CNN
Practical advice

Use open-source implementation

Transfer learning

Data augmentation

Use open-source implementation

- CNN can be difficult or finicky to replicate
  - Tuning, hyperparameters, etc

- Look for open-source license implementation
  - Much faster
- Use GitHub (download / clone)
- The CNNs in open-source implementation are usually already trained (download the code and the weights)
  - Need lots of time and computer resources (multiple GPUs) for training
- Contribute back with your code

Transfer learning

Transfer learning is not a machine learning model or technique; it is rather a **design methodology**.

The general idea of transfer learning is to **use knowledge learned** from tasks for which **a lot of labelled data is available** in settings where **only little labelled data is available**.

Creating labelled data is expensive, so optimally leveraging existing datasets is key.

- To solve a **new problem**, we may use a pre-trained model of a **similar problem.**
- Instead of building a model from scratch to solve the new problem, we may use **the model trained on other similar problem as a starting point.**

- Rather than training the weights from scratch, from random initialization, we often make much faster progress if we download weights (from open source) that someone else has already trained on the network architecture and use that as pre-training and transfer that to our new task.
  - Sometimes these training can take several weeks and might take many GPUs.

[Andrew Ng, Transfer Learning, https://www.coursera.org/learn/convolutional-neural-networks/lecture/4THzO/transfer-learning]

Transfer learning is usually done for tasks where your dataset has too little data to train a full-scale model from scratch.

The most **common workflow** of transfer learning in the context of deep learning:

1. Take layers from a previously trained model.
2. **Freeze them**, to avoid destroying any of the information they contain during future training rounds.
3. Add some new, trainable layers on top of the frozen layers. They will learn to turn the old features into predictions on a new dataset.
4. Train the new layers on your dataset.

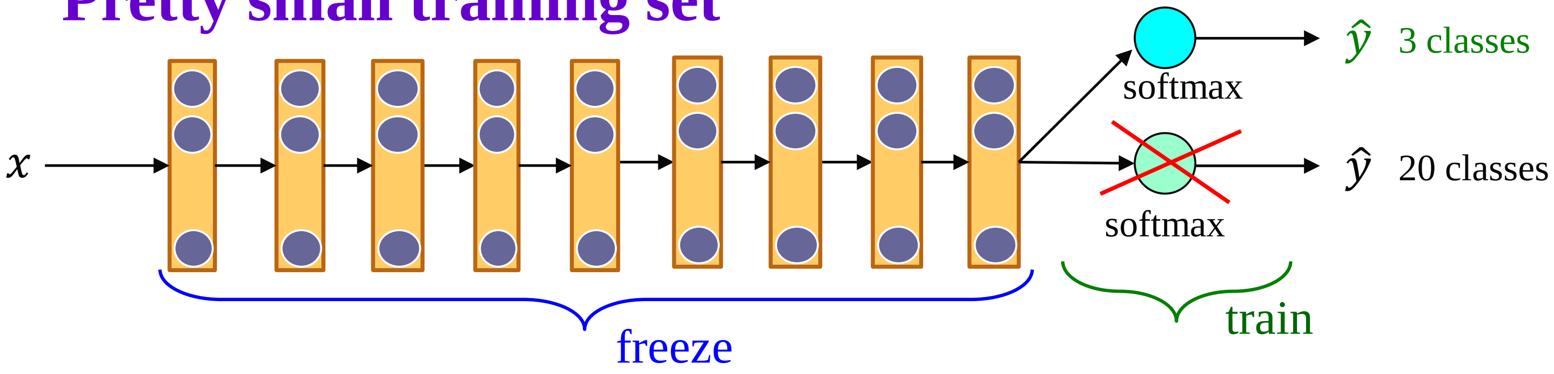
A last, optional step, is **fine-tuning**, which consists of:

1. Unfreezing the entire model you obtained above (or part of it)
2. Re-training it on the new data with a very low learning rate.

- This can potentially achieve meaningful improvements, by incrementally adapting the pretrained features to the new data.

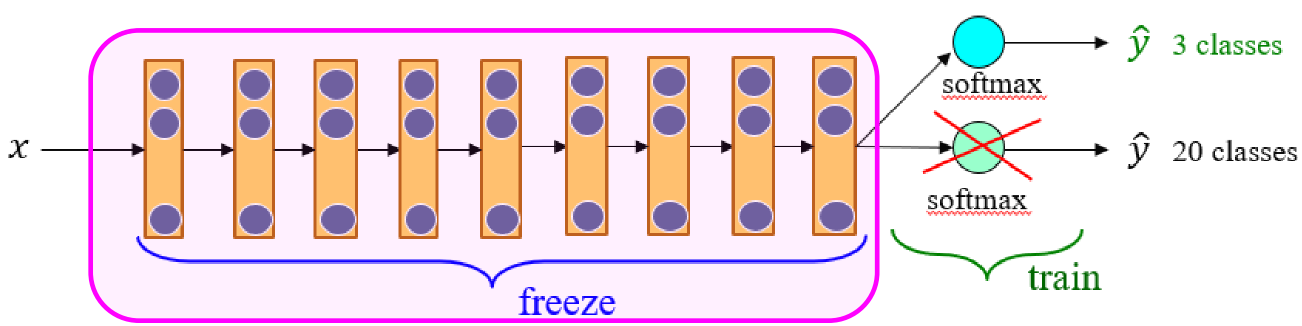
François Chollet, Transfer learning & fine-tuning, 2020/05/12, https://keras.io/guides/transfer_learning/

Pretty small training set

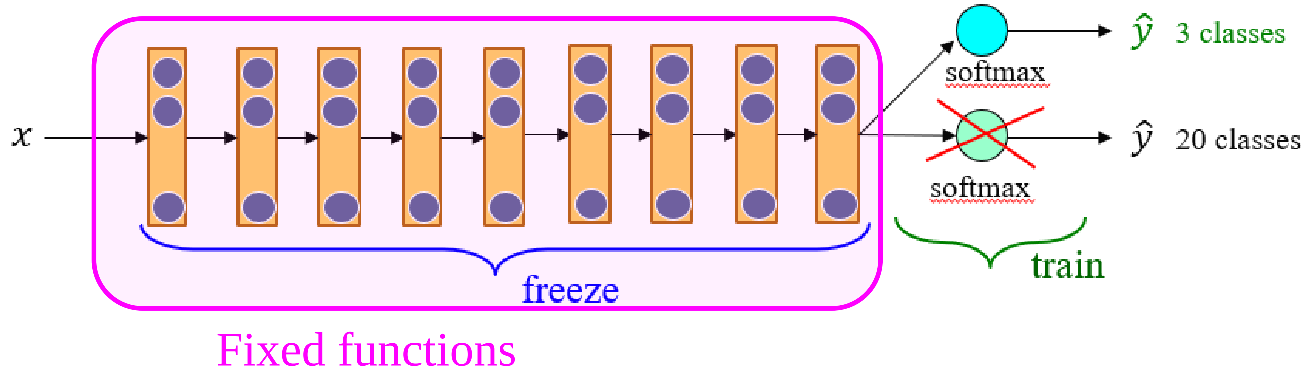


The top layers that are trained will use the pre-trained weight as initialization

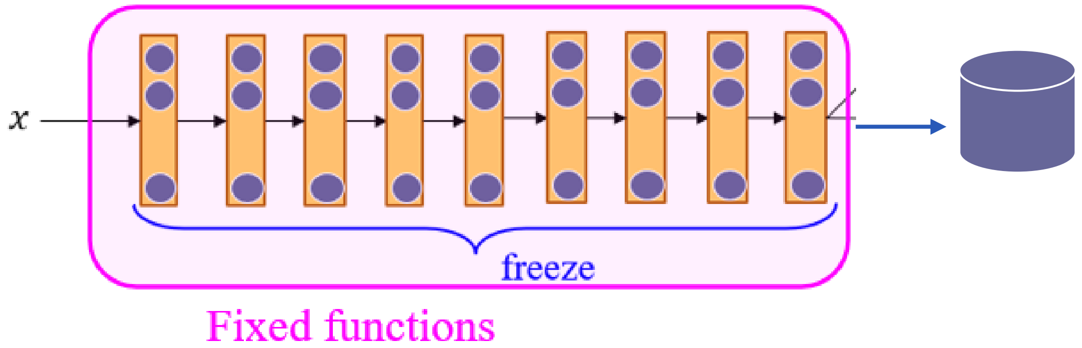
Because all early layers are frozen, there are some fixed functions that doesn't change.
You can take an input image *x* and map it to a set of activations in last frozen layer.
The trick that could speed up training is to just pre-compute the activations of that layer and save them to disk using that fixed function in the first part of the neural network.
Take as input any image *x* and compute the feature vector for it and then you will train a shallow softmax model from this feature vector to make a prediction.
Pre-compute that layer activation, for all the examples in training sets and save them to disk and then just train the softmax classifier right on top of that.
The advantage of the save to disk or to pre-compute method is that you don't need to recompute those activations every time you take a training epoch.



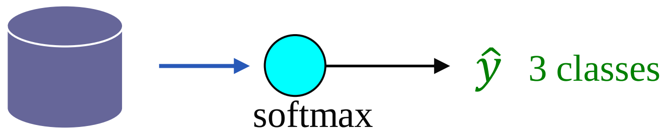
Fixed functions



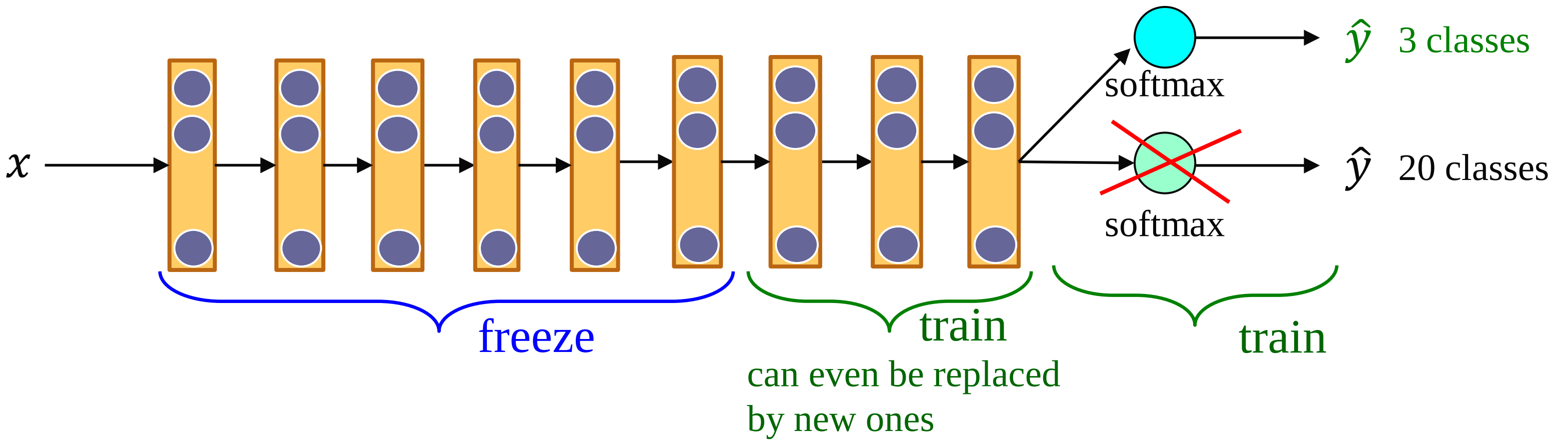
Pre-compute the feature vectors for the last layer, for all the examples in training sets, and save them to the disk (only once; forward propagation)



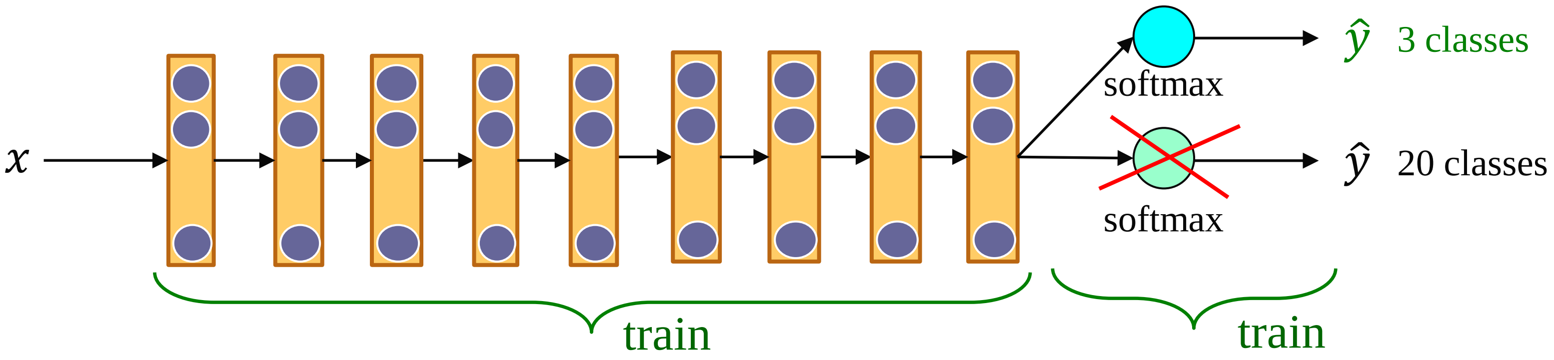
Then train only the softmax layer (shallow network)

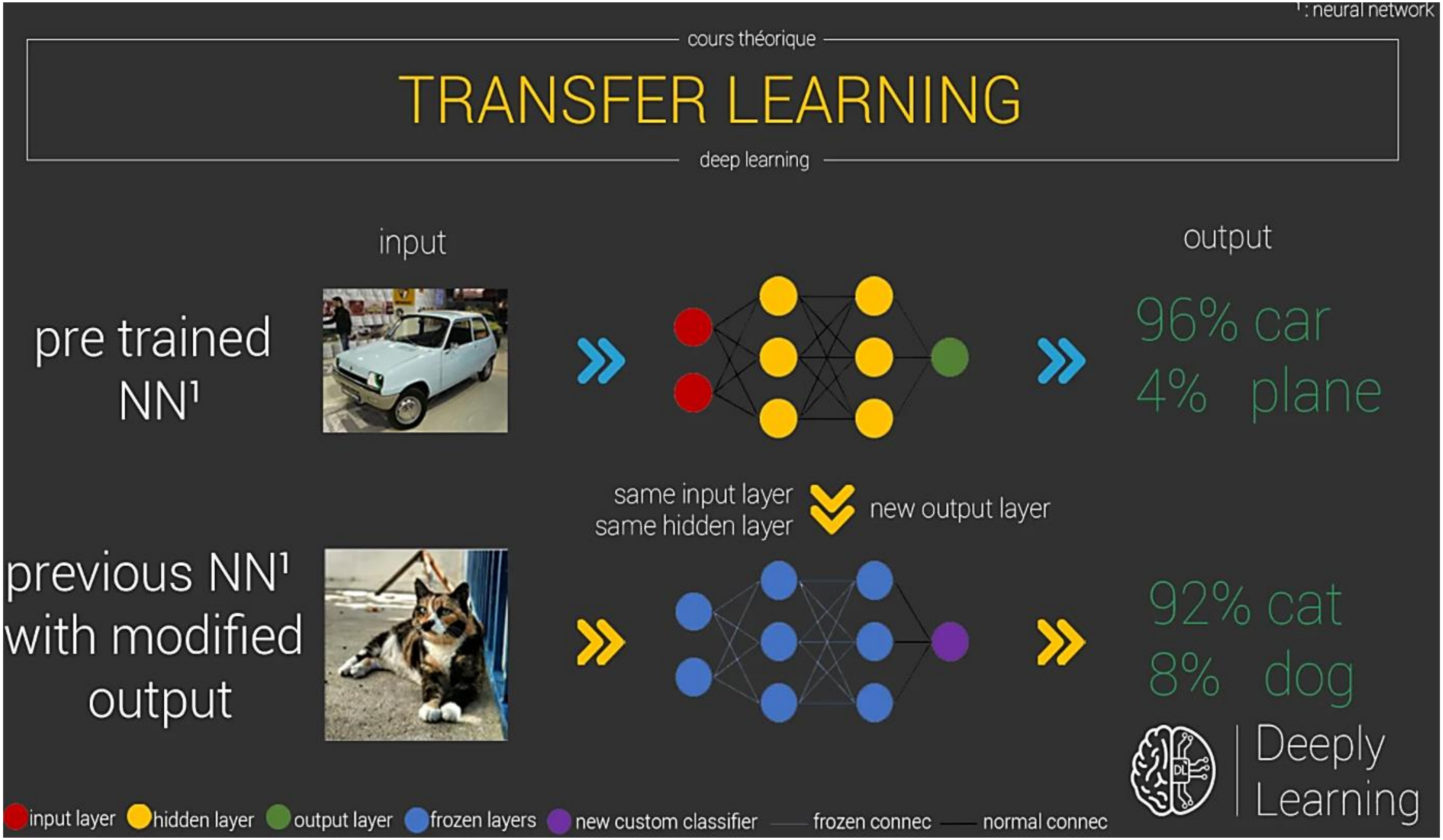


Larger training set



Very large training set





Bastien Maurice , Deeply Learning, 11 septembre 2018,
https://deeplylearning.fr/cours-theoriques-deep-learning/transfer-learning/

Freezing layers

```
# Make a model with 2 layers
layer1 = keras.layers.Dense(4, activation="relu", name = 'Dense1')
layer2 = keras.layers.Dense(2, activation="sigmoid", name = 'Dense2')
model = keras.Sequential([keras.Input(shape=(3,)), layer1, layer2])
```

```
# Freeze the first layer
layer1.trainable = False
```

```
inner_model = keras.Sequential(
        [
            keras.Input(shape=(3,)),
            keras.layers.Dense(4, activation="relu", name = 'Dense1_inner'),
            keras.layers.Dense(4, activation="relu", name = 'Dense2_inner'),
            keras.layers.Dense(4, activation="relu", name = 'Dense3_inner'),
        ],
name = 'Inner_model'
)
```

```
## Add one more layer
model = keras.Sequential(
    [keras.Input(shape=(3,)), inner_model, keras.layers.Dense(2, activation="sigmoid", name = 'Dense4')]
)
```

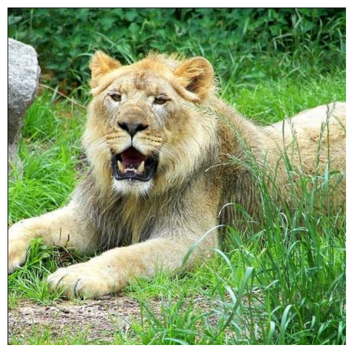
```
model.trainable = True

""" You can play around freezing some layers """
# inner_model.layers[0].trainable = False
inner_model.trainable = False
# model.layers[1].trainable = False
```

Data augmentation

Deep Learning sometimes may run into problem where data has limited size (e.g. overfitting).
To get better generalization in the model we need more data and as much variation possible in the data.
Sometimes, dataset is not big enough to capture enough variation, in such cases we need **to generate more data from given dataset**.
That is where **Data augmentation** can play a very important role.

Original image

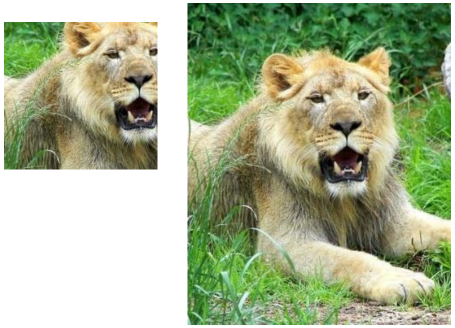
Mirroring (flip)



Shearing

Rotation

Random cropping



Local warping

Data augmentation – cont.

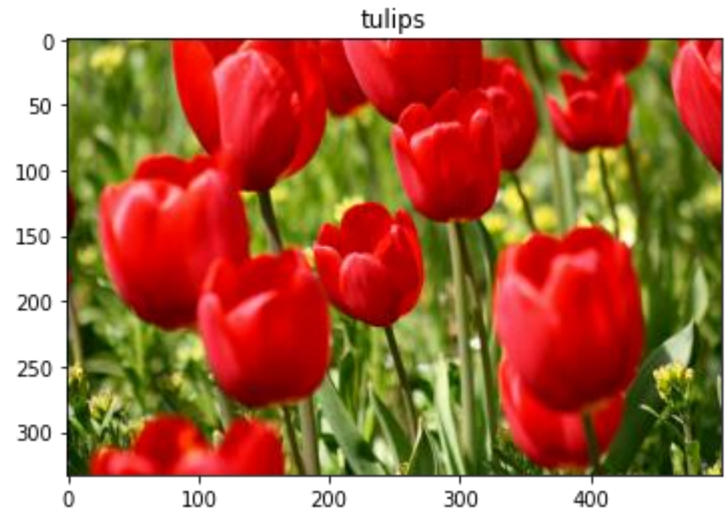
Color shifting

Original image

Noise injection

https://www.photopea.com

Use Keras preprocessing layers

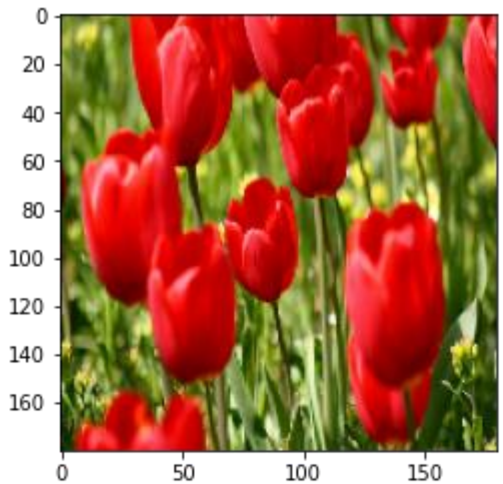


tf_flowers dataset

Resizing and rescaling

Use preprocessing layers to resize images to a consistent shape, and to rescale pixel values.

```
IMG_SIZE = 180
resize_and_rescale = tf.keras.Sequential([
  layers.experimental.preprocessing.Resizing(IMG_SIZE,
IMG_SIZE),
  layers.experimental.preprocessing.Rescaling(1./255)
])
```

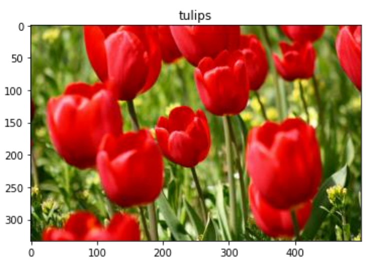


Data augmentation

Use preprocessing layers for data augmentation

```
data_augmentation = tf.keras.Sequential([
  layers.experimental.preprocessing.RandomFlip("horizontal_and_vertical"),
  layers.experimental.preprocessing.RandomRotation(0.2),
])
```

```
for i in range(9):
  augmented_image = data_augmentation(image)
  ax = plt.subplot(3, 3, i + 1)
  plt.imshow(augmented_image[0])
  plt.axis("off")
```



There are two ways you can use these preprocessing layers, with important tradeoffs.

Option 1: Make the preprocessing layers part of your model

```
model = tf.keras.Sequential([
  resize_and_rescale,
  data_augmentation,
  layers.Conv2D(16, 3, padding='same', activation='relu'),
  layers.MaxPooling2D(),
  # Rest of your model
])
```

Note: Data augmentation is inactive at test time so input images will only be augmented during calls to **`model.fit`** (not **`model.evaluate`** or **`model.predict`**).

Option 2: Apply the preprocessing layers to your dataset

```
aug_ds = train_ds.map(
  lambda x, y: (data_augmentation(x, training=True), y))
```

Note: data augmentation should only be applied to the training set.