are different types of segmentation tasks... in general SEGMENTATION has the objective of dividing the image in a set of pixels





it depends on the specific segmentation objective

IMAGE SEGMENTATION



divide in "SUPERPIXELS", grouping based on color/texture and other low level primitives (NOT very INFORMATIVE)

SEMANTIC SEGMENTATION

create a mask where a class is associated to each pixel

↓

assign label to every pixel in the image

(Deeplab V3)

INSTANCE SEGMENTATION

create a mask or contour differentiating each instance of different objects, without classifing all image pixels

(Mask R-CNN)

PANOPTIC SEGMENTATION (INSTANCE + SEMANTIC)

combine instance + semantic to classify all pixels and differentiating the instances of separate objects

(UPSNet)

↑ to label every pixel

Image segmentation has the objective of dividing the image in different regions

↑ can to differentiate class instances of same type of objects

↑ both belong to same class but are different instance

↓ it doesn't label all the image pixels, BUT the boundaries of specific class instances...

In semantic, every pixel has a class label

In instance, like object detection but instead of bounding box we find a mask of exact pixels, panoptic combine semantic + instance

OBJECT DETECTION (INTRODUCTION)

OBJECT DETECTION USING TensorFlow Hub

using pre-trained tensorFlow model shared in Tensor Flow Hub (central repository for sharing a variety of machine learning models provided by Google)

object detection involves identification and localization of objects in an image frame

↓

typically identified by color coded bounding box indicating the region where the object has been detected + label with class and confidence score

here the efficientdet/d4 model is used (you can copy URL or download the model properly)

↓

- this require INPUT with batch dimension (but don't accept batching)
- and wants a 0÷255 RGB image
- OUTPUT is a dictionary containing detection items (typical output of such models)

1. Download some sample images to test detection model
   - load images
   - add batch dimension
   - visualize images it has different object types of COCO Dataset

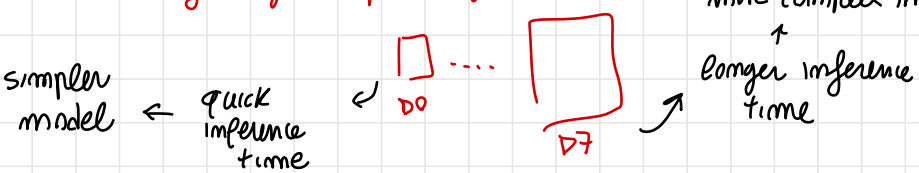
- define dictionary that maps ClassID to ClassNames in COCO dataset

+ Color code the detection with random colors to render bounding box of each object types! (we generate eight colors to be readible)

2. Load tensorFlow model We use the family of Efficient Det models

↓

designed with efficency in mind! exists D0 ÷ D7 models (8 tot) each one taking longer input image size



here we use D4 (medium model)
- specify model URL
- load simply in memory in our od_model object

↓

than easy to use for inference

3. perform inference

(NO predict(.)! → we just pass the image as argument to model

model(img) ↳ returning the output in the form of a dictionary

the keys contains items related to detection (+number) and raw_detection

large number of generated detection

we also get the boxes of detections ← removed by post processing steps

↓

those results are finally used to annotate the image

- post-process and annotate the image with bound_box based on a detection threshold
  - we apply this threshold to the scores

↓

we extract only satisfactory score detection to avoid low/redundant detection that we should filter

Logic to annotate the image

- extract bounding-box, class id, score ∀ detection threshold
  - we take the coordinates of it to pixel coordinates, because model produces normalized coordinates (we map to pixel space)
  - create color list with our random colors list
    - ↓
  - finally cv.rectangle() is used to draw the rectangle

We conclude with annotating the class name and score

↓

and properly set rectangle background using text dimension (managing possible limit when box on border)

⇓

this can now be used to process and classify images...

- IF threshold = 0: we get all detected objects even if NOT very accurate
  1. use the model for inference
  2. process and get annotated image

  clearly NOT smart choice, very WRONG approximation...

↓

- with threshold = 0.3 only sure results are shown! ⇒ better accuracy (but sometimes redundancy can occur)

⇓

it is possible to further formalize the task by defining a function that process a list of images ↳ than post-process, removing the batch image when annotating