Foundations for Machine Learning

L. Y. Stefanus

TU Dresden, June-July 2019

Machine Learning with PyTorch

Reference

- Eli Stevens and Luca Antiga. Deep Learning with PyTorch. Manning Publications, 2019/2020.

What Is PyTorch?

- PyTorch is a machine learning tool developed by Facebook's AI division to process large-scale object detection, segmentation, classification, etc.
- It is written using Python and C++.
- PyTorch provides a framework to write functions that automatically run in a GPU environment.

PyTorch Installation

- Installing PyTorch is simple. In Windows, Linux, or macOS, it is recommended to use the Anaconda distribution environment, including Python and the IDE Spyder. See https://www.anaconda.com
- The following steps describe how to install PyTorch in Windows/macOS/Linux environments.

1. Install the Anaconda Distribution
2. Open the Anaconda navigator and go to the environment page.

PyTorch Installation

3. Open the conda terminal and type the following command:

   ```
   conda install pytorch torchvision
   ```

   to install pytorch and torchvision.
4. Launch the IDE Spyder.
5. Type the following command to check whether the PyTorch is installed or not and check the version of the PyTorch:

   ```
   import torch
   torch.version.__version__
   ```

```
Python 3.6.8 |Anaconda, Inc.| (default, Feb 21 2019, 18:30:04) [MSC v.1916 64 bit (AMD64)]
Type "copyright", "credits" or "license" for more information.

IPython 7.5.0 -- An enhanced Interactive Python.

In [1]: import torch

In [2]: torch.version.__version__
Out[2]: '1.1.0'
```

A Tour of Machine Learning with PyTorch

- Tasks that only a few years ago were thought to require higher cognition are getting solved by machines at near- to super-human levels of performance. For example, describing a photographic image with a correct English sentence and playing complex strategy games.
- Even more impressively, the ability to solve such tasks is acquired by computers *through examples*, rather than encoded by a human as a set of hand-crafted rules.

- That general class of algorithms we're talking about falls under the category of *deep learning*, which deals with training mathematical entities named *deep neural networks* on the basis of examples. Deep learning leverages large amounts of data to approximate complex functions whose inputs and outputs are far apart, like an input image and as output a line of text describing the input, or a written script as input and a natural-sounding voice reciting the script as output.
- This kind of capability allows us to create programs that has functionality that, until very recently, exclusively was the domain of human beings.

- PyTorch is a library for Python programs that facilitates building deep learning projects. It emphasises flexibility and allows deep learning models to be expressed in Python.
- PyTorch provides a core data structure, the tensor, a multi-dimensional array that shares many similarities with Numpy arrays.
- Tensors provide acceleration of mathematical operations and PyTorch has packages for distributed training, worker processes for efficient data loading, and an extensive library of common deep learning functions.

- While PyTorch was initially focused on research workflows, it has been equipped with a high-performance C++ runtime that can be leveraged to deploy models for inference without relying on Python. This allows us to keep most of the flexibility of PyTorch without having to pay the overhead of the Python runtime.

The Deep Learning Revolution

- Until the late 2000's, the broader class of AI systems that fell under the label "machine learning" heavily relied on *feature engineering*.
- Feature engineering is taking the original data and coming up with *representations* of the same data that can then be fed to an algorithm to solve a problem.
- For instance, in order to tell 1's from 0's in images of handwritten digits, one would come up with a set of filters to estimate the direction of edges over the image, and then train a classifier to predict the correct digit given a distribution of edge directions. Another useful feature could be the number of enclosed holes, as seen in a zero, or in an eight.

The Deep Learning Revolution

- Deep learning, on the other hand, deals with finding such representations automatically, from raw data, in order to successfully perform a task.
- In the 1's vs 0's example, filters would be refined during training, by iteratively looking at pairs of examples and target labels.
- The ability of a neural network to process data and extract useful representations on the basis of examples is what makes deep learning so powerful.
- The focus of deep learning practitioners is on operating on a mathematical entity so that it discovers representations from the training data autonomously.

The Deep Learning Revolution

- Often, these automatically-created features are better than those that are hand-crafted! As with many disruptive technologies, this fact has led to a change in perspective.
- On the left of Figure-1.1, we see a practitioner busy defining engineering features and feeding them to a learning algorithm; the results on their task will be as good as the features he or she engineers.
- On the right, with deep learning, the raw data is fed to an algorithm that extracts hierarchical features automatically, based on optimizing the performance of the algorithm on the task.
- The results will be as good as the ability of the practitioner to drive the algorithm towards its goal.

The Deep Learning Revolution

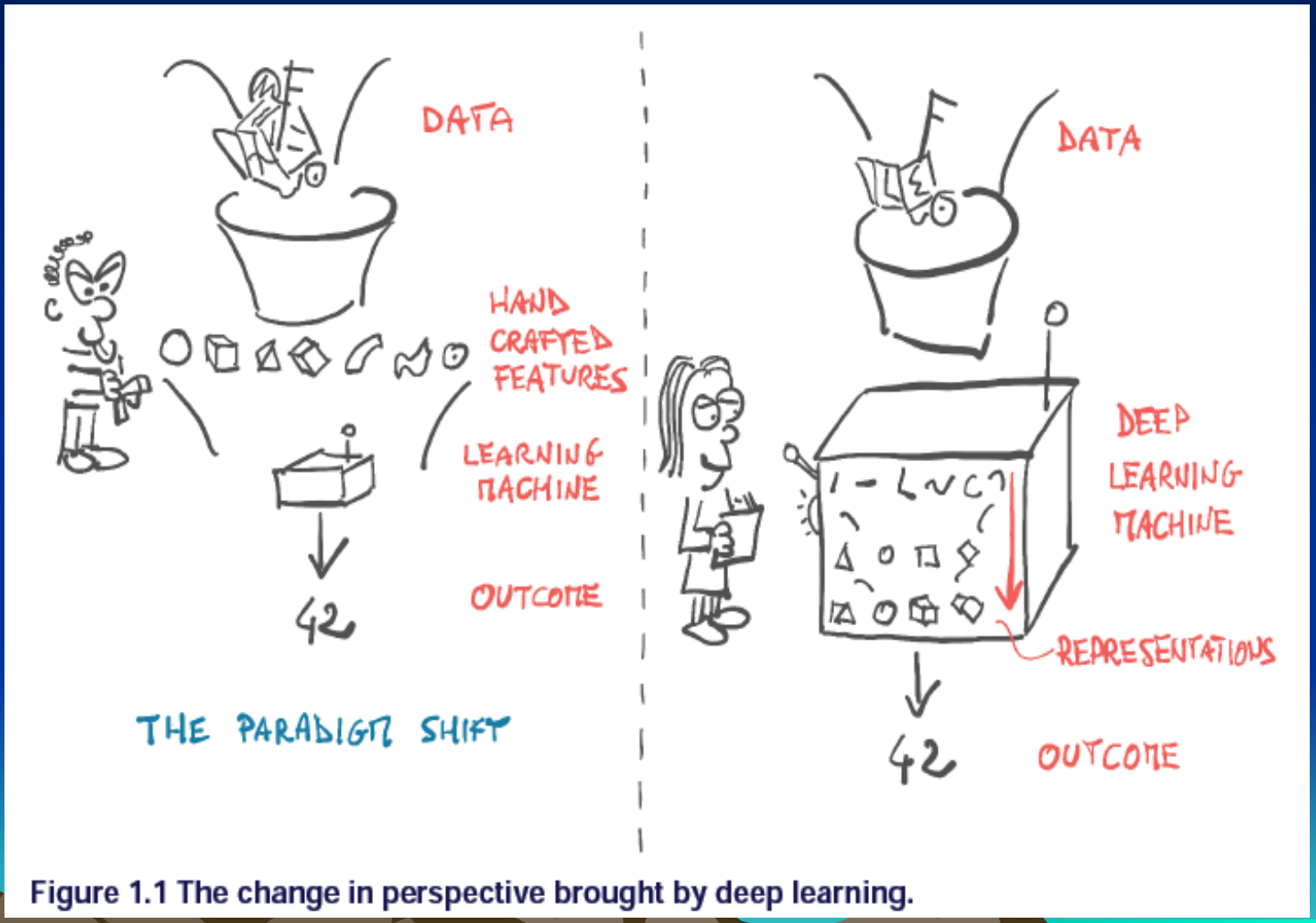

Figure 1.1 The change in perspective brought by deep learning.

The deep learning competitive landscape

- Theano, one of the first deep learning frameworks, has ceased active development.
- TensorFlow:
  - Consumed Keras entirely, promoting it to a first-class API
  - Provided an immediate execution "eager mode"
  - Announced that TF 2.0 will enable eager mode by default
- PyTorch:
  - Consumed Caffe2 for its backend
  - Added support for ONNX, a vendor-neutral model description and exchange format
  - Added a delayed execution "graph mode" runtime called *TorchScript*

The Main Components of PyTorch

- Firstly, PyTorch has the "Py" as in Python, but there's a lot of non-Python code in it. Actually, for performance reasons, most of PyTorch is written in C++ and CUDA, a C++-like language from NVIDIA that can be compiled to run with massive parallelism on NVIDIA GPUs. There are ways to run PyTorch directly from C++.
- One of the main motivations for this capability is to provide a reliable strategy for deploying models in production. However, most of the time we'll interact with PyTorch from Python, building models, training them, and using the trained models to solve actual problems.

The Main Components of PyTorch

- Indeed, the Python API is where PyTorch shines in term of usability and integration with the wider Python ecosystem.
- At the core, PyTorch is a library that provides tensors (multidimensional arrays) and an extensive library of operations on them, provided by the torch module.
- Both tensors and related operations can run on the CPU, or on the GPU. Running on the GPU results in massive speedups compared to CPU.
- The second core thing of PyTorch is the ability of tensors to keep track of the operations performed on them and to compute derivatives of an output with respect to any of its inputs analytically via back-propagation. This is provided natively by tensors, and further refined in torch.autograd.

The Main Components of PyTorch

- By having tensors and the autograd-enabled tensor standard library, PyTorch could be used for more than "just" neural networks. PyTorch can be used for physics, rendering, optimization, simulation, and other scientific applications.
- But PyTorch is first and foremost a deep learning library, and as such it provides all the building blocks needed to build neural networks and train them.
- Figure-1.4 shows a standard setup that loads data, trains a model, and then deploys that model to production.

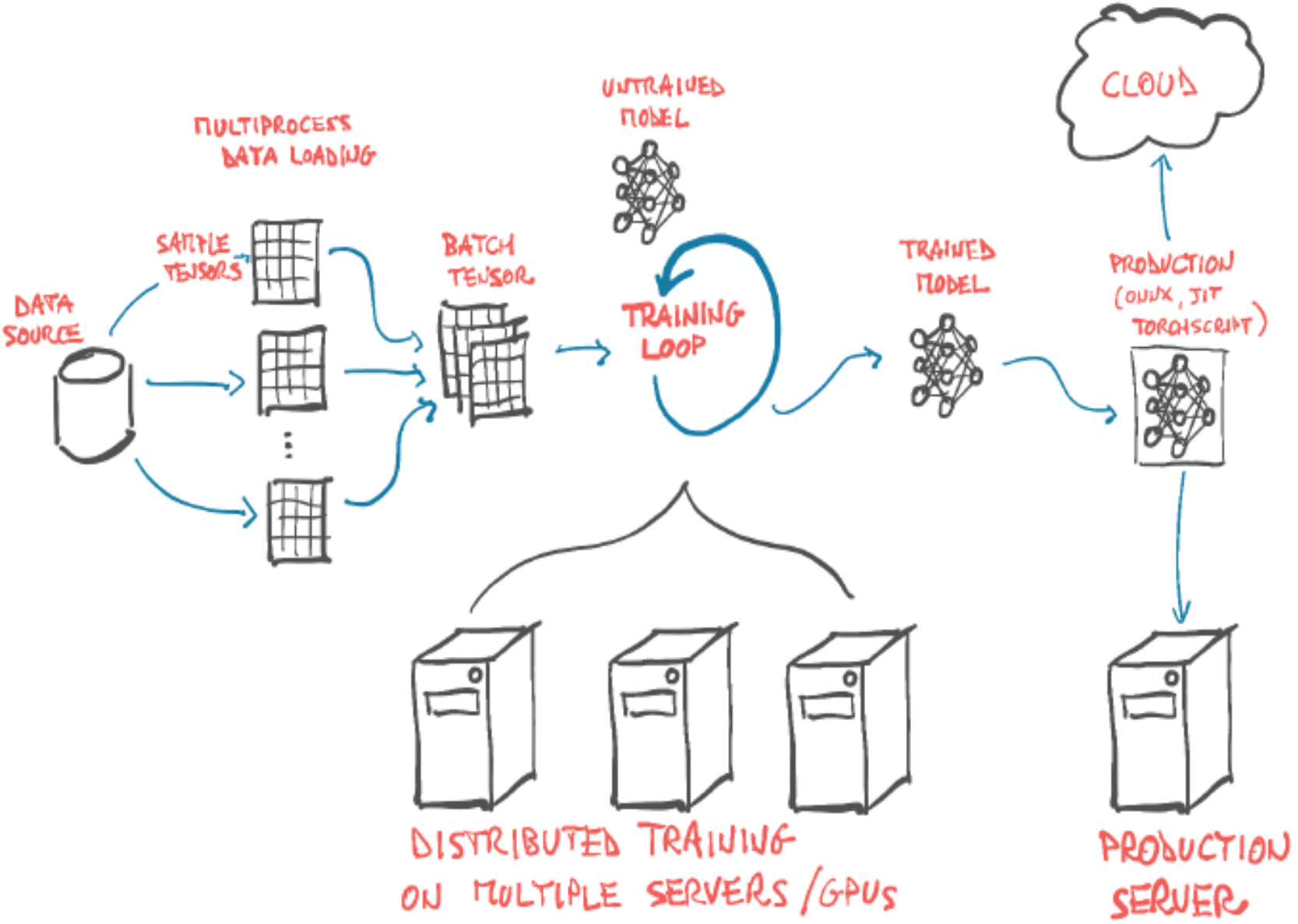

Figure 1.4 Basic, high-level structure of a PyTorch project, with data loading, training, and deployment to production.

The Main Components of PyTorch

- The core PyTorch modules for building neural networks are located in torch.nn, which provides common neural network layers and other architectural components.
- Fully connected layers, convolutional layers, activation functions, and loss functions can all be found here (we'll go into more detail about what all of that means as we go through the rest of this course). These components can be used to build and initialize the untrained model we see in the center of Figure-1.4.

The Main Components of PyTorch

- In order to train our model, we need a few things (besides the loop itself, which can just be a standard Python for loop): a source of training data, an optimizer to adapt the model to the training data, and a way to get the model and data to the hardware that will actually be performing the calculations needed for training the model.

The Main Components of PyTorch

- Utilities for data loading and handling can be found in torch.util.data.
- The two main classes we will work with are Dataset, which acts as the bridge between your custom data (in whatever format it might be in), and a standardized PyTorch Tensor. The other class we'll see often is the DataLoader, which can spawn child processes to load data from a Dataset in the background so that it's ready and waiting for the training loop as soon as the loop can use it.

Pre-Trained Networks

- Computer vision is one of the fields that have been most impacted by the advent of deep learning.
- We are going to learn how to use the work of the best researchers in the field by downloading and running very interesting models that have already been trained on open, large-scale datasets.
- We can consider a pre-trained neural network as similar to a program that has already been written. The behavior of such program is dictated by the architecture of the neural network and by the examples that were seen during training. Using an off-the-shelf model can be a quick way to jumpstart a deep learning project, since it leverages expertise from the researchers who designed the model, as well as the computation time that went into training it.

Pre-Trained Networks

- We will explore a popular pre-trained model that can label an image according to its contents.
- We will learn how to load and run a pre-trained model in PyTorch, along with some important skills.
- Learning how to run a pre-trained model using PyTorch is a useful skill, especially if the model has been trained on a large dataset. We will need to get accustomed to the mechanics of obtaining and running a neural network on real-world data, and then visualizing and evaluating its outputs, no matter if we trained it or not.

A pre-trained network that recognizes the subject of an image

- There are many pre-trained networks that can be accessed through source code repositories. Using one of these models could enable us to, for example, equip our next web-service with image recognition capabilities with very little effort.
- The pre-trained network we'll explore now has been trained on a subset of the ImageNet dataset (see imagenet.stanford.edu). ImageNet is a very large dataset of over 14 million images maintained by Stanford University. All images are labeled with a hierarchy of nouns coming from the WordNet dataset (see wordnet.princeton.edu), a large lexical database of the English language.

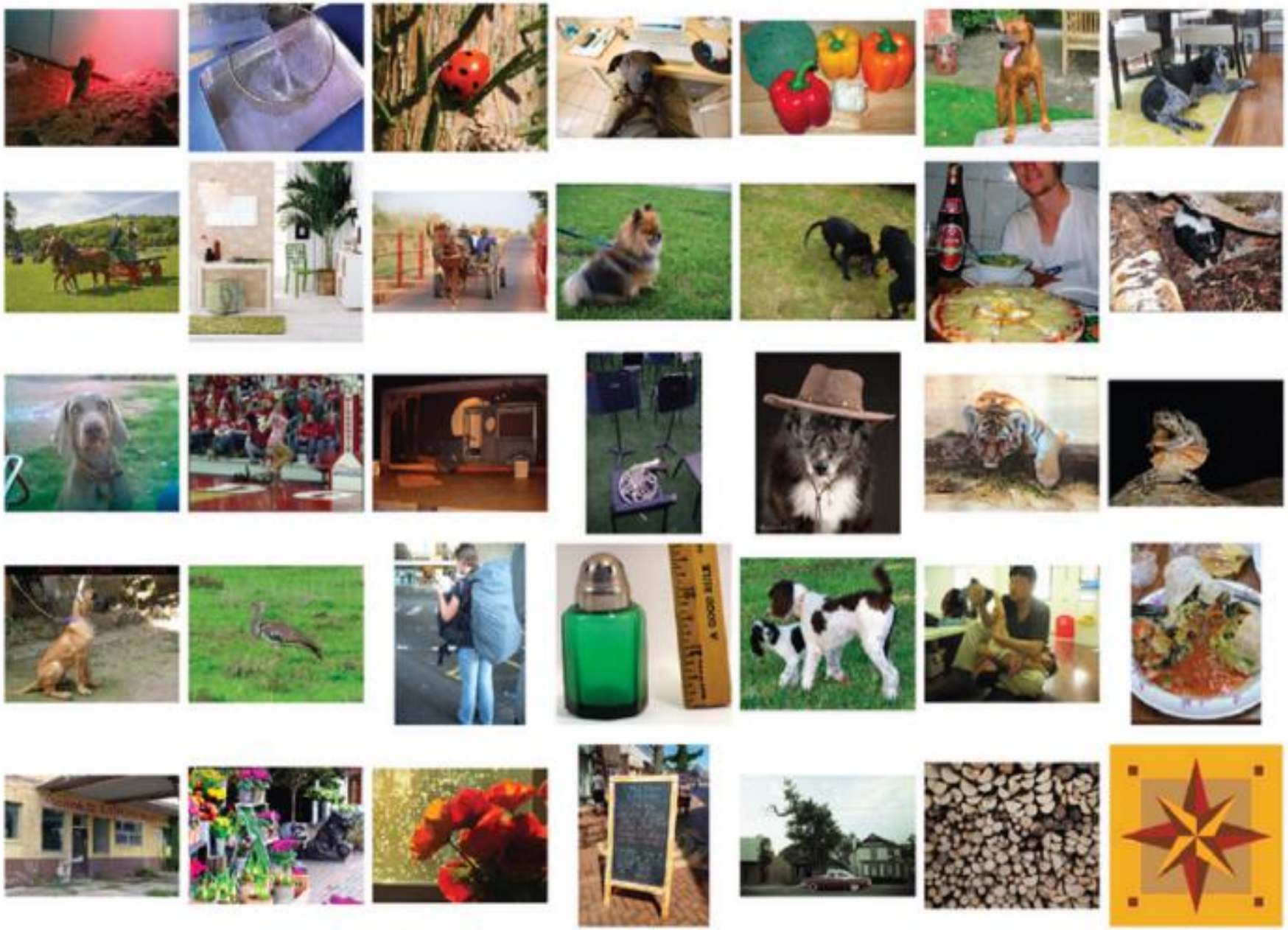

Figure 2.1 A small sample of ImageNet images

A pre-trained network that recognizes the subject of an image

- We will be able to take our own images and feed them into the pre-trained model, as pictured in Figure-2.2.
- This will result in a list of predicted labels for that image, which we can then examine to see what the model thinks our image is. Some images will have predictions that are accurate, some will not!

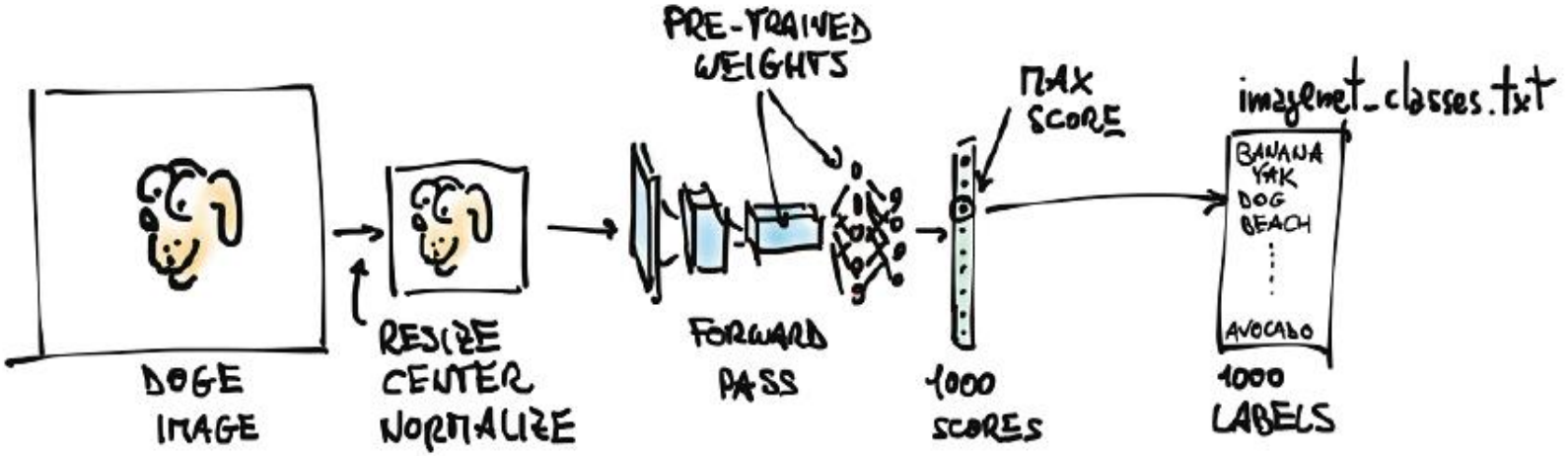

Figure 2.2 The inference process.

A pre-trained network that recognizes the subject of an image

- The ImageNet dataset, like several other public datasets, has its origin in academic competitions.
- Competitions have traditionally been one of the main playing fields where researchers at institutions and companies regularly challenge each other. Among others, the ImageNet Large Scale Visual Recognition Challenge (ILSVRC), which takes place on ImageNet, has gained popularity since its inception in 2010. This particular competition in based on a few tasks, which can vary by the year, such as image classification (telling what object categories the image contains), object localization (identifying objects' position in images), etc.

A pre-trained network that recognizes the subject of an image

- In particular, the image classification task consists of taking an image as input and producing a list of 5 labels out of 1000 total categories, ranked by confidence, describing the content of the image.
- The training set for ILSVRC consists of 1.2 million images, labeled with one of 1000 nouns (e.g. "dog"), referred to as the class or label of the image.

A pre-trained network that recognizes the subject of an image

- The input image will first be pre-processed into an instance of the multi-dimensional array class: torch.FloatTensor.
- We'll get into the details of what a tensor is later, but for now think of it as like a vector or matrix of floating-point numbers. Our model will take that processed input image and pass it into the pre-trained network to obtain scores for each label. The highest score corresponds to the most likely label.
- That output is contained in a torch.FloatTensor with 1000 elements, each representing the score associated with that label.

A pre-trained network that recognizes the subject of an image

- Before we can do all that, we'll need to get the network itself, take a peek under the hood and get a sense of how it's structured, and learn about how we need to prepare our data before the model can use it.
- We will use a network trained on ImageNet, taken from the TorchVision project, which contains a few of the best performing neural network architectures for computer vision, such as AlexNet and ResNet. TorchVision also has easy access to datasets like ImageNet and other utilities for getting up to speed with computer vision applications in PyTorch. See https://github.com/pytorch/vision.

A pre-trained network that recognizes the subject of an image

- We will load up and run the networks: AlexNet, one of early breakthrough networks for image recognition, and ResNet, which won the ImageNet classification, detection and localization competitions, in 2015.
- The pre-defined models can be found in torchvision.models.

```
In[1]: from torchvision import models
```

A pre-trained network that recognizes the subject of an image

- We can take a look at the actual models:

```
In[2]: dir(models)
Out[2]:
['AlexNet',
'DenseNet',
'Inception3',
'ResNet',
...
'alexnet',
'densenet',
...
'resnet',
'resnet101',
...
]
```

A pre-trained network that recognizes the subject of an image

- The capitalized names refer to Python classes that implement a number of popular models. They differ in their architecture, that is, in the arrangement of the operations occurring between the input and the output.
- The lowercase names are convenience functions that return models instantiated from those classes, sometimes with different parameter sets. For instance, resnet101 returns an instance of ResNet with 101 layers.
- We'll now look into AlexNet, designed by the SuperVision group, consisting of Alex Krizhevsky, Geoffrey Hinton, and Ilya Sutskever.

AlexNet

- The AlexNet architecture won the 2012 ImageNet Large-Scale Visual Recognition Challenge by a large margin, with a top 5 test error rate (i.e. correct label must be in the top 5 predictions) of 15.4%. By comparison, the second best submission, not based on a deep network, trailed at 26.2%. It was a defining moment in the history of computer vision, the moment when the community started to realize the potential of deep learning for vision tasks.
- That leap was followed by constant improvement, with more modern architectures and training methods getting top 5 error rates as low as 3%.

AlexNet

- We can see the structure of AlexNet in Figure-2.3.
- Each block consists of a bunch of multiplications and additions, plus a sprinkle of other functions on the output. We can think of it as a filter, a function that takes one or more images as input and produces other images in output. The way it does so is determined during training.
- Input images come in from the left and go through five stacks of filters, each producing a number of output images as reported in the figure. After each filter, images are reduced in size, as annotated. The images produced by the last stack of filters are layed out as a 4096-elements 1D vector and classified to produce 1000 output probabilities, one for each output class.

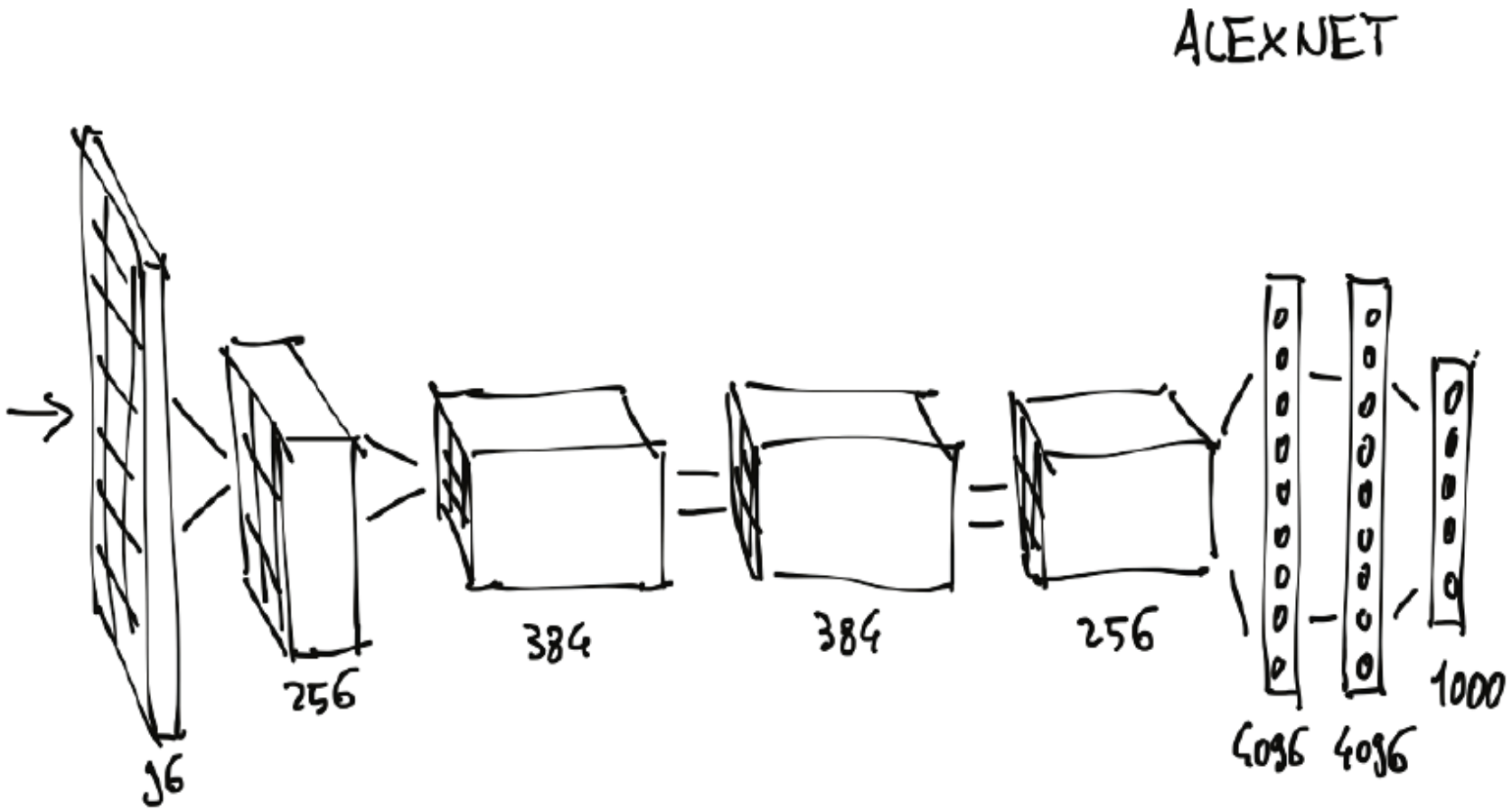

Figure 2.3 The AlexNet architecture.

AlexNet

- In order to run the AlexNet architecture on an input image we can create an instance of the AlexNet class. This is how it's done:

```
In[3]: alexnet = models.AlexNet()
```

AlexNet

- Here alexnet is an object that can run the AlexNet architecture.
- By providing alexnet with some precisely-sized input data, we will run a *forward pass* through the network. That is, the input will run through the first set of neurons, whose outputs will be fed to the next set of neurons, all the way to the output. Practically speaking, assuming we have an input object of the right type, we can run the forward pass with output = alexnet(input).
- We'd need to either train the network from scratch or load weights from a prior training, which we'll do now, using ResNet, an improved version of AlexNet.

ResNet

- Using the resnet101 function, we'll instantiate a 101-layer convolutional neural network.
- We'll pass an argument that will instruct the function to download the weights of a ResNet101 trained on the ImageNet dataset, with 1.2 million images and 1000 categories.

```
In[4]: resnet = models.resnet101(pretrained=True)
```

- It takes some time in downloading. Note that ResNet101 has 44.5 million parameters - that's a lot of parameters to optimize automatically!

ResNet

- We'll take a peek at what a ResNet101 looks like.
- We can do so just by printing the value of the returned model. This gives us a textual representation of the same kind of information we saw in Figure-2.3, which gives us some details about the structure of the network.
- What we are seeing here is the building blocks or layers of a neural network.
- That's the anatomy of a typical deep neural network for computer vision: a more or less sequential cascade of filters and non-linear functions, ending with a layer (fc) producing scores for each of the 1000 output classes (out_features).

```
# In[5]:
resnet

# Out[5]:
ResNet(
  (conv1): Conv2d(3, 64, kernel_size=(7, 7), stride=(2, 2), padding=(3, 3), bias=False)
  (bn1): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
  (relu): ReLU(inplace)
  (maxpool): MaxPool2d(kernel_size=3, stride=2, padding=1, dilation=1, ceil_mode=False)
  (layer1): Sequential(
    (0): Bottleneck(
...
    )
  )
  (avgpool): AvgPool2d(kernel_size=7, stride=1, padding=0)
  (fc): Linear(in_features=2048, out_features=1000, bias=True)
)
```

ResNet

- The resnet variable can be called like a function, taking in input one or more images and producing an equal number of scores for each of the 1000 ImageNet classes.
- Before we can do that, we have to pre-process any input image so that it has the right size and so that its values (its colors) sit roughly in the same numerical range. In order to do that, the torchvision module provides transforms, which allow to quickly define pipelines of basic pre-processing functions:

```
# In[6]:
from torchvision import transforms
preprocess = transforms.Compose([
        transforms.Resize(256),
        transforms.CenterCrop(224),
        transforms.ToTensor(),
        transforms.Normalize(
            mean=[0.485, 0.456, 0.406],
            std=[0.229, 0.224, 0.225]
        )])
```

ResNet

- In this case, we defined a preprocess function that will scale the input image to 256x256, crop the image to 224x224 around the center, transform it to a tensor, and normalize its RGB (red, green, blue) components so that they have defined means and standard deviations.
- These need to match what was presented to the network during training, if we wish that the network will produce meaningful answers.

ResNet

- We can now grab a picture of, for example, a dog (say, bobby.jpg from the GitHub repository), preprocess it, and then see what ResNet thinks of it.
- We can start by loading an image from the local filesystem using Pillow, an image manipulation module for Python:

```
# In[7]:
from PIL import Image
path = "C:/Kuliah/machineLearning2019/PyTorchCode/dlwpt/“
# In[8]:
img = Image.open(path + "data/p1ch2/bobby.jpg")
img.show()
```

ResNet

- We can now pass the image through our pre-processing pipeline:

```
# In[9]:
img_t = preprocess(img)
# In[10]:
import torch
batch_t = torch.unsqueeze(img_t, 0)
```

Running the Model

- The process of running a trained model on new data is called *inference* in deep learning talks.
- In order to do inference, we need to put the network in *eval* mode.

```
# In[11]:
resnet.eval()

# Out[11]:
ResNet(
  (conv1): Conv2d(3, 64, kernel_size=(7, 7), stride=(2, 2), padding=(3, 3), bias=False)
  (bn1): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
  (relu): ReLU(inplace)
  (maxpool): MaxPool2d(kernel_size=3, stride=2, padding=1, dilation=1, ceil_mode=False)
  (layer1): Sequential(
    (0): Bottleneck(
...
```

Running the Model

- Next we are ready for inference:

```
# In[12]:
out = resnet(batch_t)
out

# Out[12]:
tensor([[ -3.4803,  -1.6618,  -2.4515,  -3.2662,  -3.2466,  -1.3611,
          -2.0465,  -2.5112,  -1.3043,  -2.8900,  -1.6862,  -1.3055,
...
           2.8674,  -3.7442,   1.5085,  -3.2500,  -2.4894,  -0.3354,
           0.1286,  -1.1355,   3.3969,   4.4584]])
```

Running the Model

- A staggering set of operations involving 44.5 million parameters has just happened, producing a vector of 1000 scores, one per ImageNet class. Is it fast?
- We now need to find out what was the label of the class that received the highest score. This will tell us what the model saw in the image. If the label matches how a human would describe the image, that's great! It means everything is working. If not, then either something went wrong during training, or the image is so different from what the model expects that it can't process it properly, or there's some other similar issue.
- To see the list of predicted labels, we will load a text file listing the labels in the same order they were presented to the network during training, then pick out the label at the index that produced the highest score from the network.
- Almost all models meant for image recognition will have ouput in a form similar to what we're about to work with.

Running the Model

- Let's load the file containing the 1000 labels for the ImageNet dataset classes:

```
# In[13]:
with open(path+ 'data/p1ch2/imagenet_classes.txt') as f:
   labels = [line.strip() for line in f.readlines()]
```

- At this point we need to find out the index corresponding to the maximum score in the out tensor we obtained above. We can do that using the max function in PyTorch, which outputs the maximum value in a tensor as well as the indices where that maximum value occurred:

```
# In[14]:
_, index = torch.max(out, 1)
```

Running the Model

- We can now use the index to access the label. Here index is not a plain Python number, but a one-element 1-dimensional tensor (specifically tensor([207])), so we need to get the actual numerical value to use as an index into our labels list using index[0].
- We also use torch.nn.functional.softmax to normalize our outputs to the range [0, 1], and divide by the sum. That gives us something roughly akin to the confidence that the model has in its prediction. In this case, it's 96% certain that it knows what it's looking at is a golden retriever.

```
# In[15]:
percentage = torch.nn.functional.softmax(out, dim=1)[0] * 100
labels[index[0]], percentage[index[0]].item()
# Out[15]:
('golden retriever', 96.29334259033203)
```

Running the Model

- Since the model produced scores, we can also find out what second best, third best, and so on were. To do this, we can use the sort function, which sorts the values in ascending or descending order and also provides the indices of the sorted values in the original array:

```
# In[16]:
_, indices = torch.sort(out, descending=True)
[(labels[idx], percentage[idx].item()) for idx in indices[0][:5]]

# Out[16]:
[('golden retriever', 96.29334259033203),
 ('Labrador retriever', 2.80812406539917),
 ('cocker spaniel, English cocker spaniel, cocker', 0.282674282789230 35),
 ('redbone', 0.2086310237646103),
 ('tennis ball', 0.1162156909704 2084)]
```

Running the Model

- We see that the first four are dogs, then things start to get funny. The fifth answer of "tennis ball" is probably because there are enough pictures of tennis balls with dogs nearby that the model is essentially saying "there's a 0.1% chance that I've completely misunderstood what a tennis ball is."
- This is a great example of the fundamental differences in how humans and neural networks view the world, as well as how easy it is for strange, subtle biases to sneak into our data.
- We could go ahead and interrogate our network with random images and see what it comes up with. How successful the network will be will largely depend on whether the subjects were well represented in the training set.

- Here are all the commands that we have execute:

```
from torchvision import models
resnet = models.resnet101(pretrained=True)
from torchvision import transforms
preprocess = transforms.Compose([
  transforms.Resize(256),
  transforms.CenterCrop(224),
  transforms.ToTensor(),
  transforms.Normalize(
  mean=[0.485, 0.456, 0.406],
  std=[0.229, 0.224, 0.225]
)])
from PIL import Image
path = "C:/Kuliah/machineLearning2019/PyTorchCode/dlwpt/"
img = Image.open(path + "data/p1ch2/bobby.jpg")
img.show()
```

```
img_t = preprocess(img)
import torch
batch_t = torch.unsqueeze(img_t, 0)
resnet.eval()
out = resnet(batch_t)
with open(path+ 'data/p1ch2/imagenet_classes.txt') as f:
  labels = [line.strip() for line in f.readlines()]
_, index = torch.max(out, 1)
percentage = torch.nn.functional.softmax(out, dim=1)[0] *
100
labels[index[0]], percentage[index[0]].item()
_, indices = torch.sort(out, descending=True)
best5 = [(labels[idx], percentage[idx].item()) for idx in
indices[0][:5]]
```