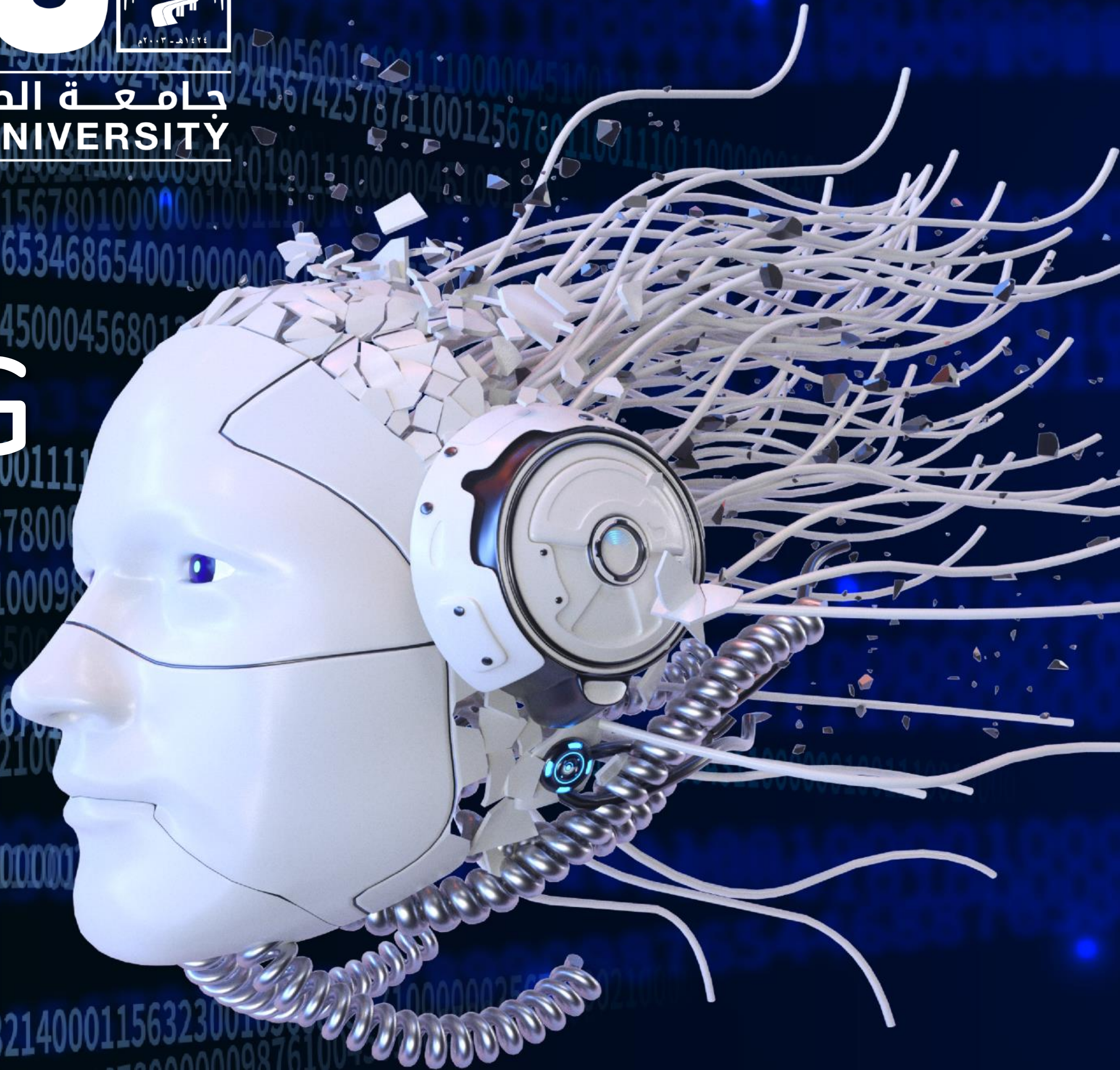
جامعة الطائف
TAIF UNIVERSITY

DEEP LEARNING



Assoc. Prof. Dr. Salha Alzahrani

Lecture Notes for MSc. in Data Science

My Notes

At glance!

- We can now elevate our knowledge to the point where we'll be able to successfully solve real-world computer vision tasks with Convolutional Neural Networks (CNNs).

- This chapter will cover the following topics:
  - Transfer learning
  - Diagram of the transfer learning scenario
  - Advanced network architectures: VGG, Residual networks
  - Advanced computer vision tasks
    - Object detection
    - Semantic segmentation
    - Artistic style transfer
  - Transfer Learning Examples with CIFAR-10: VGG, ResNet
  - Transfer Learning Examples with Dogs vs Cats: VGG, ResNet

Transfer learning

- So far, we've trained small models on datasets, where the training took no more than an hour. But if we want to work with large datasets, such as ImageNet, we will need a much bigger network that trains for a lot longer. More importantly, large datasets are not always available for the tasks we're interested in. Keep in mind that besides obtaining the images, they have to be labeled, and this could be expensive and time-consuming. So, what does a data scientist do when they want to solve a real ML problem with limited resources?

- **Transfer learning** is the process of applying an existing trained ML model to a new, but related, problem.
    - For example, we can take a network trained on ImageNet and repurpose it to classify grocery store items.
    - Alternatively, we could use a driving simulator game to train a neural network to drive a simulated car, and then use the network to drive a real car.

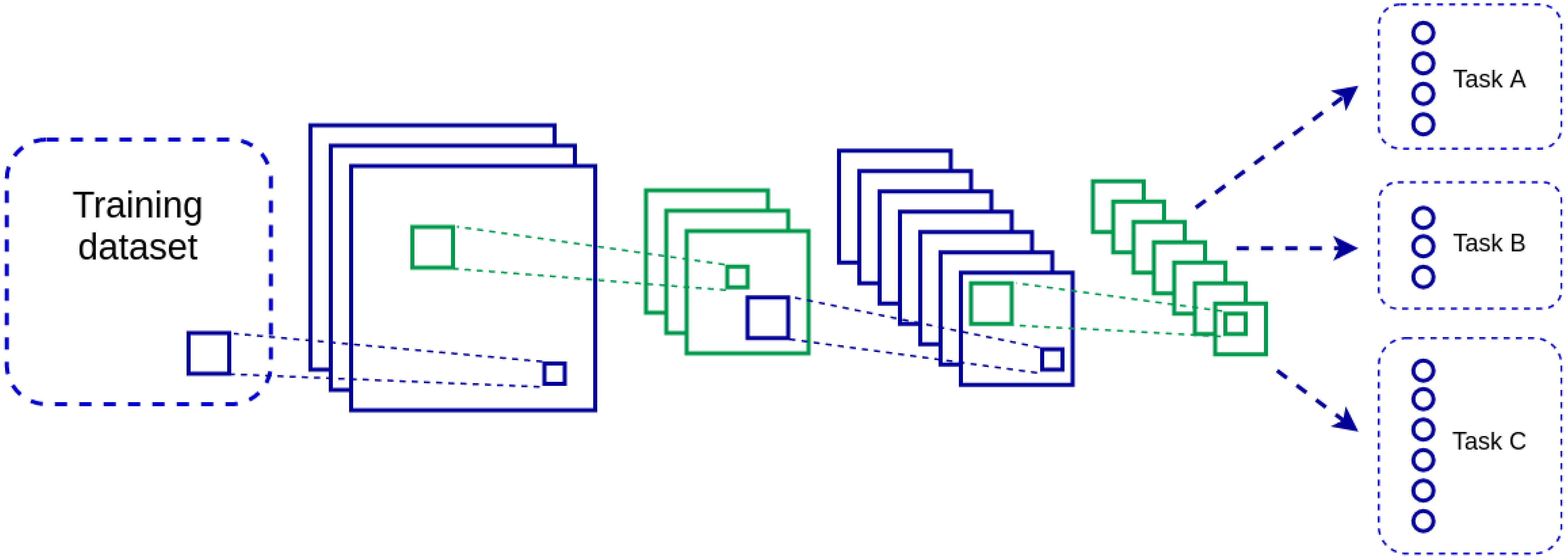
Transfer learning

- We start with an existing pre-trained net. The most common scenario is to take a net pre-trained with ImageNet, but it could be any dataset. TensorFlow/Keras/PyTorch all have popular ImageNet pre-trained neural architectures that we can use.

- In Convolutional Networks, we mentioned how the fully-connected layers at the end of a CNN act as **translators, or mappers**, between the abstract feature representations learned during training and the class of each sample.

- In transfer learning, we start with the network's features, which is the output of the last convolutional or pooling layer. Then, we translate them to a different set of classes of the new task. We can do this by:
  - removing the last fully-connected layer (or all fully-connected layers) of an existing pre-trained network, and
  - replacing it with another layer, which represents the classes of the new problem.

Diagram of the transfer learning scenario

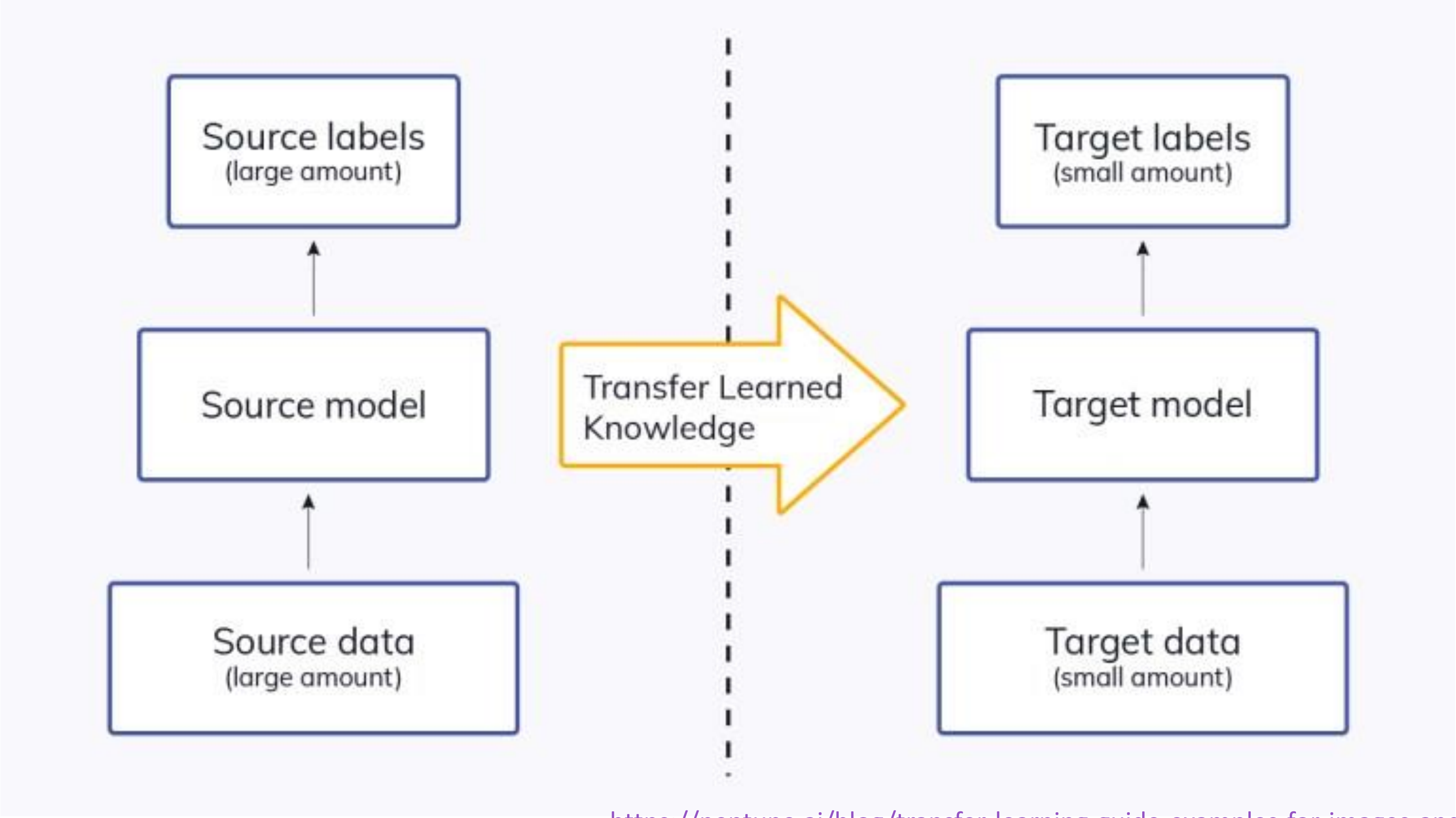


Transfer learning

- In transfer learning, we can replace the last layer of a pre-trained net and repurpose it for a new problem. But we cannot do this mechanically and expect the new network to work, because we still have to train the new layer with data related to the new task. **Here, we have two options:**
  - Use the original part of the network as feature extractor and only train the new layer(s): we feed the network a training batch of the new data and propagate it forward to see the network output. This part works just such as regular training would. But in the backward pass, we lock the weights of the original net and only update the weights of the new layers. This is recommended when we have limited training data on the new problem. By locking most of the network weights, we prevent overfitting on the new data.
  - Fine-tuning the whole network: we train the whole network, and not just the newly added layers at the end. It is possible to update all network weights, but we can also lock some of the weights in the first layers. The idea here is that the initial layers detect general features – not related to a specific task – and it makes sense to reuse them. On the other hand, the deeper layers might detect task-specific features and it would be better to update them. We can use this method when we have more training data to avoid overfitting.

Transfer learning

Transfer learning is about leveraging feature representations from a pre-trained model, so you don't have to train a new model from scratch.



https://neptune.ai/blog/transfer-learning-guide-examples-for-images-and-text-in-keras

Transfer learning

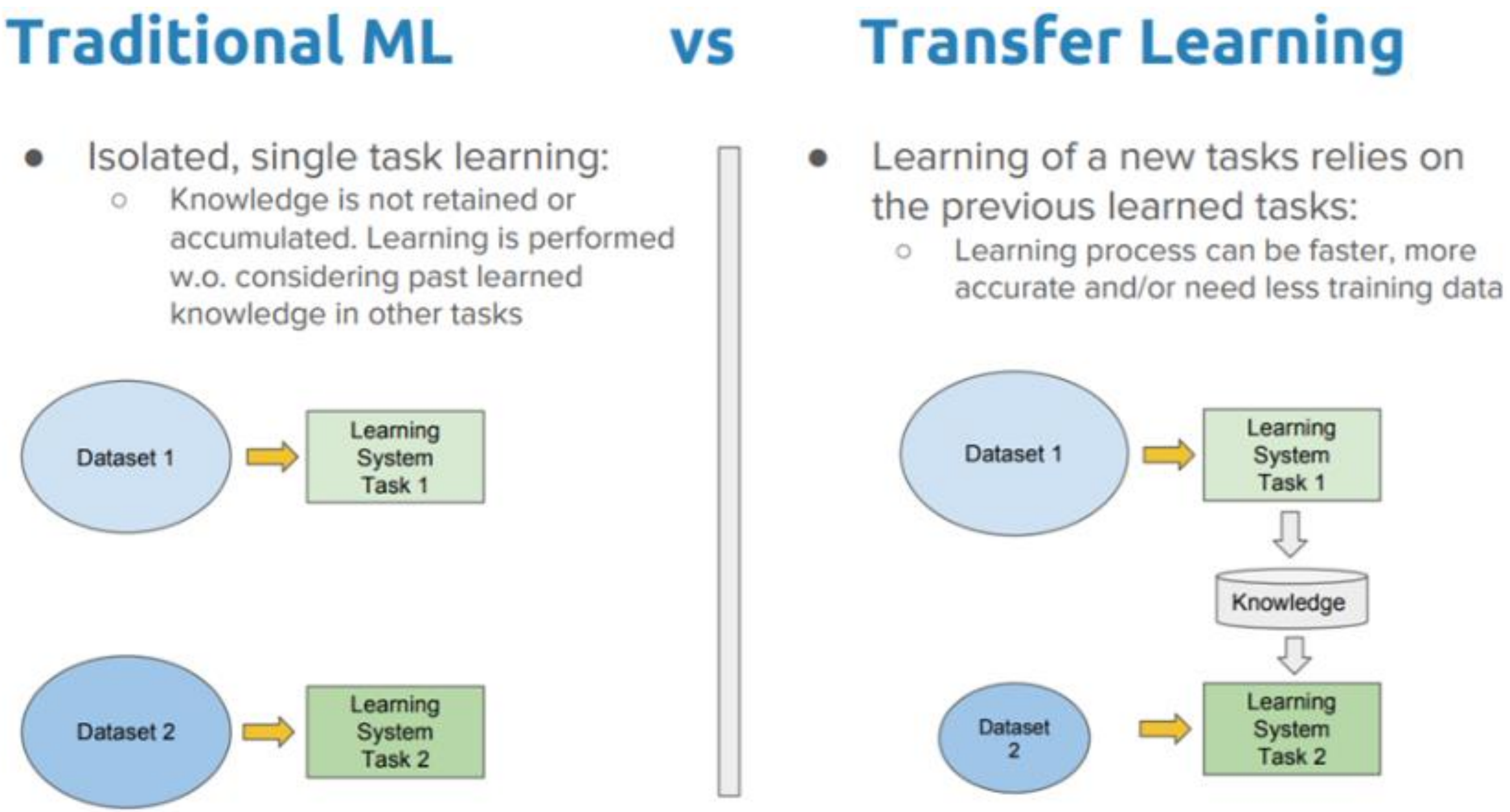
The advantage of pre-trained models is that they are generic enough for use in other real-world applications. For example:

- **Models trained on the ImageNet can be used in real-world image classification problems**. This is because the dataset contains over 1000 classes. Let's say you are an insect researcher. You can use these models and fine-tune them to classify insects.

- **Classifying text requires knowledge of word representations in some vector space.** You can train vector representations yourself. The challenge here is that you might not have enough data to train the embeddings. Furthermore, training will take a long time. In this case, you can use a pre-trained word embedding like GloVe to hasten your development process.

https://neptune.ai/blog/transfer-learning-guide-examples-for-images-and-text-in-keras

A comparison of traditional learning and transfer learning (Image credits: Dipanjar Sarkar)



Watch this video: What is Transfer Learning!

Advanced network architectures

VGG

- The first architecture we're going to discuss is VGG (from Oxford's Visual Geometry Group, https://arxiv.org/abs/1409.1556).
- It was introduced in 2014, when it became a runner-up in the ImageNet challenge of that year. The VGG family of networks remains popular today and is often used as a benchmark against newer architectures.
- Prior to VGG, for example, LeNet-5 (http://yann.lecun.com/exdb/lenet/) and AlexNet (https://papers.nips.cc/paper/4824-imagenet-classification-with-deep-convolutional-neural-networks.pdf), the initial convolutional layers of a network used filters with large receptive fields, such as 7 x 7. Additionally, the networks usually had alternating single convolutional and pooling layers.
- The authors of VGG paper observed that a convolutional layer with a large filter size can be replaced with a stack of two or more convolutional layers with smaller filters (factorized convolution).
- For example, we can replace one 5 x 5 layer with a stack of two 3 x 3 layers, or a 7 x 7 layer with a stack of three 3 x 3 layers.

Advanced network architectures

- The VGG networks consist of multiple blocks of two, three, or four stacked convolutional layers combined with a max-pooling layer.
- Architecture of **VGG16** and **VGG19** networks, named so after the number of weighted layers in each network.
- As the depth of the VGG network increases, so does the width (number of filters) in the convolutional layers. We have multiple pairs of convolutional layers with a volume depth of 128/256/512 connected to other layers with the same depth. In addition, we also have two 4,096-neuron fully-connected layers.
- Because of this, the VGG networks have large number of parameters (weights), which makes them memory-inefficient, as well computationally expensive.
- Still, this is a popular and straightforward network architecture, which has been further improved by the addition of batch normalization.

<table>
<tr><th>VGG16</th><th>VGG19</th></tr>
<tr><td>conv 3x3, 64</td><td>conv 3x3, 64</td></tr>
<tr><td>conv 3x3, 64</td><td>conv 3x3, 64</td></tr>
<tr><td colspan="2">max pool</td></tr>
<tr><td>conv 3x3, 128</td><td>conv 3x3, 128</td></tr>
<tr><td>conv 3x3, 128</td><td>conv 3x3, 128</td></tr>
<tr><td colspan="2">max pool</td></tr>
<tr><td>conv 3x3, 256</td><td>conv 3x3, 256</td></tr>
<tr><td>conv 3x3, 256</td><td>conv 3x3, 256</td></tr>
<tr><td>conv 3x3, 256</td><td>conv 3x3, 256</td></tr>
<tr><td></td><td>conv 3x3, 256</td></tr>
<tr><td colspan="2">max pool</td></tr>
<tr><td>conv 3x3, 512</td><td>conv 3x3, 512</td></tr>
<tr><td>conv 3x3, 512</td><td>conv 3x3, 512</td></tr>
<tr><td>conv 3x3, 512</td><td>conv 3x3, 512</td></tr>
<tr><td></td><td>conv 3x3, 512</td></tr>
<tr><td colspan="2">max pool</td></tr>
<tr><td>conv 3x3, 512</td><td>conv 3x3, 512</td></tr>
<tr><td>conv 3x3, 512</td><td>conv 3x3, 512</td></tr>
<tr><td>conv 3x3, 512</td><td>conv 3x3, 512</td></tr>
<tr><td></td><td>conv 3x3, 512</td></tr>
<tr><td colspan="2">max pool</td></tr>
<tr><td colspan="2">fc-4096</td></tr>
<tr><td colspan="2">fc-4096</td></tr>
<tr><td colspan="2">fc-1000</td></tr>
<tr><td colspan="2">softmax</td></tr>
</table>

Advanced network architectures

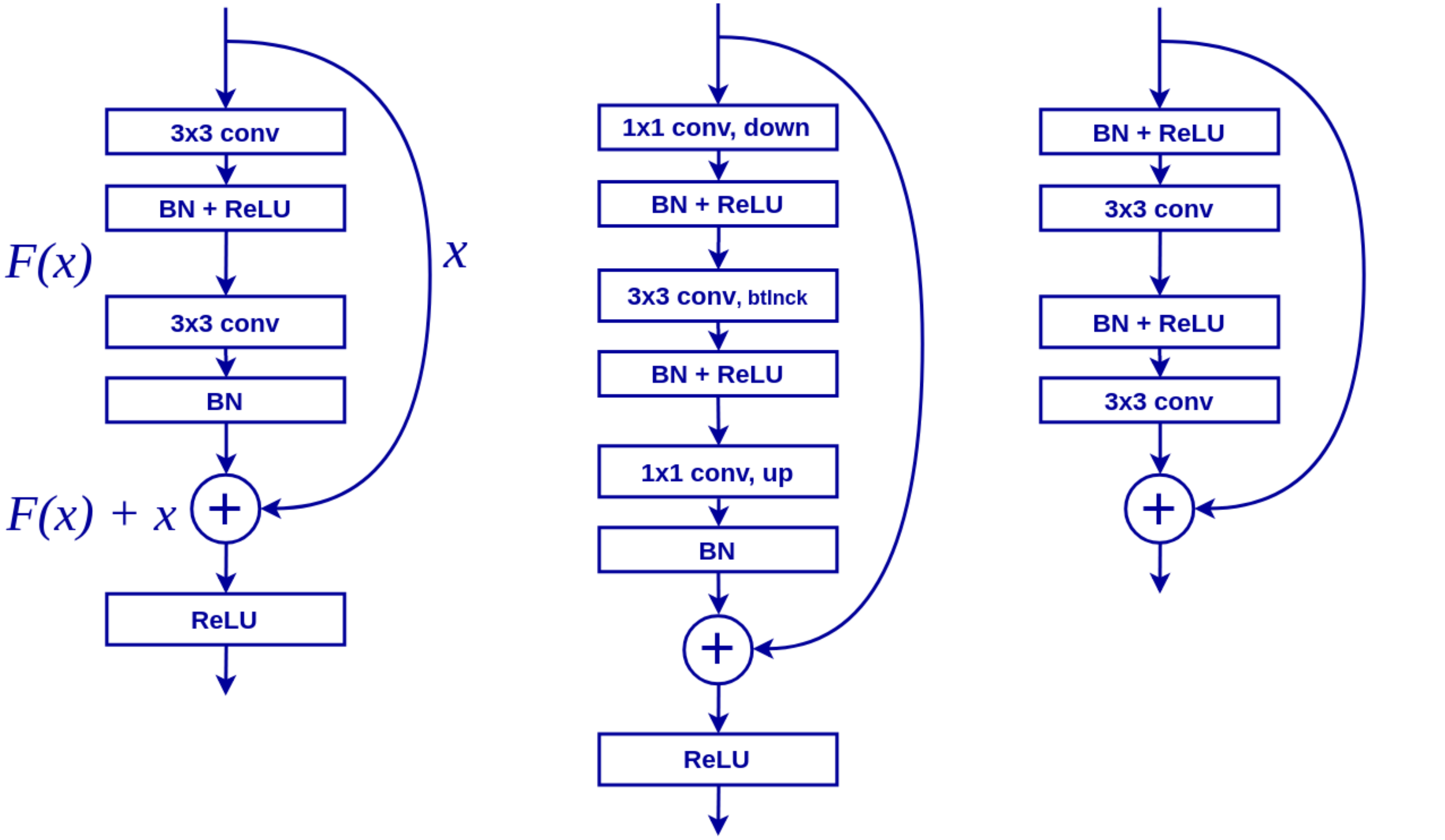
Residual networks

- Residual networks (ResNets, https://arxiv.org/abs/1512.03385) were released in 2015, when they won all five categories of the ImageNet challenge that year.
- Thanks to better weight initializations, new activation functions, as well as normalization layers, it's now possible to train deep networks. But the authors of the paper conducted some experiments and observed that a network with 56 layers had higher training and testing errors compared to a network with 20 layers. They argue that this should not be the case. In theory, we can take a shallow network and stack identity layers (these are layers whose output just repeats the input) on top of it to produce a deeper network that behaves in exactly the same way as the shallow one. Yet, their experiments have been unable to match the performance of the shallow network.

- To solve this problem, they proposed a network constructed of residual blocks. A residual block consists of two or three sequential convolutional layers and a separate parallel identity (repeater) shortcut connection, which connects the input of the first layer and the output of the last one.

Advanced network architectures

- We can see three types of residual blocks in the diagram:
- Each block has two parallel paths. The left path is similar to the other networks we've seen, and consists of sequential convolutional layers + batch normalization (BN). The right path contains the identity shortcut connection (also known as skip connection).
- The two paths are merged via an element-wise sum. That is, the left and right tensors have the same shape and an element of the first tensor is added to the element of the same position of the second tensor.
- The output is a single tensor with the same shape as the input.



Original residual block Original bottleneck residual block Residual block v2

Advanced network architectures

Keras Applications

- Keras Applications are deep learning models that are made available alongside pre-trained weights. These models can be used for prediction, feature extraction, and fine-tuning.

- Weights are downloaded automatically when instantiating a model. They are stored at ~/.keras/models/

Model	Size (MB)	Top-1 Accuracy	Top-5 Accuracy	Parameters	Depth	Time (ms) per inference step (CPU)	Time (ms) per inference step (GPU)
Xception	88	79.0%	94.5%	22.9M	81	109.4	8.1
VGG16	528	71.3%	90.1%	138.4M	16	69.5	4.2
VGG19	549	71.3%	90.0%	143.7M	19	84.8	4.4
ResNet50	98	74.9%	92.1%	25.6M	107	58.2	4.6
ResNet50V2	98	76.0%	93.0%	25.6M	103	45.6	4.4
ResNet101	171	76.4%	92.8%	44.7M	209	89.6	5.2
ResNet101V2	171	77.2%	93.8%	44.7M	205	72.7	5.4
ResNet152	232	76.6%	93.1%	60.4M	311	127.4	6.5
ResNet152V2	232	78.0%	94.2%	60.4M	307	107.5	6.6
InceptionV3	92	77.9%	93.7%	23.9M	189	42.2	6.9
InceptionResNetV2	215	80.3%	95.3%	55.9M	449	130.2	10.0
MobileNet	16	70.4%	89.5%	4.3M	55	22.6	3.4

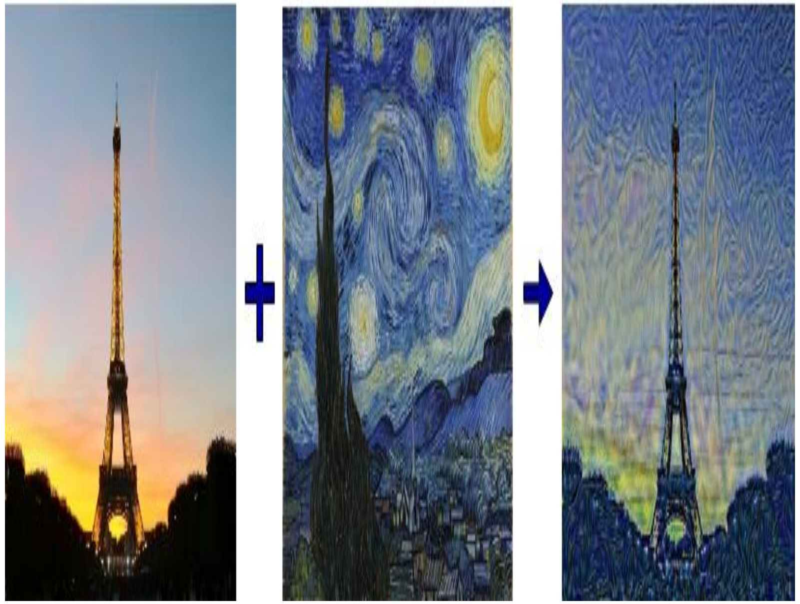
https://keras.io/api/applications/

Advanced computer vision tasks

Object detection

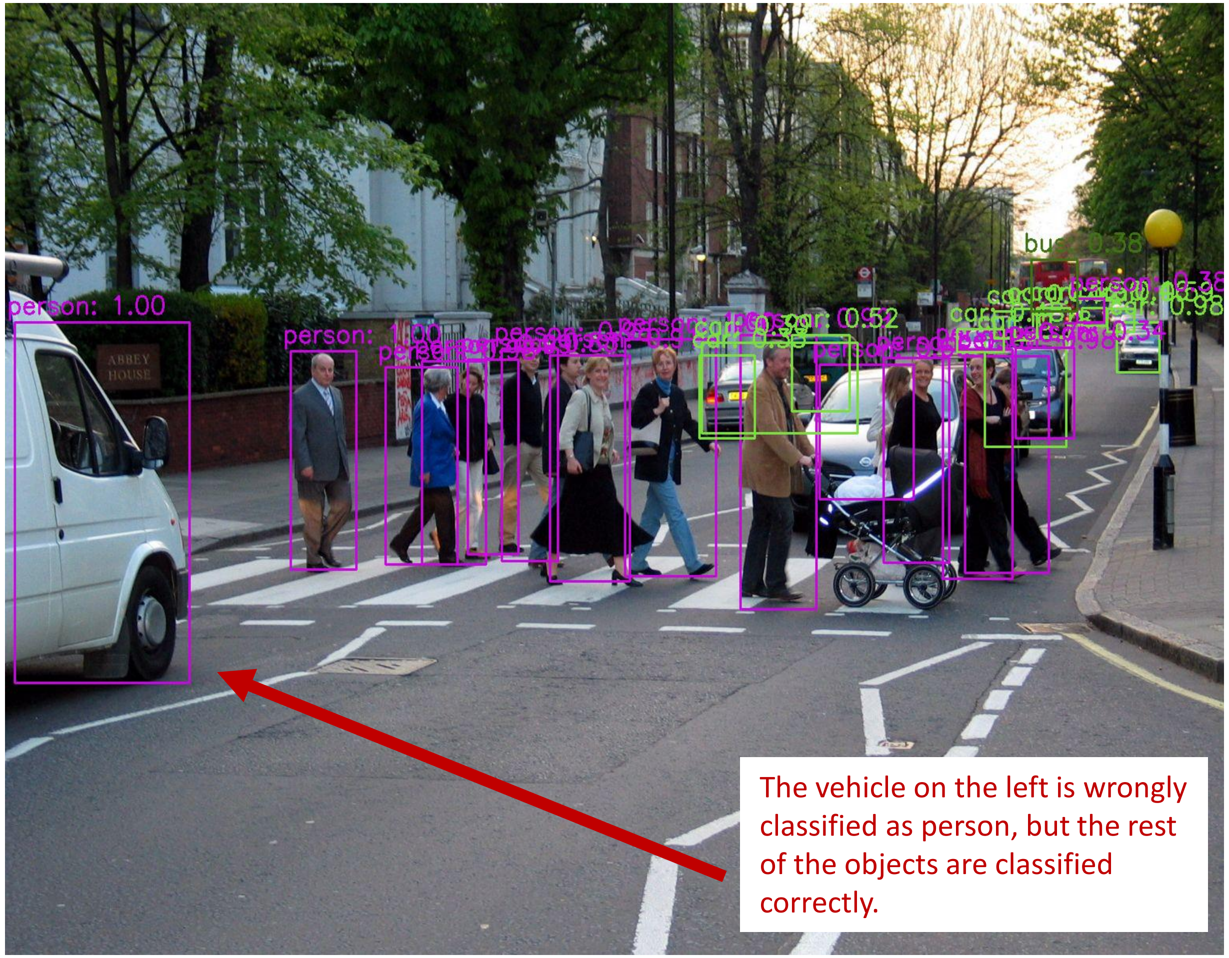
Semantic segmentation

Artistic style transfer



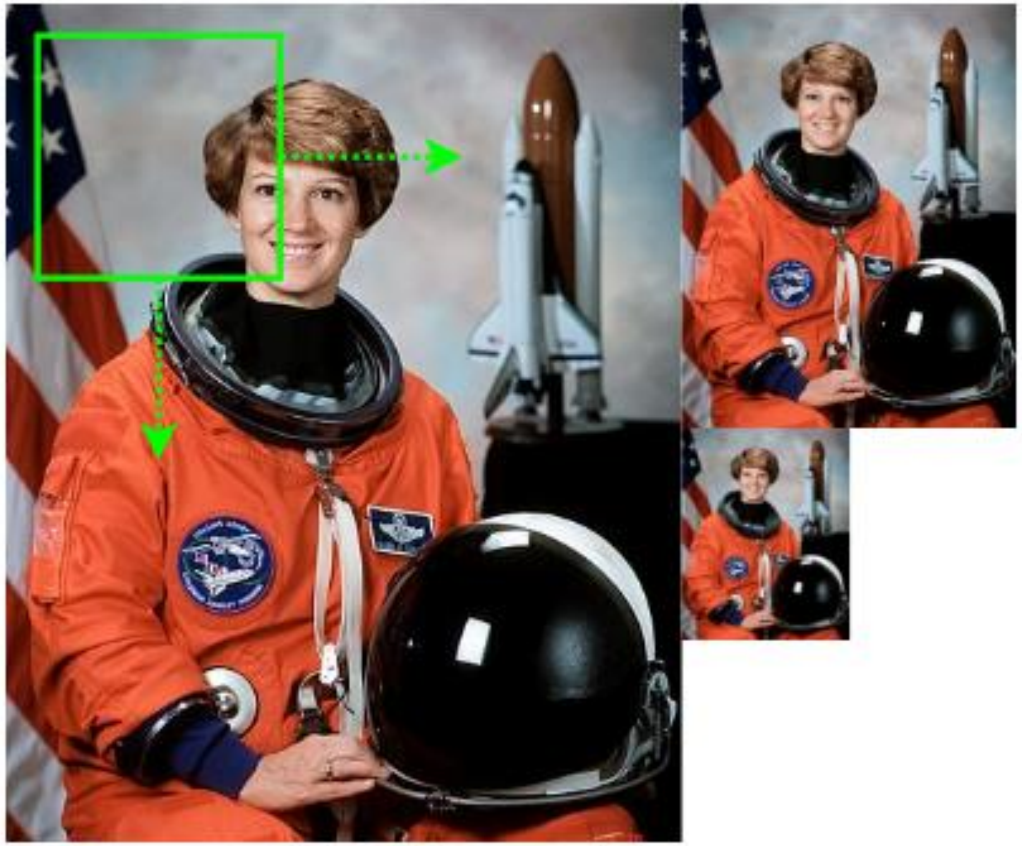
Object detection

- Object detection is the process of finding object instances of a certain class, such as faces, cars, and trees, in images or videos.
- Unlike classification, object detection can detect multiple objects, as well as their location.
- An object detector returns a list of detected objects with the following information for each object:
  1) Define the class of the object (person, car, tree, etc.).
  2) The probability (or confidence score) in the [0, 1] range, conveys how confident the detector is that the object exists in that location. This is similar to the output of a regular classifier.
  3) The coordinates of the rectangular region of the image where the object is located. This rectangle is called a bounding box.



Approaches to object detection

- **Classic sliding window:** This approach use a regular classification network (classifier) and can work with any type of classification algorithm, but it's relatively slow and error-prone:
  1) **Build an image pyramid.** This is a combination of different scales of the same image (see the following photograph). For example, each scaled image can be two times smaller than the previous one. In this way, we'll be able to detect objects regardless of their size in the original image.
  2) **Slide the classifier across the whole image.** That is, we'll use each location of the image as an input to the classifier and the result will determine what type of object is in that location. The bounding box of that location is just the image region that we used as input.
  3) **We'll have multiple overlapping bounding boxes for each object.** We'll use some heuristics to combine them in a single prediction.



Sliding window + image pyramid object detection

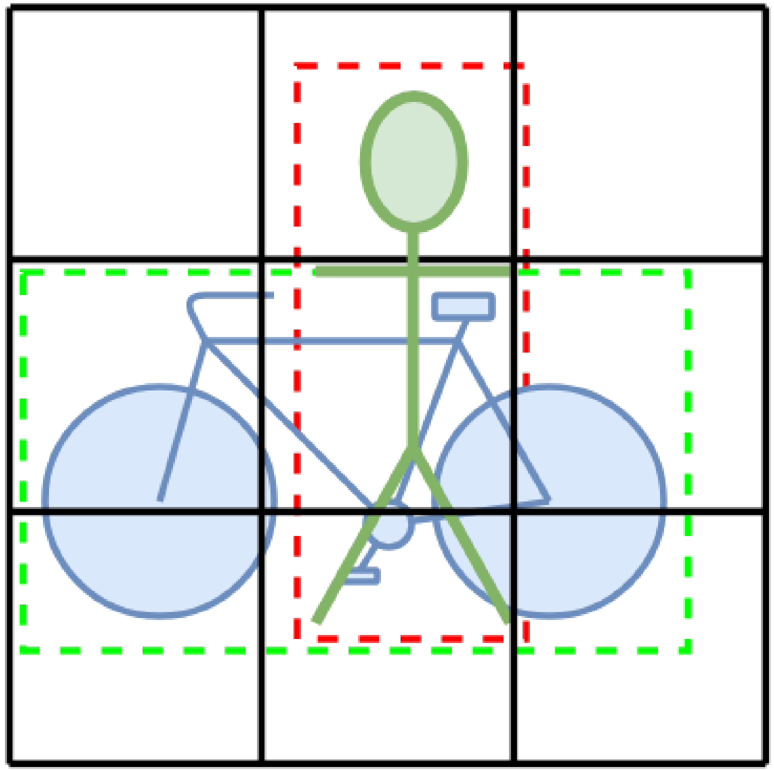
Approaches to object detection

- **Two-stage detection methods:** These methods are very accurate, but relatively slow. As the name suggests, they involve two steps:
  1) **A special type of CNN, called a Region Proposal Network**, scans the image and proposes a number of possible bounding boxes where objects might be located. However, this network doesn't detect the type of the object, but only whether an object is present in the region.
  2) The regions of interest are sent to the second stage for object classification.

- **One-stage detection methods:** Here, a single CNN produces both the object type and the bounding box. These approaches are usually faster, but less accurate compared to two-stage methods.

Object detection with YOLOv3

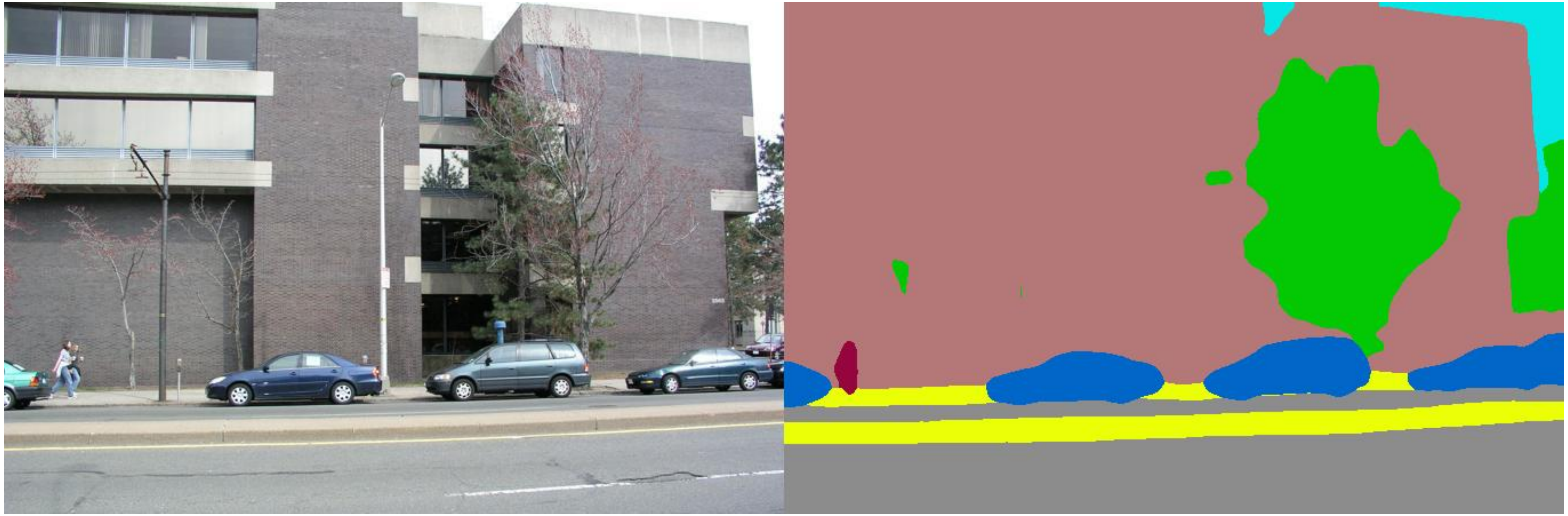
- In this section, we'll discuss one of the most popular detection algorithms, called **YOLO**. The name is an acronym for the popular motto "You only live once," which reflects the one-stage nature of the algorithm. The authors have released three versions with incremental improvements of the algorithm. We'll first discuss the latest, v3.
- How YOLO works:
  - It works with a fully-convolutional network (without pooling layers), not unlike the ones we've seen in this chapter. It uses residual connections and batch normalization. The YOLOv3 network uses three different scales of the image for prediction. What makes it different, though, is the use of special type of ground-truth/output data, which is a combination of classification and regression.
  - The network takes the whole image as an input and outputs the bounding boxes, object classes, and confidence scores of all detected objects in just a single pass. For example, the bounding boxes in the image of people on the crosswalk at the beginning of this section were generated using a single network pass.



An object detection YOLO example with a 3 x 3 cell grid, 2 objects, and their bounding boxes (dashed lines). Both objects are associated with the middle cell, because the centers of their bounding boxes lie in that cell

Semantic segmentation

- **Semantic segmentation** is the process of assigning a class label (such as person, car, or tree) to each pixel of the image. You can think of it as classification, but on a pixel level – instead of classifying the entire image under one label, we'll classify each pixel separately. Here is an example of semantic segmentation:



Semantic segmentation

- To train a segmentation algorithm, we need a special type of ground-truth data, where the labels for each image are the semantically segmented version of the image.
- There are many approaches to semantic segmentation:
  1) The easiest way is using the **sliding-window technique**, which we described in object detection. That is, we'll use a regular classifier and we'll slide it in either direction with stride 1. After we get the prediction for a location, we'll take the pixel that lies in the middle of the input region and we'll assign it with the predicted class. Predictably, this approach is very slow, due to the large number of pixels in an image (even a 1024 x 1024 image has more than 1,000,000 pixels).
  2) We can use a special type of CNN, called **Fully Convolutional Network (FCN)**, to classify all pixels in the input region in a single pass. We can separate an FCN into two virtual components:
     - **The encoder** is the first part of the network. It is such as a regular CNN, without the fully-connected layers at the end. The role of the encoder is to learn highly abstract representations of the input image.
     - **The decoder** is the second part of the network. It starts after the encoder and uses it as input. The role of the decoder is to "translate" these abstract representations into the segmented groundtruth data. To do this, the decoder uses the opposite of the encoder operations. This includes unpooling (the opposite of pooling) and deconvolutions (the opposite of convolutions). We'll talk more about this concept (but in different context) in Chapter 6, Generating images with GANs and VAEs.

Artistic style transfer

- **Artistic style transfer** is the use of the style (or texture) of one image to reproduce the semantic content of another. It can be implemented with different algorithms, but the most popular way was introduced in 2015 in the paper A Neural Algorithm of Artistic Style (https://arxiv.org/abs/1508.06576). It's also known as neural style transfer, and it uses CNNs. The basic algorithm has been improved and tweaked over the past few years.

- **The algorithm takes two images as input: Content image (C) we would like to redraw + Style image (I) whose style (texture) we'll use to redraw C**
- **The result of the algorithm is a new image: G = C + S**

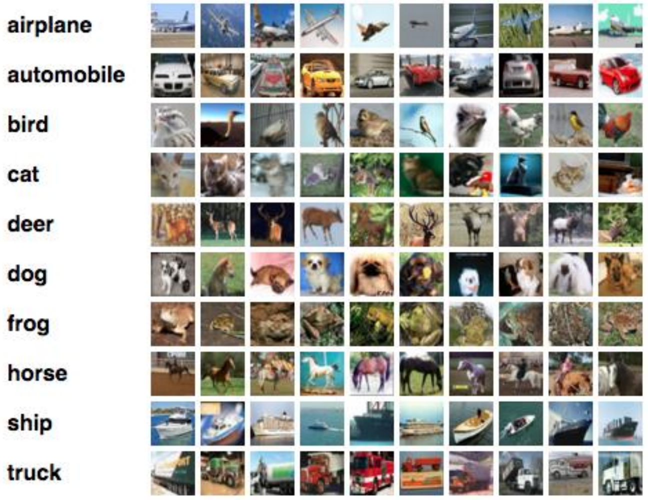
Let's have fun ☺

Transfer Learning with VGG and CIFAR-10 dataset

1. Prepare & Explore Dataset

In [1]:

```
# import pachages
from tensorflow import keras
from keras.datasets import cifar10
from keras.layers import Conv2D, MaxPooling2D
from keras.layers import Dense, Dropout, Activation, Flatten, BatchNormalization
from keras.models import Sequential
from keras.preprocessing.image import ImageDataGenerator
from keras.utils import to_categorical

from keras.applications import VGG19

from keras.optimizers import SGD,Adam
```



Transfer Learning with VGG and CIFAR-10 dataset

In [2]:
```
# Load data.
(x_train,y_train),(x_test,y_test)=cifar10.load_data()
```

In [3]:
```
from sklearn.model_selection import train_test_split
x_train, x_val, y_train, y_val = train_test_split(x_train, y_train, test_size=.3)
```

In [4]:
```
print((x_train.shape,y_train.shape))
print((x_val.shape,y_val.shape))
print((x_test.shape,y_test.shape))
```

```
((35000, 32, 32, 3), (35000, 1))
((15000, 32, 32, 3), (15000, 1))
((10000, 32, 32, 3), (10000, 1))
```

In [5]:
```
# We have 10 classes, so, our network will have 10 output neurons
y_train = to_categorical(y_train)
y_val = to_categorical(y_val)
y_test = to_categorical(y_test)
```

In [6]:
```
print((x_train.shape,y_train.shape))
print((x_val.shape,y_val.shape))
print((x_test.shape,y_test.shape))
```

```
((35000, 32, 32, 3), (35000, 10))
((15000, 32, 32, 3), (15000, 10))
((10000, 32, 32, 3), (10000, 10))
```

Transfer Learning with VGG and CIFAR-10 dataset

We can now begin the actual process of model building. The following a set process and following consistently makes learning this easier :

- Define the Data Augmentation (ImageDataGenerator)
- Build the model (Base Model + Flatten + Dense)
- Check model summary
- Initialize Batch Size,Number of Epochs
- Compile model
- Fit the model
- Evaluate the model

Transfer Learning with VGG and CIFAR-10 dataset

In [7]:

```
#Data Augmentation Function: Let's define an instance of the ImageDataGenerator class and set the parameters
#We have to instantiate for the Train,Validation and Test datasets
train_generator = ImageDataGenerator(
                                    rotation_range=2,
                                    horizontal_flip=True,
                                    zoom_range=.1 )

val_generator = ImageDataGenerator(
                                    rotation_range=2,
                                    horizontal_flip=True,
                                    zoom_range=.1)

test_generator = ImageDataGenerator(
                                    rotation_range=2,
                                    horizontal_flip= True,
                                    zoom_range=.1)
```

Transfer Learning with VGG and CIFAR-10 dataset

2. Define the neural network architecture

In [8]:
```
# define the CNN model
'The first base model used is VGG19. The pretrained weights from the imagenet challenge are used'
base_model_1 = VGG19(include_top=False, weights='imagenet', input_shape=(32,32,3), classes=y_train.shape[1])
```

In [9]:
```
#Lets add the final layers to these base models where the actual classification is done in the dense layers
model_1 = Sequential()
model_1.add(base_model_1)
model_1.add(Flatten())
```

In [10]:
```
print(model_1.summary())
```

```
Model: "sequential"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
vgg19 (Functional)           (None, 1, 1, 512)         20024384
_________________________________________________________________
flatten (Flatten)            (None, 512)               0
=================================================================
```

Transfer Learning with VGG and CIFAR-10 dataset

In [11]:
```
#Add the Dense layers along with activation and batch normalization
model_1.add(Dense(1024,activation=('relu'),input_dim=512))
model_1.add(Dense(512,activation=('relu')))
model_1.add(Dense(256,activation=('relu')))
model_1.add(Dropout(.3))#Adding a dropout layer that will randomly drop 30% of the weights
model_1.add(Dense(128,activation=('relu')))
model_1.add(Dropout(.2))
model_1.add(Dense(10,activation=('softmax'))) #This is the classification layer
```

In [12]:
```
#Check final model summary
model_1.summary()
```

Layer (type)	Output Shape	Param #
vgg19 (Functional)	(None, 1, 1, 512)	20024384
flatten (Flatten)	(None, 512)	0
dense (Dense)	(None, 1024)	525312
dense_1 (Dense)	(None, 512)	524800
dense_2 (Dense)	(None, 256)	131328
dropout (Dropout)	(None, 256)	0
dense_3 (Dense)	(None, 128)	32896
dropout_1 (Dropout)	(None, 128)	0
dense_4 (Dense)	(None, 10)	1290

3. Compile the neural net

In [13]:
```
# compile your model
model_1.compile(loss='categorical_crossentropy', optimizer='sgd', metrics=['accuracy'])
```

Transfer Learning with VGG and CIFAR-10 dataset

4. Fit / train the neural net

In [14]:

```
batch_size= 100
epochs=20

model_1.fit(x_train, y_train, batch_size=batch_size, epochs=epochs, validation_data=(x_val, y_val), verbose=1)
```

```
acy: 0.7823
Epoch 16/50
350/350 [==============================] - 1089s 3s/step - loss: 0.0976 - accuracy: 0.9707 - val_loss: 0.7304 - val_accur
acy: 0.8228
Epoch 17/50
350/350 [==============================] - 1087s 3s/step - loss: 0.0727 - accuracy: 0.9801 - val_loss: 0.7629 - val_accur
acy: 0.8231
Epoch 18/50
350/350 [==============================] - 1077s 3s/step - loss: 0.0569 - accuracy: 0.9839 - val_loss: 0.7344 - val_accur
acy: 0.8311
Epoch 19/50
350/350 [==============================] - 1088s 3s/step - loss: 0.0477 - accuracy: 0.9878 - val_loss: 0.8520 - val_accur
acy: 0.8291
Epoch 20/50
350/350 [==============================] - 1073s 3s/step - loss: 0.0888 - accuracy: 0.9762 - val_loss: 0.8307 - val_accur
acy: 0.8251
```

Transfer Learning with VGG and CIFAR-10 dataset

5. Evaluate the neural net

In [16]:
```
score = model_1.evaluate(x_val, y_val, verbose=1)
print('Test accuracy:', score[1])
```

```
469/469 [==============================] - 33s 69ms/step - loss: 1.3226 - accuracy: 0.8456
Test accuracy: 0.8456000089645386
```

6. Make predictions / classifications for unseen data

In [18]:
```
#not yet until we enhanced the results
predictions = model_1.predict(x_test)
predictions
```

Transfer Learning with ResNet and CIFAR-10 dataset

1. Prepare & Explore Dataset

In [1]:

```
# import pachages
from tensorflow import keras
from keras.datasets import cifar10
from keras.layers import Conv2D, MaxPooling2D
from keras.layers import Dense, Dropout, Activation, Flatten, BatchNormalization
from keras.models import Sequential
from keras.preprocessing.image import ImageDataGenerator
from keras.utils import to_categorical

from keras.applications import ResNet50

from keras.optimizers import SGD,Adam
```

Transfer Learning with ResNet and CIFAR-10 dataset

2. Define the neural network architecture

In [10]:
```
# define the CNN model
'For the 2nd base model we will use Resnet 50 and compare the performance against the previous one'
'The hypothesis is that Resnet 50 should perform better because of its deeper architecture'
base_model_2 = ResNet50(include_top=False, weights='imagenet', input_shape=(32,32,3), classes=y_train.shape[1])
```

```
Downloading data from https://storage.googleapis.com/tensorflow/keras-applications/resnet/resnet5
f_kernels_notop.h5
94773248/94765736 [==============================] - 14s 0us/step
```

In [11]:
```
#Lets add the final layers to these base models where the actual classification is done in the de
#Since we have already defined Resnet50 as base_model_2, let us build the sequential model.
model_2 = Sequential()
#Add the Dense layers along with activation and batch normalization
model_2.add(base_model_2)
model_2.add(Flatten())

#Add the Dense layers along with activation and batch normalization
model_2.add(Dense(4000,activation=('relu'),input_dim=512))
model_2.add(Dense(2000,activation=('relu')))
model_2.add(Dropout(.4))
model_2.add(Dense(1000,activation=('relu')))
model_2.add(Dropout(.3))#Adding a dropout layer that will randomly drop 30% of the weights
model_2.add(Dense(500,activation=('relu')))
model_2.add(Dropout(.2))
model_2.add(Dense(10,activation=('softmax'))) #This is the classification layer
```

```
print(model_2.summary())
```

resnet50 (Functional)	(None, 1, 1, 2048)	23587712
flatten (Flatten)	(None, 2048)	0
dense (Dense)	(None, 4000)	8196000
dense_1 (Dense)	(None, 2000)	8002000
dropout (Dropout)	(None, 2000)	0
dense_2 (Dense)	(None, 1000)	2001000
dropout_1 (Dropout)	(None, 1000)	0
dense_3 (Dense)	(None, 500)	500500
dropout_2 (Dropout)	(None, 500)	0
dense_4 (Dense)	(None, 10)	5010

Transfer Learning with ResNet and CIFAR-10 dataset

4. Fit / train the neural net

```
In [*]:
batch_size= 100
epochs=20
model_2.fit(x_train, y_train, batch_size=batch_size, epochs=epochs, validation_data=(x_val, y_val), verbose=1)
```

```
Epoch 1/20
350/350 [==============================] - 990s 3s/step - loss: 2.1358 - accuracy: 0.2356 - val_loss: 1.3044 - val_accuracy:
0.5507
Epoch 2/20
350/350 [==============================] - 968s 3s/step - loss: 1.1031 - accuracy: 0.6161 - val_loss: 0.9298 - val_accuracy:
0.6858
Epoch 3/20
350/350 [==============================] - 968s 3s/step - loss: 0.7916 - accuracy: 0.7323 - val_loss: 0.8582 - val_accuracy:
0.7109
Epoch 4/20
350/350 [==============================] - 988s 3s/step - loss: 0.6036 - accuracy: 0.7965 - val_loss: 0.7845 - val_accuracy:
0.7419
Epoch 5/20
350/350 [==============================] - 1011s 3s/step - loss: 0.4691 - accuracy: 0.8404 - val_loss: 0.8299 - val_accurac
y: 0.7398
Epoch 6/20
221/350 [=================>............] - ETA: 6:13 - loss: 0.3592 - accuracy: 0.8809
```

Transfer Learning with ResNet and CIFAR-10 dataset

5. Evaluate the neural net

In [15]:
```
score = model_2.evaluate(x_val, y_val, verbose=1)
print('Test accuracy:', score[1])
```

```
469/469 [==============================] - 33s 68ms/step - loss: 0.9280 - accuracy: 0.7317
Test accuracy: 0.7317333221435547
```

6. Make predictions / classifications for unseen data

In [ ]:
```
#not yet until we enhanced the results
predictions = model_2.predict(x_test)
predictions
```

Classification of Cats vs. Dogs using Transfer Learning with **VGG and ResNet**

Dogs vs. Cats

Create an algorithm to distinguish dogs from cats

Kaggle · 213 teams · 9 years ago

Overview Data Code Discussion Leaderboard Rules Team

« Windows (C:) › Users › HP › #JupyterNotebooks# Deep Learning › dogs-vs-cats ›

Name	Date modified	Type	Size
test	10/9/2022 5:59 PM	File folder	
train	10/10/2022 12:19 PM	File folder	
test1	10/1/2022 7:08 PM	Compressed (zipp...	277,658 KB
train	10/1/2022 7:08 PM	Compressed (zipp...	556,198 KB

Dataset Description

The training archive contains 25,000 images of dogs and cats. Train your algorithm on these files and predict the labels for test1.zip (1 = dog, 0 = cat).

Files

3 files

Size

853.96 MB

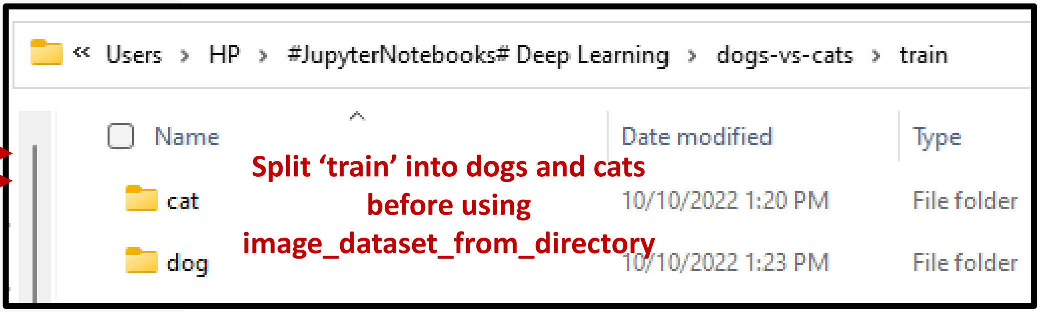
https://www.kaggle.com/competitions/dogs-vs-cats/data

Classification of Cats vs. Dogs using Transfer Learning with **VGG and ResNet**

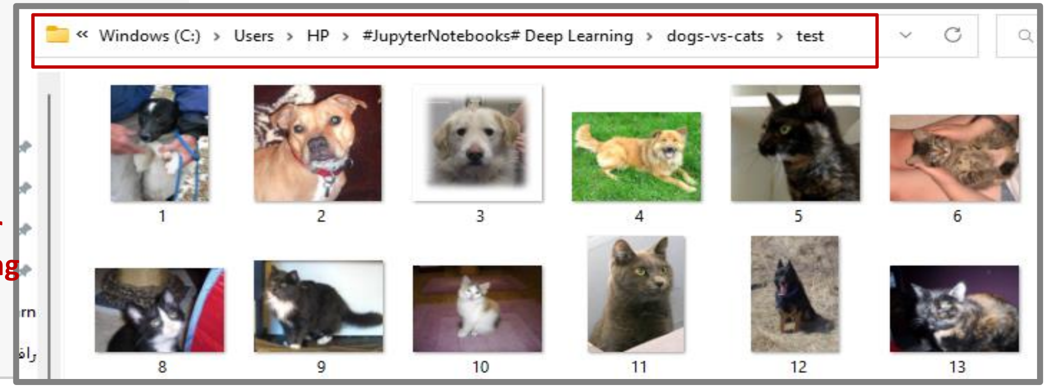
In [57]:

```
from tensorflow.keras.preprocessing import image_dataset_from_directory

train_ds = image_dataset_from_directory(
    directory='dogs-vs-cats/train/',
    labels='inferred',
    label_mode='categorical',
    batch_size=32,
    image_size=(160, 160),
    validation_split=0.1,
    subset="training",
    seed=1024
)

val_ds = image_dataset_from_directory(
    directory='dogs-vs-cats/train/',
    labels='inferred',
    label_mode='categorical',
    batch_size=32,
    image_size=(160, 160),
    validation_split=0.1,
    subset="validation",
    seed=1024
)
```

```
Found 25000 files belonging to 2 classes.
Using 22500 files for training.
Found 25000 files belonging to 2 classes.
Using 2500 files for validation.
```



Split ‘train’ into dogs and cats before using image_dataset_from_directory



‘test’ has no names
So can be used only for testing and implementing the model

Object Detection using YOLOv3

Step 5: Create Model and Load Weights

Now let us use all the above defined functions to detect objects in an image.First we define a model and load weights in it.Then we start loading the picture and preproceesing it.

```
# define the model
model = make_yolov3_model()
```

```
#read weights from yolov3 weights file provided in the data
weight_reader = WeightReader('data-for-yolo-v3-kernel/yolov3.weights')
```

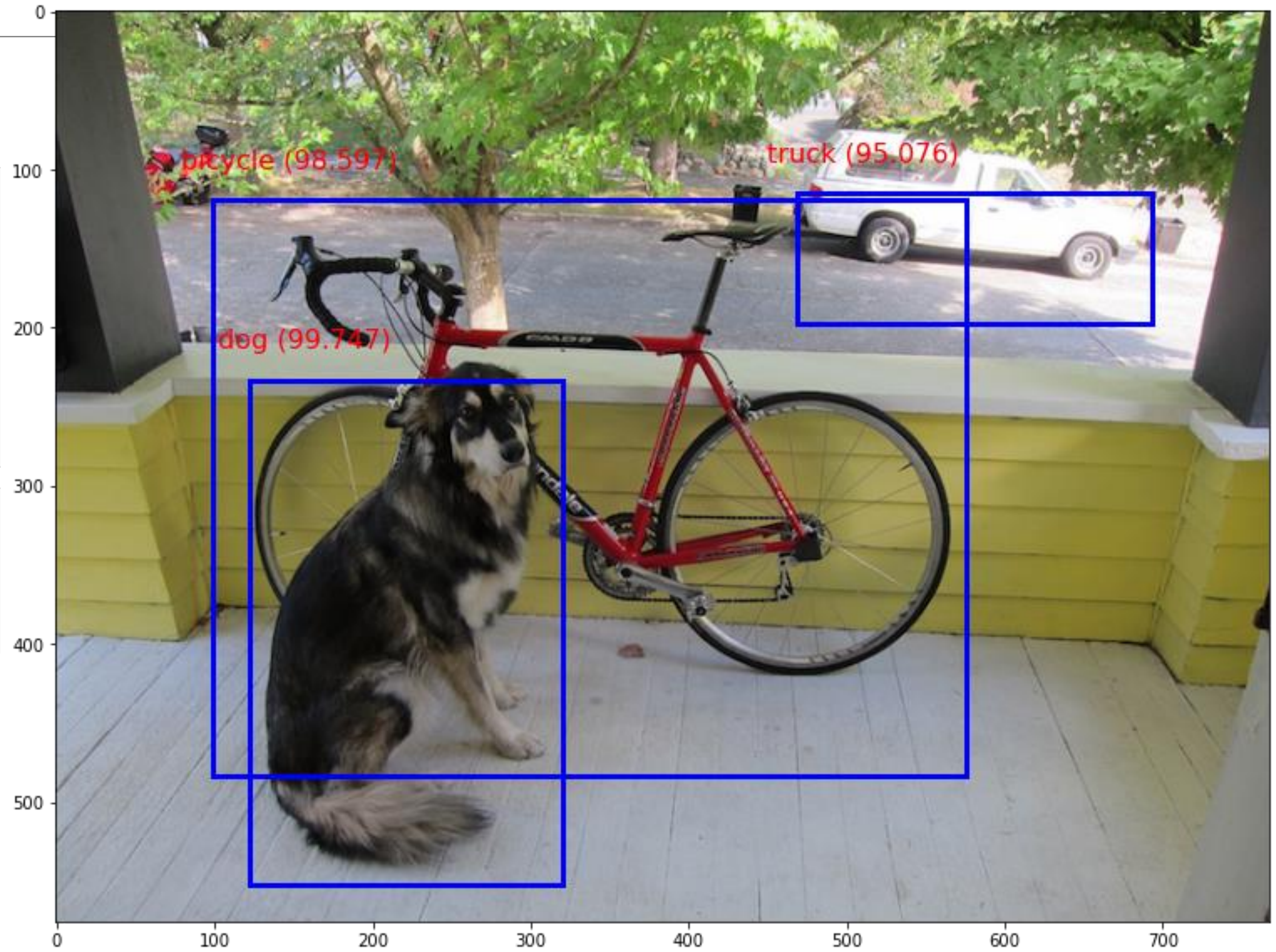
```
# set the model weights into the model
weight_reader.load_weights(model)
```

Object Detection using YOLOv3

Step 6: Use YOLO to Detect Objects in New Images

```python
# define the labels
labels = []
with open("data-for-yolo-v3-kernel/coco.names", "r") as f:
    labels = [line.strip() for line in f.readlines()]

# define the anchors
anchors = [[116,90, 156,198, 373,326], [30,61, 62,45, 59,119], [10,13, 16,30, 33,23]]

# define the probability threshold for detected objects
class_threshold = 0.7
```

```python
# define the expected input shape for the model
input_w, input_h = 416, 416

# define our new photo
photo_filename = 'data-for-yolo-v3-kernel/dog.jpg'
```




```
truck 95.0762391090393
bicycle 98.59656095504761
dog 99.74659085273743
```

Object Detection using YOLOv3

```python
# define our new photo
photo_filename = 'data-for-yolo-v3-kernel/office.jpg'
```

```
person 97.65632748603821
tvmonitor 92.88041591644287
person 84.24031138420105
person 99.7834324836731
person 97.21069931983948
chair 74.42909479141235
chair 84.44295525550842
```

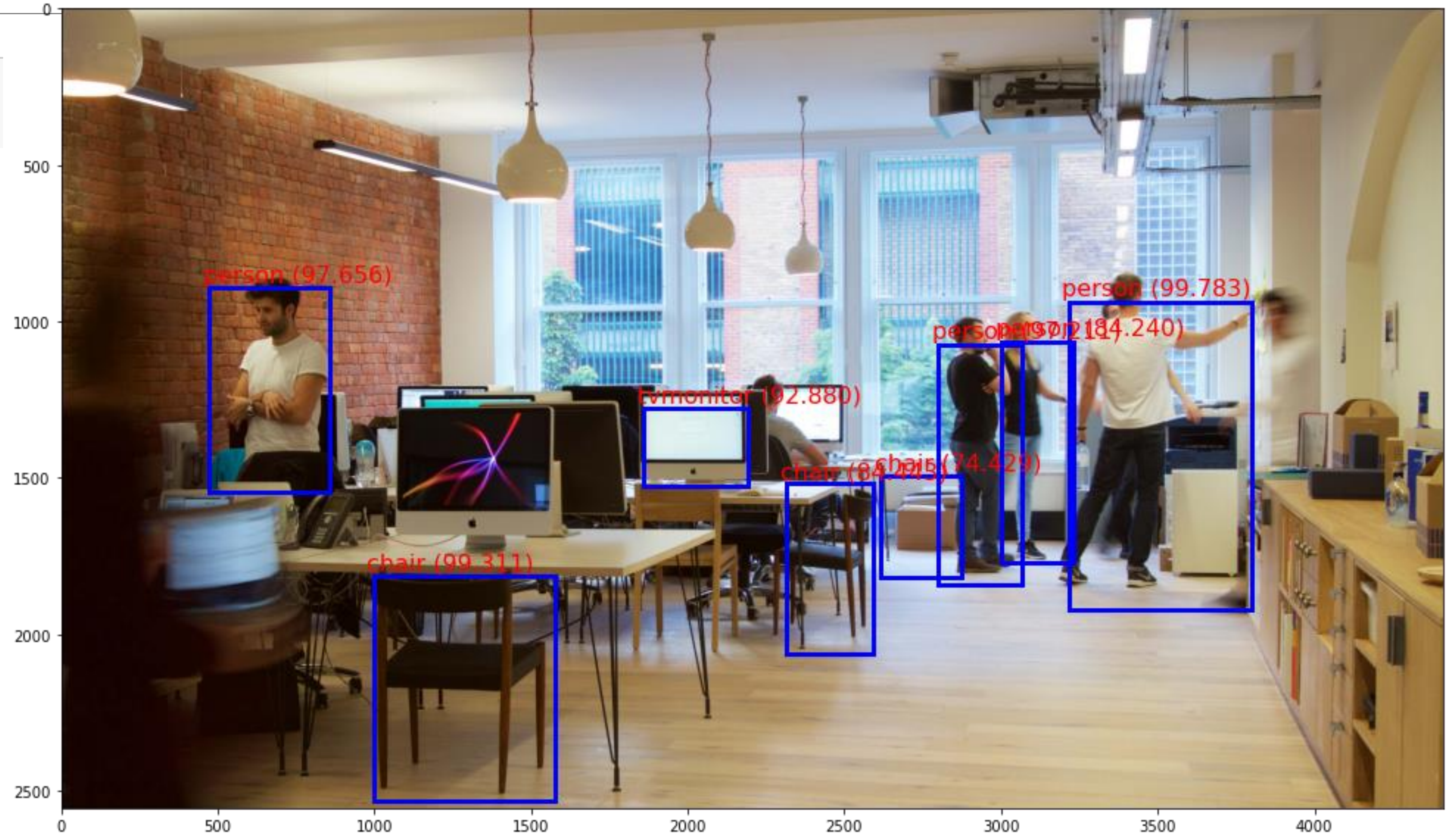


Image Segmentation on Oxford-IIIT Pets dataset

In [1]:

```
import os

input_dir = "oxford-pets/images/"
target_dir = "oxford-pets/annotations/trimaps/"

input_img_paths = sorted(
    [os.path.join(input_dir, fname)
     for fname in os.listdir(input_dir)
     if fname.endswith(".jpg")])
target_paths = sorted(
    [os.path.join(target_dir, fname)
     for fname in os.listdir(target_dir)
     if fname.endswith(".png") and not fname.startswith(".")])

print("Number of samples:", len(input_img_paths))
```

```
Number of samples: 7390
```

In [2]:

```
import matplotlib.pyplot as plt
from tensorflow.keras.preprocessing.image import load_img, img_to_array

plt.axis("off")
plt.imshow(load_img(input_img_paths[9]))
```

Image Segmentation on Oxford-IIIT Pets dataset

In [5]:

```
from tensorflow import keras
from tensorflow.keras import layers

num_classes=3

model = keras.Sequential([
    layers.experimental.preprocessing.Rescaling(1./255, input_shape=img_size + (3,)),
    layers.Conv2D(64, 3, strides=2, activation="relu", padding="same"),
    layers.Conv2D(64, 3, activation="relu", padding="same"),
    layers.Conv2D(128, 3, strides=2, activation="relu", padding="same"),
    layers.Conv2D(128, 3, activation="relu", padding="same"),
    layers.Conv2D(256, 3, strides=2, padding="same", activation="relu"),
    layers.Conv2D(256, 3, activation="relu", padding="same"),

    layers.Conv2DTranspose(256, 3, activation="relu", padding="same"),
    layers.Conv2DTranspose(256, 3, activation="relu", padding="same", strides=2),
    layers.Conv2DTranspose(128, 3, activation="relu", padding="same"),
    layers.Conv2DTranspose(128, 3, activation="relu", padding="same", strides=2),
    layers.Conv2DTranspose(64, 3, activation="relu", padding="same"),
    layers.Conv2DTranspose(64, 3, activation="relu", padding="same", strides=2),

    layers.Conv2D(num_classes, 3, activation="softmax", padding="same")
    ]
)

model.summary()
```

Model: "sequential"

Layer (type)	Output Shape	Param #
rescaling (Rescaling)	(None, 200, 200, 3)	0
conv2d (Conv2D)	(None, 100, 100, 64)	1792
conv2d_1 (Conv2D)	(None, 100, 100, 64)	36928
conv2d_2 (Conv2D)	(None, 50, 50, 128)	73856
conv2d_3 (Conv2D)	(None, 50, 50, 128)	147584
conv2d_4 (Conv2D)	(None, 25, 25, 256)	295168
conv2d_5 (Conv2D)	(None, 25, 25, 256)	590080
conv2d_transpose (Conv2DTran	(None, 25, 25, 256)	590080
conv2d_transpose_1 (Conv2DTr	(None, 50, 50, 256)	590080
conv2d_transpose_2 (Conv2DTr	(None, 50, 50, 128)	295040
conv2d_transpose_3 (Conv2DTr	(None, 100, 100, 128)	147584
conv2d_transpose_4 (Conv2DTr	(None, 100, 100, 64)	73792
conv2d_transpose_5 (Conv2DTr	(None, 200, 200, 64)	36928
conv2d_6 (Conv2D)	(None, 200, 200, 3)	1731

Image Segmentation on Oxford-IIIT Pets dataset

In [11]:

```
from keras.preprocessing.image import array_to_img

model = keras.models.load_model("oxford_segmentation.keras")

i = 4
test_image = val_input_imgs[i]
plt.axis("off")
plt.imshow(array_to_img(test_image))

mask = model.predict(np.expand_dims(test_image, 0))[0]

def display_mask(pred):
    mask = np.argmax(pred, axis=-1)
    mask *= 127
    plt.axis("off")
    plt.imshow(mask)

display_mask(mask)
```

My Notes

Recap!

- ☑ Transfer learning
- ☑ Diagram of the transfer learning scenario
- ☑ Advanced network architectures:
  - ❑ VGG
  - ❑ Residual networks
  - ❑ Keras Applications
- ☑ Advanced computer vision tasks
  - ❑ Object detection
  - ❑ Semantic segmentation
  - ❑ Artistic style transfer
- ☑ Transfer Learning with CIFAR-10: VGG, ResNet
- ☑ Transfer Learning with Dogs vs Cats: VGG, ResNet

@SalhaAlzahrani