**UNIT-3(CHAPTER-2)**

**What is Theano?**

Theano is an open source project released under the BSD license and was developed by the LISA (now [MILA](http://mila.umontreal.ca/)) group at the University of Montreal, Quebec, Canada (home of [Yoshua Bengio](http://www.iro.umontreal.ca/~bengioy/yoshua_en/index.html)). It is named after a [Greek mathematician](https://en.wikipedia.org/wiki/Theano_(philosopher)).

At it’s heart Theano is a compiler for mathematical expressions in Python. It knows how to take your structures and turn them into very efficient code that uses NumPy, efficient native libraries like [BLAS](http://www.netlib.org/blas/) and native code (C++) to run as fast as possible on CPUs or GPUs.

It uses a host of clever code optimizations to squeeze as much performance as possible from your hardware. If you are into the nitty-gritty of mathematical optimizations in code, [check out this interesting list](http://deeplearning.net/software/theano/optimizations.html#optimizations).

The actual syntax of Theano expressions is symbolic, which can be off putting to beginners used to normal software development. Specifically, expression are defined in the abstract sense, compiled and later actually used to make calculations.

It was specifically designed to handle the types of computation required for large neural network algorithms used in Deep Learning. It was one of the first libraries of its kind (development started in 2007) and is considered an industry standard for Deep Learning research and development.

**How to Install Theano**

Theano provides extensive installation instructions for the major operating systems: Windows, OS X and Linux. Read the [Installing Theano guide for your platform](http://deeplearning.net/software/theano/install.html).

Theano assumes a working Python 2 or Python 3 environment with [SciPy](https://www.scipy.org/). There are ways to make the installation easier, such as using [Anaconda](https://www.continuum.io/downloads) to quickly set up Python and SciPy on your machine as well as using [Docker images](http://deeplearning.net/software/theano/install.html" \l "docker-images).

With a working Python and SciPy environment, it is relatively straightforward to install Theano. from PyPI using pip, for example:

|  |  |
| --- | --- |
| 1 | pip install Theano |

At the time of writing the last official release of Theano was version 0.8 which was released 21th March 2016.

New releases may be announced and you will want to update to get any bug fixes and efficiency improvements. You can upgrade Theano using pip as follows:

|  |  |
| --- | --- |
| 1 | sudo pip install --upgrade --no-deps theano |

You may want to use the bleeding edge version of Theano checked directly out of Github.

This may be required for some wrapper libraries that make use of bleeding edge API changes. You can install Theano directly from a Github checkout as follows:

|  |  |
| --- | --- |
| 1 | pip install --upgrade --no-deps git+git://github.com/Theano/Theano.git |

You are now ready to run Theano on your CPU, which is just fine for the development of small models.

Large models may run slowly on the CPU. If you have a Nvidia GPU, you may want to look into configuring Theano to use your GPU. Read the [Using the GPU guides for Linux](http://deeplearning.net/software/theano/install.html#using-the-gpu) or [Mac OS X to set up Theano to use the GPU](http://deeplearning.net/software/theano/install.html#gpu-macos) and the [Using the GPU guide](http://deeplearning.net/software/theano/tutorial/using_gpu.html) for how to test whether it is working.

**Simple Theano Example**

In this section we demonstrate a simple Python script that gives you a flavor of Theano.

It is taken from the [Theano at a Glance guide](http://deeplearning.net/software/theano/introduction.html). In this example we define two symbolic floating point variables a and b.

We define an expression that uses these variables (c = a + b).

We then compile this symbolic expression into a function using Theano that we can use later.

Finally, we use our complied expression by plugging in some real values and performing the calculation using efficient compiled Theano code under the covers.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12 | import theano  from theano import tensor  # declare two symbolic floating-point scalars  a = tensor.dscalar()  b = tensor.dscalar()  # create a simple expression  c = a + b  # convert the expression into a callable object that takes (a,b)  # values as input and computes a value for c  f = theano.function([a,b], c)  # bind 1.5 to 'a', 2.5 to 'b', and evaluate 'c'  assert 4.0 == f(1.5, 2.5) |

Running the example does not provide any output. The assertion that 1.5 + 2.5 = 4.0 is true.

This is a useful example as it gives you a flavor for how a symbolic expression can be defined, compiled and used. You can see how this may be scaled up to large vector and matrix operations required for deep learning.

**Extensions and Wrappers for Theano**

If you are new to deep learning you do not have to use Theano directly.

In fact, you are highly encouraged to use one of many popular Python projects that make Theano a lot easier to use for deep learning.

These projects provide data structures and behaviors in Python, specifically designed to quickly and reliably create deep learning models whilst ensuring that fast and efficient models are created and executed by Theano under the covers.

The amount of Theano syntax exposed by the libraries varies.

* For example the [Lasagne library](http://lasagne.readthedocs.org/en/latest/) provides convenience classes for creating deep learning model but still expects you to know and make use of Theano syntax. This is good for beginners that know or are willing to learn a little Theano as well.
* Another example is [Keras](http://keras.io/) that hides Theano completely and provides a very simple API to work with to create Deep Learning models. It hides Theano so well, that it can in fact run as a wrapper for another popular foundation framework called [TensorFlow](https://www.tensorflow.org/).

I highly recommend experimenting a little with Theano directly, then selecting a wrapper library to learn and practice deep learning.

For a full list of libraries built on Theano see the [Related projects guide](https://github.com/Theano/Theano/wiki/Related-projects) on the Theano Wiki.

## What is Microsoft Cognitive Toolkit (CNTK)?

Microsoft Cognitive Toolkit (CNTK), formerly known as Computational Network Toolkit, is a free, easy-to-use, open-source, commercial-grade toolkit that enables us to train deep learning algorithms to learn like the human brain. It enables us to create some popular deep learning systems like **feed-forward neural network time series prediction systems and Convolutional neural network (CNN) image classifiers**.

For optimal performance, its framework functions are written in C++. Although we can call its function using C++, but the most commonly used approach for the same is to use a Python program.

## CNTK’s Features

Following are some of the features and capabilities offered in the latest version of Microsoft CNTK:

### Built-in components

* CNTK has highly optimised built-in components that can handle multi-dimensional dense or sparse data from Python, C++ or BrainScript.
* We can implement CNN, FNN, RNN, Batch Normalisation and Sequence-to-Sequence with attention.
* It provides us the functionality to add new user-defined core-components on the GPU from Python.
* It also provides automatic hyperparameter tuning.
* We can implement Reinforcement learning, Generative Adversarial Networks (GANs), Supervised as well as Unsupervised learning.
* For massive datasets, CNTK has built-in optimised readers.

### Usage of resources efficiently

* CNTK provides us parallelism with high accuracy on multiple GPUs/machines via 1-bit SGD.
* To fit the largest models in GPU memory, it provides memory sharing and other built-in methods.

### Express our own networks easily

* CNTK has full APIs for defining your own network, learners, readers, training and evaluation from Python, C++, and BrainScript.
* Using CNTK, we can easily evaluate models with Python, C++, C# or BrainScript.
* It provides both high-level as well as low-level APIs.
* Based on our data, it can automatically shape the inference.
* It has fully optimised symbolic Recurrent Neural Network (RNN) loops.

### Measuring model performance

* CNTK provides various components to measure the performance of neural networks you build.
* Generates log data from your model and the associated optimiser, which we can use to monitor the training process.

## Version 1.0 vs Version 2.0

Following table compares CNTK Version 1.0 and 2.0:

|  |  |
| --- | --- |
| **Version 1.0** | **Version 2.0** |
| It was released in 2016. | It is a significant rewrite of the 1.0 Version and was released in June 2017. |
| It used a proprietary scripting language called BrainScript. | Its framework functions can be called using C++, Python. We can easily load our modules in C# or Java. BrainScript is also supported by Version 2.0. |
| It runs on both Windows and Linux systems but not directly on Mac OS. | It also runs on both Windows (Win 8.1, Win 10, Server 2012 R2 and later) and Linux systems but not directly on Mac OS. |

## Important Highlights of Version 2.7

**Version 2.7** is the last main released version of Microsoft Cognitive Toolkit. It has full support for ONNX 1.4.1. Following are some important highlights of this last released version of CNTK.

* Full support for ONNX 1.4.1.
* Support for CUDA 10 for both Windows and Linux systems.
* It supports advance Recurrent Neural Networks (RNN) loop in ONNX export.
* It can export more than 2GB models in ONNX format.
* It supports FP16 in BrainScript scripting language’s training action.
* Here, we will understand about the installation of CNTK on Windows and on Linux. Moreover, the chapter explains installing CNTK package, steps to install Anaconda, CNTK files, directory structure and CNTK library organisation.

## Prerequisites

* In order to install CNTK, we must have Python installed on our computers. You can go to the link <https://www.python.org/downloads/> and select the latest version for your OS, i.e. Windows and Linux/Unix. For basic tutorial on Python, you can refer to the link <https://www.tutorialspoint.com/python3/index.htm>.
* CNTK is supported for Windows as well as Linux so we will walk through both of them.

## Installing on Windows

* In order to run CNTK on Windows, we will be using the **Anaconda version** of Python. We know that, Anaconda is a redistribution of Python. It includes additional packages like **Scipy** and**Scikit-learn** which are used by CNTK to perform various useful calculations.
* So, first let see the steps to install Anaconda on your machine −
* **Step 1**−First download the setup files from the public website <https://www.anaconda.com/distribution/>.
* **Step 2** − Once you downloaded the setup files, start the installation and follow the instructions from the link <https://docs.anaconda.com/anaconda/install/>.
* **Step 3** − Once installed, Anaconda will also install some other utilities, which will automatically include all the Anaconda executables in your computer PATH variable. We can manage our Python environment from this prompt, can install packages and run Python scripts.

## Installing CNTK package

* Once Anaconda installation is done, you can use the most common way to install the CNTK package through the pip executable by using following command −
* pip install cntk
* There are various other methods to install Cognitive Toolkit on your machine. Microsoft has a neat set of documentation that explains the other installation methods in detail. Please follow the link <https://docs.microsoft.com/en-us/cognitive-toolkit/Setup-CNTK-on-your-machine>.

## Installing on Linux

* Installation of CNTK on Linux is a bit different from its installation on Windows. Here, for Linux we are going to use Anaconda to install CNTK, but instead of a graphical installer for Anaconda, we will be using a terminal-based installer on Linux. Although, the installer will work with almost all Linux distributions, we limited the description to Ubuntu.
* So, first let see the steps to install Anaconda on your machine −

### Steps to install Anaconda

* **Step 1** − Before installing Anaconda, make sure that the system is fully up to date. To check, first execute the following two commands inside a terminal −
* sudo apt update
* sudo apt upgrade
* **Step 2**− Once the computer is updated, get the URL from the public website <https://www.anaconda.com/distribution/> for the latest Anaconda installation files.
* **Step 3** − Once URL is copied, open a terminal window and execute the following command −
* wget -0 anaconda-installer.sh url SHAPE \\* MERGEFORMAT
* y

* f

* x
* | }
* Replace the **url** placeholder with the URL copied from the Anaconda website.
* **Step 4** − Next, with the help of following command, we can install Anaconda −
* sh ./anaconda-installer.sh
* The above command will by default install **Anaconda3** inside our home directory.

## Installing CNTK package

* Once Anaconda installation is done, you can use the most common way to install the CNTK package through the pip executable by using following command −
* pip install cntk

## Examining CNTK files & directory structure

* Once CNTK is installed as a Python package, we can examine its file and directory structure. It’s at **C:\Users\\Anaconda3\Lib\site-packages\cntk,** as shown below in screenshot.

## Verifying CNTK installation

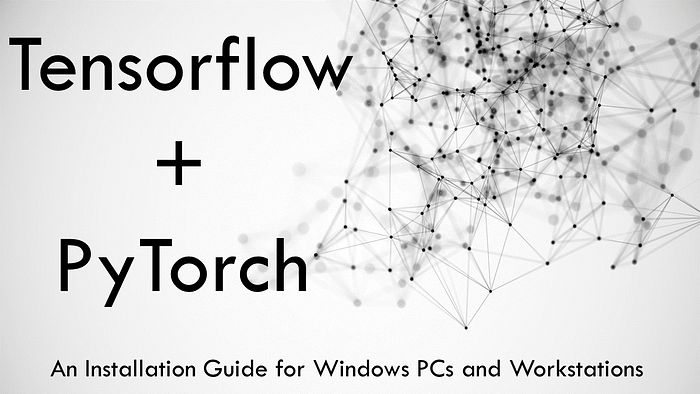
* Once CNTK is installed as a Python package, you should verify that CNTK has been installed correctly. From Anaconda command shell, start Python interpreter by entering **ipython.** Then, import **CNTK** by entering the following command.
* import cntk as c
* Once imported, check its version with the help of following command −
* print(c.\_\_version\_\_)
* The interpreter will respond with installed CNTK version. If it doesn’t respond, there will be a problem with the installation.

## The CNTK library organisation

* CNTK, a python package technically, is organised into 13 high-level sub-packages and 8 smaller sub-packages. Following table consist of the 10 most frequently used packages:

|  |  |
| --- | --- |
| **Sr.No** | **Package Name & Description** |
| 1 | **cntk.io**  Contains functions for reading data. For example: next\_minibatch() |
| 2 | **cntk.layers**  Contains high-level functions for creating neural networks. For example: Dense() |
| 3 | **cntk.learners**  Contains functions for training. For example: sgd() |
| 4 | **cntk.losses**  Contains functions to measure training error. For example: squared\_error() |
| 5 | **cntk.metrics**  Contains functions to measure model error. For example: classificatoin\_error |
| 6 | **cntk.ops**  Contains low-level functions for creating neural networks. For example: tanh() |
| 7 | **cntk.random**  Contains functions to generate random numbers. For example: normal() |
| 8 | **cntk.train**  Contains training functions. For example: train\_minibatch() |
| 9 | **cntk.initializer**  Contains model parameter initializers. For example: normal() and uniform() |
| 10 | **cntk.variables**  Contains low-level constructs. For example: Parameter() and Variable() |

**Setting up your PC/Workstation for Deep Learning: Tensorflow and PyTorch — Windows**



# Minimum Hardware and Software Requirements

You definitely need an Nvidia GPU to follow along if you’re planning to set it up with GPU support.

Developing Deep Learning applications involves training neural networks, which are compute-hungry by nature. It is also by nature more and more parallelization friendly which takes us more and more towards GPUs which are good at exactly that. That is why GPUs come in handy, the vast majority of the deep learning frameworks support GPU-acceleration out of the box so developers and researchers can get productive in minutes without doing any GPU programming which can tend to hold them back.

Most of these frameworks (only) support something called CUDA — which can only work with Nvidia GPUs, that’s why you specifically need one from Nvidia. However, it is not impossible on AMD’s cards, for more information visit [this page](https://rocmdocs.amd.com/en/latest/).

If you’re not going to set up for GPU, no problem you can still follow along.

But of course, you should have a decent CPU, RAM and Storage to be able to do some Deep Learning.

My hardware — I set this up on my personal laptop which has the following configuration,

**CPU**— AMD Ryzen 7 4800HS 8C -16T@ 4.2GHz on Turbo.

**RAM —**16 GB DDR4 RAM@ 3200MHz

**GPU —**Nvidia GeForce RTX 2060 Max-Q @ 6GB GDDR6 Memory

For anyone who is interested in knowing about the configurations, I recommend a decent CPU with a minimum of 4 cores and at 2.6GHz, at least 16GB of RAM and an Nvidia GPU with at least of 6GB VRAM.

For this tutorial, you obviously need to be on Windows 10. I assume some basic knowledge of Python packages and environments. Nonetheless, it will all be explained.

It is recommended that you’re in the latest stable build of 64-bit Windows 10.

This tutorial also assumes that you start the installation process on a freshly installed Operating System. If not you can still follow along if you really know what you’re doing.

# Installing Python and required tools

The first step is, of course, to install Python. I recommend installing Python through Mini-Conda. For absolute beginners, I will briefly explain why.

Conda is a package manager that helps you with installing, managing and removing all your different packages. It is not the only one though, there is pip — python’s default package manager which I really like. Here we go with conda as it is much more straight-forward and simple to set up in Windows.

Anaconda and Mini-Conda are software distributions that come with some very useful Data Science/ML packages preinstalled to save some of your time. Anaconda contains over 150 packages that help in doing Data Science and Machine Learning, which includes everything you might ever need whereas Mini-Conda only comes with a handful of really necessary tools and packages.

I recommend going with Mini-Conda because I like to have (almost) complete control over what packages get installed. Keeping things light is indeed not a bad thing at all. It can save you some storage space and of course, you’ll not have some 50 odd packages that you probably never use.

To install Mini-Conda, go to this link below, <https://docs.conda.io/en/latest/miniconda.html>

Download the Python3 installers for Windows 64-bit and install it as you’d install any other Windows software. Make sure to tick the checkbox which asks if you want conda and python to be added to PATH.

Now you can check if you have python and conda installed by running the following commands. They should display the version numbers otherwise you might need to correctly install mini-conda and add it to PATH.

> python --version  
Python 3.8.3> conda --version  
conda 4.8.4

Next step is to install jupyter-notebook, paste the following command in your command-line interface,

> conda install -y jupyter

You can verify your installation by running the jupyter notebook, which opens up jupyter notebook for you on the browser.

> jupyter notebook

## Setting up your Development Environment

This is an important step often missed by many people. It is understandable to use something that contains every known package like Anaconda but to work on your projects and actually build stuff you probably need a custom environment specific to that project or the nature of the work you’re doing. Another great advantage of using a dedicated virtual environment is you can isolate your packages from interacting with global settings which means if by any chance you mess up the packages in your environment you can always throw it away without affecting any of the global packages.

It also gives you the flexibility to create environments with any python version backwards in time. So you can stay away from all the new unstable stuff for a while and upgrade later based on support.

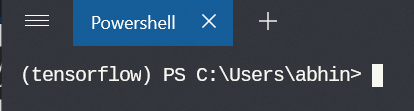
Creating a conda environment is fairly simple, I am creating it with the name tensorflow for the sake of explainability, you can set it to whatever you want. I am going with python 3.7 because I know it is well supported by Tensorflow. You can verify this through their [documentation](https://www.tensorflow.org/install/pip). By the way, this is where we will end up installing Tensorflow and create a similar environment with the name torch where PyTorch will be installed.

> conda create --name tensorflow python=3.7

Once the environment is created you can enter in using this command below, where tensorflow just means the name we gave to this environment earlier.

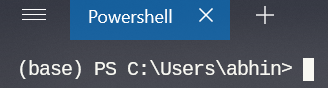
> conda activate tensorflow

Once you enter inside the environment you’ll be able to see something like this one the left-hand side of the prompt



If by any chance you don’t see this on Powershell, you might want to initialize conda in Powershell only once beforehand,

> conda init powershell



After that, you might see (base) on the left side as seen in the above image when you’re not inside any environments. You’ll see the env name whenever you enter an env after this.

Additionally, you can also install nb tools inside this environment and link it with jupyter notebook which we installed earlier

> conda install nb\_conda

To register the environment with Jupyter Notebooks run this command without breaking lines,

> python -m ipykernel install --user --name tensorflow --display-name “Python 3.7 (tensorflow)”

To exit out of the conda environment…

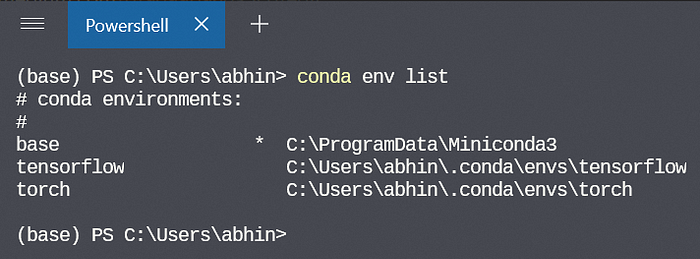
> conda deactivate

Now follow the same steps to create an environment with the name torch

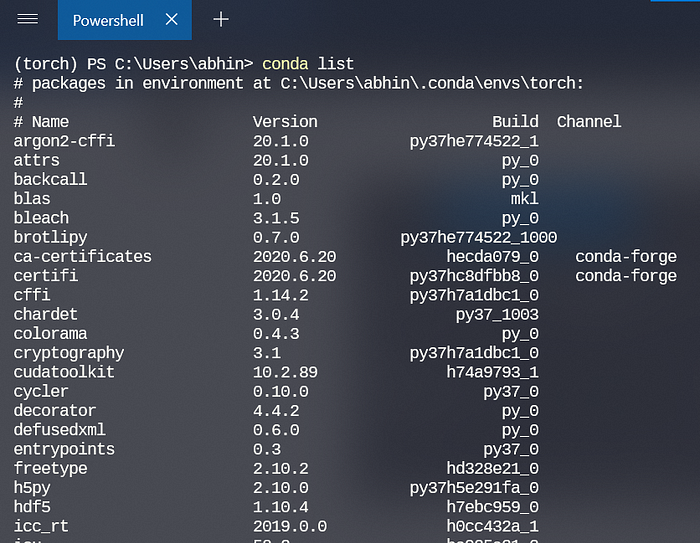
> conda create --name torch python=3.7> conda activate torch> conda install nb\_conda> python -m ipykernel install --user --name torch --display-name “Python 3.7 (torch)”

If the environments are successfully set up you would be able to see this while listing the environments,

> conda env list



To verify if the respective packages are actually installed inside each environment, you can enter into the environment and do conda list which displays the list of all packages installed in that environment.



Don’t worry if the list is pretty long. Conda has already taken care of the bare bone stuff and dependencies

# Some GPU jargon

Before jumping in and installing a few GPU related stuff, it is necessary to understand what is what and why you’ll need these things in place.

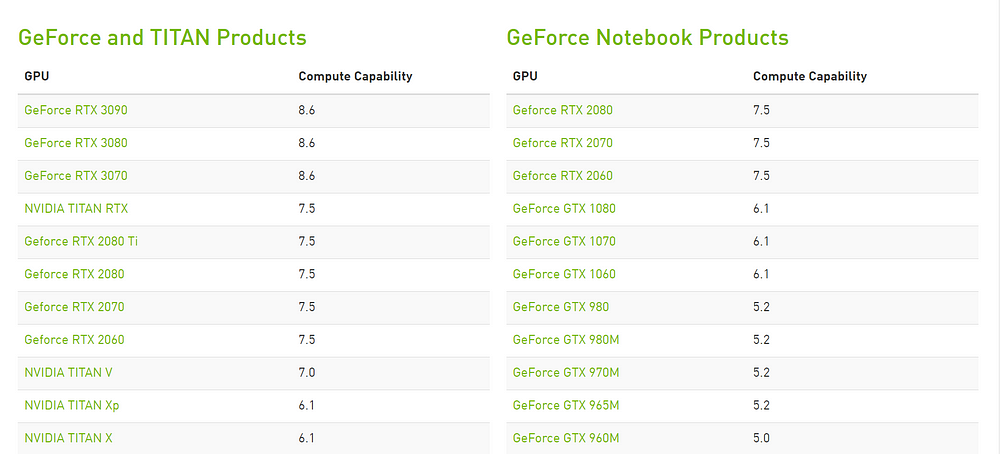
**GPU Drivers —**As the name suggests GPU driver is a piece of software that allows your Operating System and its programs to use the GPU hardware. Gamers certainly know this better, if you’re into gaming then you probably need to have this software up to date for the best experience.

**CUDA —**In simple terms**,**it is a programming interface layer developed by Nvidia that gives access to the GPU’s instruction set and its parallel computation units.

Since the GeForce 8 series of GPUs from the late 2010s, almost all GPUs are CUDA capable.

For more information, you can visit [this](https://developer.nvidia.com/cuda-gpus) page from Nvidia’s website to know if your GPU is CUDA enabled or not.

If you own a consumer GPU, for example, something from the GeForce lineup or Titan lineup, for instance, you can see a glimpse of what is supported and what is not in the image below,



As of Sept. 2020 (This is just the upper half of the list)

If you own a laptop you should check the Notebook products list and if you own a full-blown desktop GPU you must obviously look for the other lineup on the left-hand side.

As mentioned before, I own an RTX 2060 Max-Q which is listed on the right side. By the way, you don’t have to worry about your card’s title exactly matching what’s listed in the webpage, Max-Q and Super are all just designs sharing the same underlying architecture with some differences in TDP, number of CUDA cores and Tensor cores.

For instance, if you own an RTX 2080 Super or 2080 Max-Q or even a 2080 Super Max-Q — it is completely fine if you can just find RTX 2080 in that list. But if your own an RTX 2080Ti or something with Ti at the end, it just means you own the highest-end variant from that particular series, so most probably you’ll find it up there with more capability in terms of VRAM and number of CUDA and Tensor cores.

# Installing PyTorch

Now that we have covered how to install Tensorflow, installing PyTorch is nothing different. Conda makes the whole process surprisingly simple.

First, you should enter into the conda environment we created for torch.

> conda activate torch

If you want to install PyTorch with CUDA support use the following command,

> conda install pytorch torchvision cudatoolkit -c pytorch

The above command will install PyTorch with the compatible CUDA toolkit through the PyTorch channel in Conda.

To install PyTorch for CPU-only, you can just remove cudatookit from the above command

> conda install pytorch torchvision cpuonly -c pytorch

This installs PyTorch without any CUDA support.

You can verify the installation using > conda list as discussed before. To verify it on Python, use the following lines of code

>>> import torch  
>>> torch.\_\_version\_\_  
'1.6.0'

If it returns back the version number, you’re done installing PyTorch.

# Validating your Installation

You might think well everything is right and start using these tools but suddenly when you do that you’ll start to see some fatal errors. If by any chance this happens to you, it might be very specific to your machine and the way you have set things up which is too much for me to cover here before actually getting to know a lot more information specific to your case.

So, I provide a couple of notebooks to at least help you better validate the installations and make sure TF or PyTorch is making use of the intended hardware.

You can find the notebooks in [this repository](https://github.com/abhinand5/blog-posts) under the folder [dl-setup-win](https://github.com/abhinand5/blog-posts/tree/master/dl-setup-win). It is up to you to clone the notebook and run the cells. If it displays the correct information then you’re good to go.

I have embedded the gist version of the same notebooks below.

**Note:**You might face some errors if you do not launch the jupyter notebook from the correct environment. If you want to use tensorflow environment for example, you can launch the notebook from base env and change your kernel to tensorflow env but I have experienced errors while doing that. So to run TF launch your notebook from tensorflow environment and to run PyTorch launch your notebook from torch environment and not from base or something else.

**Binary Classification :**

It is a process or task of classification, in which a given data is being classified into two

classes. It’s basically a kind of prediction about which of two groups the thing belongs to.

➔ **Multi-output classification** is a type of machine learning that predicts multiple outputs

simultaneously. In multi-output classification, the model will give two or more outputs

after making any prediction. In other types of classifications, the model usually predicts

only a single output.

➔ In multi-label classification, **zero or more labels are required as output for each input**

**sample, and the outputs are required simultaneously**. The assumption is that the

output labels are a function of the inputs.

➔ Let us suppose, two emails are sent to you, one is sent by an insurance company that

keeps sending their ads, and the other is from your bank regarding your credit card bill.

The email service provider will classify the two emails, the first one will be sent to the

spam folder and the second one will be kept in the primary one.

**Binary vs Multiclass Classification Parameters Binary classification Multi-class classification No. of classes**

It is a classification of twogroups, i.e. classifies objects in at most two classes. There can be any number of classes in it, i.e., classifies the object into more than two classes.

**Algorithms used**

The most popular algorithms used by the binary classification are-

• Logistic Regression

Popular algorithms that can be used for multi-class classification include:

• k-Nearest Neighbors

• Decision Trees

• Naive Bayes

• k-Nearest Neighbors

• Decision Trees

• Support Vector Machine

• Naive Bayes

• Random Forest.

• Gradient Boosting

**Examples**

Examples of binary classification include-

• Email spam detection (spam or not).

• Churn prediction (churn or not).

• Conversion prediction (buy or not).

Examples of multi-class classification include:

• Face classification.

• Plant species classification.

• Optical character recognition.

**->** This process is known as binary classification, as there are two discrete classes, one is

spam and the other is primary. So, this is a problem of binary classification.

**->** Binary classification uses some algorithms to do the task, some of the most common

algorithms used by binary classification.

**->**Multilabel classification is an important subfield of structured output prediction where

multiple labels must be assigned that respect semantic relationships such as subsumption,

mutual exclusion or weak forms of correlation.

**Classifying movie reviews: a binary classification example**

Two-class classification, or binary classification, may be the most widely applied kind of machine learning problem. In this example, we will learn to classify movie reviews into “positive” reviews and “negative” reviews, just based on the text content of the reviews.

## The IMDB dataset

We’ll be working with “IMDB dataset”, a set of 50,000 highly-polarized reviews from the Internet Movie Database. They are split into 25,000 reviews for training and 25,000 reviews for testing, each set consisting in 50% negative and 50% positive reviews.

Why do we have these two separate training and test sets? You should never test a machine learning model on the same data that you used to train it! Just because a model performs well on its training data doesn’t mean that it will perform well on data it has never seen, and what you actually care about is your model’s performance on new data (since you already know the labels of your training data – obviously you don’t need your model to predict those). For instance, it is possible that your model could end up merely memorizing a mapping between your training samples and their targets – which would be completely useless for the task of predicting targets for data never seen before. We will go over this point in much more detail in the next chapter.

Just like the MNIST dataset, the IMDB dataset comes packaged with Keras. It has already been preprocessed: the reviews (sequences of words) have been turned into sequences of integers, where each integer stands for a specific word in a dictionary.

The following code will load the dataset (when you run it for the first time, about 80MB of data will be downloaded to your machine):

**library**(keras)

imdb <- dataset\_imdb(num\_words = 10000)

c(c(train\_data, train\_labels), c(test\_data, test\_labels)) %<-% imdb

The argument num\_words = 10000 means that we will only keep the top 10,000 most frequently occurring words in the training data. Rare words will be discarded. This allows us to work with vector data of manageable size.

The variables train\_data and test\_data are lists of reviews, each review being a list of word indices (encoding a sequence of words). train\_labels and test\_labels are lists of 0s and 1s, where 0 stands for “negative” and 1 stands for “positive”:

str(train\_data[[1]])

## int [1:218] 1 14 22 16 43 530 973 1622 1385 65 ...

train\_labels[[1]]

## [1] 1

Since we restricted ourselves to the top 10,000 most frequent words, no word index will exceed 10,000:

max(sapply(train\_data, max))

## [1] 9999

For kicks, here’s how you can quickly decode one of these reviews back to English words:

# word\_index is a dictionary mapping words to an integer index

word\_index <- dataset\_imdb\_word\_index()

# We reverse it, mapping integer indices to words

reverse\_word\_index <- names(word\_index)

names(reverse\_word\_index) <- word\_index

# We decode the review; note that our indices were offset by 3

# because 0, 1 and 2 are reserved indices for "padding", "start of sequence", and "unknown".

decoded\_review <- sapply(train\_data[[1]], **function**(index) {

word <- **if** (index >= 3) reverse\_word\_index[[as.character(index - 3)]]

**if** (!is.null(word)) word **else** "?"

})

cat(decoded\_review)

## ? this film was just brilliant casting location scenery story direction everyone's really suited the part they played and you could just imagine being there robert ? is an amazing actor and now the same being director ? father came from the same scottish island as myself so i loved the fact there was a real connection with this film the witty remarks throughout the film were great it was just brilliant so much that i bought the film as soon as it was released for ? and would recommend it to everyone to watch and the fly fishing was amazing really cried at the end it was so sad and you know what they say if you cry at a film it must have been good and this definitely was also ? to the two little boy's that played the ? of norman and paul they were just brilliant children are often left out of the ? list i think because the stars that play them all grown up are such a big profile for the whole film but these children are amazing and should be praised for what they have done don't you think the whole story was so lovely because it was true and was someone's life after all that was shared with us all

## Preparing the data

You can’t feed lists of integers into a neural network. You have to turn your lists into vectors. There are two ways to do that:

* Pad your lists so that they all have the same length, turn them into an integer tensor of shape (samples, word\_indices), and then use as the first layer in your network a layer capable of handling such integer tensors (the “embedding” layer, which we’ll cover in detail later in the book).
* One-hot-encode your lists to turn them into vectors of 0s and 1s. This would mean, for instance, turning the sequence [3, 5] into a 10,000-dimensional vector that would be all zeros except for indices 3 and 5, which would be ones. Then you could use as the first layer in your network a dense layer, capable of handling floating-point vector data.

Let’s go with the latter solution and vectorize the data, which you’ll do manually for maximum clarity.

vectorize\_sequences <- **function**(sequences, dimension = 10000) {

# Create an all-zero matrix of shape (len(sequences), dimension)

results <- matrix(0, nrow = length(sequences), ncol = dimension)

**for** (i **in** 1:length(sequences))

# Sets specific indices of results[i] to 1s

results[i, sequences[[i]]] <- 1

results

}

# Our vectorized training data

x\_train <- vectorize\_sequences(train\_data)

# Our vectorized test data

x\_test <- vectorize\_sequences(test\_data)

Here’s what our samples look like now:

str(x\_train[1,])

## num [1:10000] 1 1 0 1 1 1 1 1 1 0 ...

We should also vectorize our labels, which is straightforward:

# Our vectorized labels

y\_train <- as.numeric(train\_labels)

y\_test <- as.numeric(test\_labels)

Now our data is ready to be fed into a neural network.

## Building our network

The input data is vectors, and the labels are scalars (1s and 0s): this is the easiest setup you’ll ever encounter. A type of network that performs well on such a problem is a simple stack of fully connected (“dense”) layers with relu activations: layer\_dense(units = 16, activation = "relu").

The argument being passed to each dense layer (16) is the number of hidden units of the layer. A hidden unit is a dimension in the representation space of the layer. You may remember from chapter 2 that each such dense layer with a relu activation implements the following chain of tensor operations:

output = relu(dot(W, input) + b)

Having 16 hidden units means that the weight matrix W will have shape (input\_dimension, 16), i.e. the dot product with W will project the input data onto a 16-dimensional representation space (and then we would add the bias vector b and apply the relu operation). You can intuitively understand the dimensionality of your representation space as “how much freedom you are allowing the network to have when learning internal representations”. Having more hidden units (a higher-dimensional representation space) allows your network to learn more complex representations, but it makes your network more computationally expensive and may lead to learning unwanted patterns (patterns that will improve performance on the training data but not on the test data).

There are two key architecture decisions to be made about such stack of dense layers:

* How many layers to use.
* How many “hidden units” to chose for each layer.

In the next chapter, you will learn formal principles to guide you in making these choices. For the time being, you will have to trust us with the following architecture choice: two intermediate layers with 16 hidden units each, and a third layer which will output the scalar prediction regarding the sentiment of the current review. The intermediate layers will use relu as their “activation function”, and the final layer will use a sigmoid activation so as to output a probability (a score between 0 and 1, indicating how likely the sample is to have the target “1”, i.e. how likely the review is to be positive). A relu (rectified linear unit) is a function meant to zero-out negative values, while a sigmoid “squashes” arbitrary values into the [0, 1] interval, thus outputting something that can be interpreted as a probability.

Here’s what our network looks like:

3-layer network

And here’s the Keras implementation, very similar to the MNIST example you saw previously:

**library**(keras)

model <- keras\_model\_sequential() %>%

layer\_dense(units = 16, activation = "relu", input\_shape = c(10000)) %>%

layer\_dense(units = 16, activation = "relu") %>%

layer\_dense(units = 1, activation = "sigmoid")

Lastly, we need to pick a loss function and an optimizer. Since we are facing a binary classification problem and the output of our network is a probability (we end our network with a single-unit layer with a sigmoid activation), is it best to use the binary\_crossentropy loss. It isn’t the only viable choice: you could use, for instance, mean\_squared\_error. But crossentropy is usually the best choice when you are dealing with models that output probabilities. Crossentropy is a quantity from the field of Information Theory, that measures the “distance” between probability distributions, or in our case, between the ground-truth distribution and our predictions.

Here’s the step where we configure our model with the rmsprop optimizer and the binary\_crossentropy loss function. Note that we will also monitor accuracy during training.

model %>% compile(

optimizer = "rmsprop",

loss = "binary\_crossentropy",

metrics = c("accuracy")

)

You’re passing your optimizer, loss function, and metrics as strings, which is possible because rmsprop, binary\_crossentropy, and accuracy are packaged as part of Keras. Sometimes you may want to configure the parameters of your optimizer or pass a custom loss function or metric function. The former can be done by passing an optimizer instance as the optimizer argument:

model %>% compile(

optimizer = optimizer\_rmsprop(lr=0.001),

loss = "binary\_crossentropy",

metrics = c("accuracy")

)

The latter can be done by passing function objects as the loss or metrics arguments:

model %>% compile(

optimizer = optimizer\_rmsprop(lr = 0.001),

loss = loss\_binary\_crossentropy,

metrics = metric\_binary\_accuracy

)

## Validating our approach

In order to monitor during training the accuracy of the model on data that it has never seen before, we will create a “validation set” by setting apart 10,000 samples from the original training data:

val\_indices <- 1:10000

x\_val <- x\_train[val\_indices,]

partial\_x\_train <- x\_train[-val\_indices,]

y\_val <- y\_train[val\_indices]

partial\_y\_train <- y\_train[-val\_indices]

We will now train our model for 20 epochs (20 iterations over all samples in the x\_train and y\_train tensors), in mini-batches of 512 samples. At this same time we will monitor loss and accuracy on the 10,000 samples that we set apart. This is done by passing the validation data as the validation\_data argument:

model %>% compile(

optimizer = "rmsprop",

loss = "binary\_crossentropy",

metrics = c("accuracy")

)

history <- model %>% fit(

partial\_x\_train,

partial\_y\_train,

epochs = 20,

batch\_size = 512,

validation\_data = list(x\_val, y\_val)

)

On CPU, this will take less than two seconds per epoch – training is over in 20 seconds. At the end of every epoch, there is a slight pause as the model computes its loss and accuracy on the 10,000 samples of the validation data.

Note that the call to fit() returns a history object. Let’s take a look at it:

str(history)

## List of 2

## $ params :List of 8

## ..$ batch\_size : int 512

## ..$ epochs : int 20

## ..$ steps : NULL

## ..$ samples : int 15000

## ..$ verbose : int 1

## ..$ do\_validation : logi TRUE

## ..$ metrics : chr [1:4] "loss" "acc" "val\_loss" "val\_acc"

## ..$ validation\_samples: int 10000

## $ metrics:List of 4

## ..$ val\_loss: num [1:20] 0.407 0.309 0.279 0.272 0.275 ...

## ..$ val\_acc : num [1:20] 0.856 0.889 0.893 0.89 0.89 ...

## ..$ loss : num [1:20] 0.527 0.321 0.232 0.183 0.147 ...

## ..$ acc : num [1:20] 0.79 0.897 0.925 0.94 0.953 ...

## - attr(\*, "class")= chr "keras\_training\_history"

The history object includes various parameters used to fit the model (history$params) as well as data for each of the metrics being monitored (history$metrics).

The history object has a plot() method that enables us to visualize the training and validation metrics by epoch:

plot(history)

The accuracy is plotted on the top panel and the loss on the bottom panel. Note that your own results may vary slightly due to a different random initialization of your network.

The dots are the training loss and accuracy, while the solid lines are the validation loss and accuracy. Note that your own results may vary slightly due to a different random initialization of your network.

As you can see, the training loss decreases with every epoch, and the training accuracy increases with every epoch. That’s what you would expect when running a gradient-descent optimization – the quantity you’re trying to minimize should be less with every iteration. But that isn’t the case for the validation loss and accuracy: they seem to peak at the fourth epoch. This is an example of what we warned against earlier: a model that performs better on the training data isn’t necessarily a model that will do better on data it has never seen before. In precise terms, what you’re seeing is overfitting: after the second epoch, you’re over-optimizing on the training data, and you end up learning representations that are specific to the training data and don’t generalize to data outside of the training set.

In this case, to prevent overfitting, you could stop training after three epochs. In general, you can use a range of techniques to mitigate overfitting, which we’ll cover in chapter 4.

Let’s train a new network from scratch for four epochs and then evaluate it on the test data.

model <- keras\_model\_sequential() %>%

layer\_dense(units = 16, activation = "relu", input\_shape = c(10000)) %>%

layer\_dense(units = 16, activation = "relu") %>%

layer\_dense(units = 1, activation = "sigmoid")

model %>% compile(

optimizer = "rmsprop",

loss = "binary\_crossentropy",

metrics = c("accuracy")

)

model %>% fit(x\_train, y\_train, epochs = 4, batch\_size = 512)

results <- model %>% evaluate(x\_test, y\_test)

results

## $loss

## [1] 0.3187578

##

## $acc

## [1] 0.8742

Our fairly naive approach achieves an accuracy of 88%. With state-of-the-art approaches, one should be able to get close to 95%.

## Using a trained network to generate predictions on new data

After having trained a network, you’ll want to use it in a practical setting. You can generate the likelihood of reviews being positive by using the predict method:

model %>% predict(x\_test[1:10,])

## [,1]

## [1,] 0.23027171

## [2,] 0.99983275

## [3,] 0.95128500

## [4,] 0.92735493

## [5,] 0.97868216

## [6,] 0.85068083

## [7,] 0.99992847

## [8,] 0.01188476

## [9,] 0.98150921

## [10,] 0.99729294

As you can see, the network is very confident for some samples (0.99 or more, or 0.02 or less) but less confident for others.

### Classifying newswires: a multiclass classification example

In this example, we will build a model to classify Reuters newswires into 46 mutually exclusive topics. Because we have many classes, this problem is an instance of multiclass classification; and because each data point should be classified into only one category, the problem is more specifically an instance of single-label, multiclass classification. If each data point could belong to multiple categories (in this case, topics), you’d be facing a multilabel, multiclass classification problem.

### Reuters dataset

We will work with the Reuters dataset, a set of short newswires and their topics, published by Reuters in 1986. It’s a simple, widely used toy dataset for text classification. There are 46 different topics; some topics are more represented than others, but each topic has at least 10 examples in the training set. Like IMDB and MNIST, the Reuters dataset comes packaged as part of Keras. Let’s begin......

In [1]:

import numpy as np

import pandas as pd

import warnings

import tensorflow as tf # import tensor flow

import numpy as np

import keras

from tensorflow.keras import layers

import matplotlib.pyplot as plt

#### Load Dataset

In [2]:

from tensorflow.keras.datasets import reuters

(train\_data, train\_labels), (test\_data, test\_labels) = reuters.load\_data(num\_words=10000)

Downloading data from https://storage.googleapis.com/tensorflow/tf-keras-datasets/reuters.npz

2113536/2110848 [==============================] - 0s 0us/step

/opt/conda/lib/python3.7/site-packages/tensorflow/python/keras/datasets/reuters.py:148: VisibleDeprecationWarning: Creating an ndarray from ragged nested sequences (which is a list-or-tuple of lists-or-tuples-or ndarrays with different lengths or shapes) is deprecated. If you meant to do this, you must specify 'dtype=object' when creating the ndarray

x\_train, y\_train = np.array(xs[:idx]), np.array(labels[:idx])

/opt/conda/lib/python3.7/site-packages/tensorflow/python/keras/datasets/reuters.py:149: VisibleDeprecationWarning: Creating an ndarray from ragged nested sequences (which is a list-or-tuple of lists-or-tuples-or ndarrays with different lengths or shapes) is deprecated. If you meant to do this, you must specify 'dtype=object' when creating the ndarray

x\_test, y\_test = np.array(xs[idx:]), np.array(labels[idx:])

In [3]:

len(train\_data),len(test\_data)

Out[3]:

(8982, 2246)

The argument num\_words=10000 restricts the data to the 10,000 most frequently occurring words found in the data. You have 8,982 training examples and 2,246 test examples:

In [4]:

print(train\_data[10])

[1, 245, 273, 207, 156, 53, 74, 160, 26, 14, 46, 296, 26, 39, 74, 2979, 3554, 14, 46, 4689, 4329, 86, 61, 3499, 4795, 14, 61, 451, 4329, 17, 12]

Each example is a list of integers (word indices):

#### Here’s how you can decode it back to words, in case you’re curious.

In [5]:

word\_index = reuters.get\_word\_index()

reverse\_word\_index = dict([(value, key) for (key, value) **in** word\_index.items()])

decoded\_newswire = ' '.join([reverse\_word\_index.get(i - 3, '?') for i **in**

train\_data[0]])

Downloading data from https://storage.googleapis.com/tensorflow/tf-keras-datasets/reuters\_word\_index.json

557056/550378 [==============================] - 0s 0us/step

Note that the indices are offset by 3 because 0, 1, and 2 are reserved indices for “padding,” “start of sequence,” and “unknown.” The label associated with an example is an integer between 0 and 45 — a topic index:

In [6]:

train\_labels[10]

Out[6]:

3

### Data Prep

#### vectorize the input data

In [7]:

def vectorize\_sequences(sequences, dimension=10000):

results = np.zeros((len(sequences), dimension))

for i, sequence **in** enumerate(sequences):

results[i, sequence] = 1.

return results

x\_train = vectorize\_sequences(train\_data)#1

x\_test = vectorize\_sequences(test\_data)#2

1. Verctorize training data
2. Vencotrize testing data

#### vectorize the label with the exact same code as in the previous example.

In [8]:

def to\_one\_hot(labels, dimension=46):

results = np.zeros((len(labels), dimension))

for i, label **in** enumerate(labels):

results[i, label] = 1.

return results

one\_hot\_train\_labels = to\_one\_hot(train\_labels)#1

one\_hot\_test\_labels = to\_one\_hot(test\_labels)#2

1. Verctorize training labels
2. Vencotrize testing labels

Note that there is a built-in way to do this in Keras:

In [9]:

from tensorflow.keras.utils import to\_categorical

one\_hot\_train\_labels = to\_categorical(train\_labels)

one\_hot\_test\_labels = to\_categorical(test\_labels)

### Building the model

This topic-classification problem looks similar to the previous movie-review classificationq: in both cases, we are trying to classify short snippets of text. But there is a new constraint here: the number of output classes has gone from 2 to 46. The dimensionality of the output space is much larger.

In a stack of Dense layers like that we have been using, each layer can only access information present in the output of the previous layer. If one layer drops some information relevant to the classification problem, this information can never be recovered by later layers: each layer can potentially become an information bottleneck. In the previous example, we used 16-dimensional intermediate layers, but a 16-dimensional space may be too limited to learn to separate 46 different classes: such small layers may act as information bottlenecks, permanently dropping relevant information. For this reason we will use larger layers. Let’s go with 64 units.

#### Model Defination

In [10]:

model = keras.Sequential([

layers.Dense(64, activation='relu'),

layers.Dense(64, activation='relu'),

layers.Dense(46, activation='softmax')

])

Note about this architecture:

1. We end the model with a Dense layer of size 46. This means for each input sample, the network will output a 46-dimensional vector. Each entry in this vector (each dimension) will encode a different output class.
2. The last layer uses a softmax activation. You saw this pattern in the MNIST example. It means the model will output a probability distribution over the 46 different output classes — for every input sample, the model will produce a 46-dimensional output vector, where output[i] is the probability that the sample belongs to class i. The 46 scores will sum to 1.
3. The best loss function to use in this case is categorical\_crossentropy. It measures the distance between two probability distributions: here, between the probability distribution output by the model and the true distribution of the labels. By minimizing the distance between these two distributions, you train the model to output something as close as possible to the true labels.

#### Compile the model

In [11]:

model.compile(optimizer='rmsprop',

loss='categorical\_crossentropy',

metrics=['accuracy'])

#### Validation of the approach

Let’s set apart 1,000 samples in the training data to use as a validation set.

In [12]:

x\_val = x\_train[:1000]

partial\_x\_train = x\_train[1000:]

y\_val = one\_hot\_train\_labels[:1000]

partial\_y\_train = one\_hot\_train\_labels[1000:]

#### let’s train the model for 20 epochs.

#### Training the model

In [13]:

history = model.fit(partial\_x\_train,

partial\_y\_train,

epochs=20,

batch\_size=512,

validation\_data=(x\_val, y\_val))

Epoch 1/20

16/16 [==============================] - 2s 81ms/step - loss: 3.1029 - accuracy: 0.4079 - val\_loss: 1.7132 - val\_accuracy: 0.6440

Epoch 2/20

16/16 [==============================] - 1s 38ms/step - loss: 1.4807 - accuracy: 0.6992 - val\_loss: 1.2964 - val\_accuracy: 0.7230

Epoch 3/20

16/16 [==============================] - 1s 36ms/step - loss: 1.0763 - accuracy: 0.7762 - val\_loss: 1.1460 - val\_accuracy: 0.7380

Epoch 4/20

16/16 [==============================] - 1s 36ms/step - loss: 0.8441 - accuracy: 0.8245 - val\_loss: 1.0389 - val\_accuracy: 0.7810

Epoch 5/20

16/16 [==============================] - 1s 37ms/step - loss: 0.6595 - accuracy: 0.8658 - val\_loss: 0.9456 - val\_accuracy: 0.8050

Epoch 6/20

16/16 [==============================] - 1s 37ms/step - loss: 0.5237 - accuracy: 0.8945 - val\_loss: 0.9203 - val\_accuracy: 0.8040

Epoch 7/20

16/16 [==============================] - 1s 36ms/step - loss: 0.4181 - accuracy: 0.9160 - val\_loss: 0.8765 - val\_accuracy: 0.8140

Epoch 8/20

16/16 [==============================] - 1s 35ms/step - loss: 0.3485 - accuracy: 0.9316 - val\_loss: 0.8895 - val\_accuracy: 0.8060

Epoch 9/20

16/16 [==============================] - 1s 36ms/step - loss: 0.2829 - accuracy: 0.9390 - val\_loss: 0.8829 - val\_accuracy: 0.8110

Epoch 10/20

16/16 [==============================] - 1s 36ms/step - loss: 0.2246 - accuracy: 0.9479 - val\_loss: 0.9112 - val\_accuracy: 0.8140

Epoch 11/20

16/16 [==============================] - 1s 36ms/step - loss: 0.1894 - accuracy: 0.9532 - val\_loss: 0.9060 - val\_accuracy: 0.8120

Epoch 12/20

16/16 [==============================] - 1s 37ms/step - loss: 0.1765 - accuracy: 0.9538 - val\_loss: 0.9068 - val\_accuracy: 0.8160

Epoch 13/20

16/16 [==============================] - 1s 37ms/step - loss: 0.1610 - accuracy: 0.9529 - val\_loss: 0.9394 - val\_accuracy: 0.8100

Epoch 14/20

16/16 [==============================] - 1s 37ms/step - loss: 0.1438 - accuracy: 0.9574 - val\_loss: 0.9254 - val\_accuracy: 0.8190

Epoch 15/20

16/16 [==============================] - 1s 35ms/step - loss: 0.1305 - accuracy: 0.9584 - val\_loss: 0.9666 - val\_accuracy: 0.8060

Epoch 16/20

16/16 [==============================] - 1s 37ms/step - loss: 0.1291 - accuracy: 0.9562 - val\_loss: 0.9537 - val\_accuracy: 0.8120

Epoch 17/20

16/16 [==============================] - 1s 36ms/step - loss: 0.1140 - accuracy: 0.9593 - val\_loss: 1.0202 - val\_accuracy: 0.8020

Epoch 18/20

16/16 [==============================] - 1s 38ms/step - loss: 0.1167 - accuracy: 0.9567 - val\_loss: 0.9942 - val\_accuracy: 0.8070

Epoch 19/20

16/16 [==============================] - 1s 38ms/step - loss: 0.0972 - accuracy: 0.9669 - val\_loss: 1.0709 - val\_accuracy: 0.7960

Epoch 20/20

16/16 [==============================] - 1s 34ms/step - loss: 0.1035 - accuracy: 0.9607 - val\_loss: 1.0530 - val\_accuracy: 0.8020

#### Plotting the training and validation loss

In [14]:

loss = history.history['loss']

val\_loss = history.history['val\_loss']

epochs = range(1, len(loss) + 1)

plt.plot(epochs, loss, 'bo', label='Training loss')

plt.plot(epochs, val\_loss, 'r', label='Validation loss')

plt.title('Training and validation loss')

plt.xlabel('Epochs')

plt.ylabel('Loss')

plt.legend()

plt.show()

#### Plotting the training and validation accuracy

In [15]:

plt.clf()

acc = history.history['accuracy']

val\_acc = history.history['val\_accuracy']

plt.plot(epochs, acc, 'bo', label='Training accuracy')

plt.plot(epochs, val\_acc, 'r', label='Validation accuracy')

plt.title('Training and validation accuracy')

plt.xlabel('Epochs')

plt.ylabel('Accuracy')

plt.legend()

plt.show()

The model begins to overfit after nine epochs. Let’s train a new model from scratch for nine epochs and then evaluate it on the test set.

#### Retraining a model from scratch

In [16]:

model = keras.Sequential([

layers.Dense(64, activation='relu'),

layers.Dense(64, activation='relu'),

layers.Dense(46, activation='softmax')

])

model.compile(optimizer='rmsprop',

loss='categorical\_crossentropy',

metrics=['accuracy'])

model.fit(partial\_x\_train,

partial\_y\_train,

epochs=10,

batch\_size=512,

validation\_data=(x\_val, y\_val))

results = model.evaluate(x\_test, one\_hot\_test\_labels)

Epoch 1/10

16/16 [==============================] - 1s 49ms/step - loss: 3.0635 - accuracy: 0.4262 - val\_loss: 1.6590 - val\_accuracy: 0.6280

Epoch 2/10

16/16 [==============================] - 1s 36ms/step - loss: 1.4873 - accuracy: 0.6789 - val\_loss: 1.3835 - val\_accuracy: 0.6790

Epoch 3/10

16/16 [==============================] - 1s 35ms/step - loss: 1.1232 - accuracy: 0.7512 - val\_loss: 1.1722 - val\_accuracy: 0.7370

Epoch 4/10

16/16 [==============================] - 1s 34ms/step - loss: 0.8697 - accuracy: 0.8107 - val\_loss: 1.0571 - val\_accuracy: 0.7690

Epoch 5/10

16/16 [==============================] - 1s 36ms/step - loss: 0.6774 - accuracy: 0.8567 - val\_loss: 1.0274 - val\_accuracy: 0.7750

Epoch 6/10

16/16 [==============================] - 1s 38ms/step - loss: 0.5542 - accuracy: 0.8814 - val\_loss: 0.9605 - val\_accuracy: 0.7870

Epoch 7/10

16/16 [==============================] - 1s 36ms/step - loss: 0.4583 - accuracy: 0.9006 - val\_loss: 0.9107 - val\_accuracy: 0.7990

Epoch 8/10

16/16 [==============================] - 1s 37ms/step - loss: 0.3609 - accuracy: 0.9235 - val\_loss: 0.9101 - val\_accuracy: 0.8120

Epoch 9/10

16/16 [==============================] - 1s 36ms/step - loss: 0.3064 - accuracy: 0.9338 - val\_loss: 0.9153 - val\_accuracy: 0.8100

Epoch 10/10

16/16 [==============================] - 1s 35ms/step - loss: 0.2477 - accuracy: 0.9433 - val\_loss: 0.9159 - val\_accuracy: 0.8100

71/71 [==============================] - 0s 2ms/step - loss: 0.9991 - accuracy: 0.7858

In [17]:

results

Out[17]:

[0.9991406202316284, 0.7858415246009827]

#### This approach reaches an accuracy of ~79%. With a balanced binary classification problem, the accuracy reached by a purely random classifier would be 50%. But in this case, we have 46 classes, and they may not be equally represented. What would be the accuracy of a random baseline? We could try quickly implementing one to check this empirically:

In [18]:

import copy

test\_labels\_copy = copy.copy(test\_labels)

np.random.shuffle(test\_labels\_copy)

hits\_array = np.array(test\_labels) == np.array(test\_labels\_copy)

float(np.sum(hits\_array))/len(test\_labels)

Out[18]:

0.176313446126447

#### As we can see, a random classifier would score around 19% classification accuracy, so the results of our model seem pretty good in that light.

### Generating prediction on the new data

Calling the predict method on new samples returns a class probability distribution over all 46 topics for each sample. Let’s generate topic predictions for all of the test data.

In [19]:

predictions = model.predict(x\_test)

#### Each entry in “predictions` is a vector of length 46:

In [20]:

predictions[0].shape

Out[20]:

(46,)

#### The coefficients in this vector sum to 1, as they form a probability distribution:

In [21]:

np.sum(predictions[0])

Out[21]:

1.0

#### The largest entry is the predicted class — the class with the highest probability:

In [22]:

np.argmax(predictions[0])

Out[22]:

3

### Key Takeaways

1. If we are trying to classify data points among N classes, model should end with a Dense layer of size N.
2. In a single-label, multiclass classification problem, our model should end with a softmax activation so that it will output a probability distribution over the N output classes.
3. Categorical crossentropy is almost always the loss function you should use for such problems. It minimizes the distance between the probability distributions output by the model and the true distribution of the targets.
4. There are two ways to handle labels in multiclass classification: Encoding the labels via categorical encoding (also known as one-hot encoding) and using  
   categorical\_crossentropy as a loss function Encoding the labels as integers and using the sparsecategorical-crossentropy loss function
5. If you need to classify data into a large number of categories, you should avoid creating information bottlenecks in your model due to interme

#### Further explorations

Try using larger or smaller layers: 32 units, 128 units, and so on. We used two intermediate layers before the final softmax classification layer. Now try using a single intermediate layer, or three intermediate layers.