EXPERT INSIGHT

PyTorch Deep Learning Hands-On

Build CNNs, RNNs, GANs, reinforcement learning, and more, quickly and easily

Sherin Thomas
with Sudhanshu Passi

Packt

PyTorch Deep Learning Hands-On

Build CNNs, RNNs, GANs, reinforcement learning, and more, quickly and easily

Sherin Thomas

with Sudhanshu Passi

Packt

BIRMINGHAM - MUMBAI

PyTorch Deep Learning Hands-On

Acquisition Editor: Andrew Waldron
Acquisition Editor - Peer Reviews: Suresh Jain
Project Editor: Tom Jacob
Development Editor: Joanne Lovell
Technical Editor: Gaurav Gavas
Proofreader: Safis Editing
Indexer: Rekha Nair
Graphics: Sandip Tadge, Tom Scaria
Production Coordinator: Sandip Tadge

First published: April 2019

Production reference: 2180719

Published by Packt Publishing Ltd.
Livery Place
35 Livery Street
Birmingham B3 2PB, UK.

ISBN 978-1-78883-413-1

`www.packtpub.com`

Contributors

About the authors

Sherin Thomas started his career as an information security expert and shifted his focus to deep learning-based security systems. He has helped several companies across the globe to set up their AI pipelines and worked recently for CoWrks, a fast-growing start-up based out of Bengaluru. Sherin is working on several open source projects including PyTorch, RedisAI, and many more, and is leading the development of TuringNetwork.ai. Currently, he is focusing on building the deep learning infrastructure for [tensor]werk, an Orobix spin-off company.

I am indebted to a multitude of professionals who have influenced me and motivated me to write this book. Among them are my colleagues from CoWrks and my friends. I can't thank enough the technical reviewers and editorial assistants. Without them, I would not have been able to meet the deadlines. Last, and most importantly, I am indebted to my wife, Merin. Writing a book along with a day job is not easy, and without her, it would have been impossible.

Sudhanshu Passi is a technologist employed at CoWrks. Among other things, he has been the driving force behind everything related to machine learning at CoWrks. His expertise in simplifying complex concepts makes his work an ideal read for beginners and experts alike. This can be verified by his many blogs and this debut book publication. In his spare time, he can be found at his local swimming pool computing gradient descent underwater.

I would like to thank Sherin for this opportunity to be a co-author on this book. I would also like to thank my parents for their continuous support throughout the years.

About the reviewers

Bharath G. S. is an independent machine learning researcher and he currently works with glib.ai as a machine learning engineer. He also collaborates with mcg.ai as a machine learning consultant. His main research areas of interest include reinforcement learning, natural language processing, and cognitive neuroscience. Currently, he's researching the issue of algorithmic fairness in decision making. He's also involved in the open source development of the privacy-preserving machine learning platform OpenMined as a core collaborator, where he works on private and secure decentralized deep learning algorithms. You can also find some of the machine learning libraries that he has co-authored on PyPI, such as parfit, NALU, and pysyft.

Liao Xingyu is pursuing his master's degree in **University of Science and Technology of China** (**USTC**). He has ever worked in Megvii.inc and JD AI lab as an intern. He has published a Chinese PyTorch book named *Learn Deep Learning with PyTorch* in China.

I am grateful for my family's support and project editor Tom's help in producing and reviewing this book.

Table of Contents

Preface

PyTorch Deep Learning Hands-On helps readers to get into the depths of deep learning quickly. In the last couple of years, we have seen deep learning become the new electricity. It has fought its way from academia into industry, helping resolve thousands of enigmas that humans could never have imagined solving without it. The mainstream adoption of deep learning as a go-to implementation was driven mainly by a bunch of frameworks that reliably delivered complex algorithms as efficient built-in methods. This book showcases the benefits of PyTorch for prototyping a deep learning model, for building a deep learning workflow, and for taking a prototyped model to production. Overall, the book concentrates on the practical implementation of PyTorch instead of explaining the math behind it, but it also links you to places that you could fall back to if you lag behind with a few concepts.

Who this book is for

We have refrained from explaining the algorithms as much as possible and have instead focused on their implementation in PyTorch, sometimes looking at the implementation of real-world applications using those algorithms. This book is ideal for those who know how to program in Python and understand the basics of deep learning. This book is for people who are practicing traditional machine learning concepts already or who are developers and want to explore the world of deep learning practically and deploy their implementations to production.

What this book covers

Chapter 1, *Deep Learning Walkthrough and PyTorch Introduction*, is an introduction to the PyTorch way of doing deep learning and to the basic APIs of PyTorch. It starts by showing the history of PyTorch and why PyTorch should be the go-to framework for deep learning development. It also covers an introduction of the different deep learning approaches that we will be covering in the upcoming chapters.

Chapter 2, *A Simple Neural Network*, helps you build your first simple neural network and shows how we can connect bits and pieces such as neural networks, optimizers, and parameter updates to build a novice deep learning model. It also covers how PyTorch does backpropagation, the key behind all state-of-the-art deep learning algorithms.

Chapter 3, *Deep Learning Workflow*, goes deeper into the deep learning workflow implementation and the PyTorch ecosystem that helps build the workflow. This is probably the most crucial chapter if you are planning to set up a deep learning team or a pipeline for an upcoming project. In this chapter, we'll go through the different stages of a deep learning pipeline and see how the PyTorch community has advanced in each stage in the workflow iteratively by making appropriate tools.

Chapter 4, *Computer Vision*, being the most successful result of deep learning so far, talks about the key ideas behind that success and runs through the most widely used vision algorithm – the **convolutional neural network** (**CNN**). We'll implement a CNN step by step to understand the working principles, and then use a predefined CNN from PyTorch's `nn` package. This chapter helps you make a simple CNN and an advanced CNN-based vision algorithm called semantic segmentation.

Chapter 5, *Sequential Data Processing*, looks at the recurrent neural network, which is currently the most successful sequential data processing algorithm. The chapter introduces you to the major RNN components, such as the **long short-term memory** (**LSTM**) network and **gated recurrent units** (**GRUs**). Then we'll go through algorithmic changes in RNN implementation, such as bidirectional RNNs, and increasing the number of layers, before we explore recursive neural networks. To understand recursive networks, we'll use the renowned example, from the Stanford NLP group, the stack-augmented parser-interpreter neural network (SPINN), and implement that in PyTorch.

Chapter 6, *Generative Networks,* talks about the history of generative networks in brief and then explains the different kinds of generative networks. Among those different categories, this chapter introduces us to autoregressive models and GANs. We'll work through the implementation details of PixelCNN and WaveNet as part of autoregressive models, and then look at GANs in detail.

Chapter 7, *Reinforcement Learning*, introduces the concept of reinforcement learning, which is not really a subcategory of deep learning. We'll first take a look at defining problem statements. Then we'll explore the concept of cumulative rewards. We'll explore Markov decision processes and the Bellman equation, and then move to deep Q-learning. We'll also see an introduction to Gym, the toolkit developed by OpenAI for developing and experimenting with reinforcement learning algorithms.

Chapter 8, *PyTorch to Production*, looks at the difficulties people face, even the deep learning experts, during the deployment of a deep learning model to production. We'll explore different options for production deployment, including using a Flask wrapper around PyTorch as well as using RedisAI, which is a highly optimized runtime for deploying models in multicluster environments and can handle millions of requests per second.

To get the most out of this book

- The code is written in Python and hosted on GitHub. Though the compressed code repository is available for download, the online GitHub repository will receive bug fixes and updates. Having a basic understanding of GitHub is therefore required, as is having good Python knowledge.
- Although not mandatory, the use of CUDA drivers would help to speed up the training process if you are not using any pretrained models.
- The code examples were developed on an Ubuntu 18.10 machine but will work on all the popular platforms. But if you find any difficulties, feel free to raise an issue in the GitHub repository.
- Some of the examples in the book require you to use other services or packages, such as redis-server and the Flask framework. All those external dependencies and "how-to" guides are documented in the chapters they appear.

Download the example code files

You can download the example code files for this book from your account at `http://www.packt.com`. If you purchased this book elsewhere, you can visit `http://www.packt.com/support` and register to have the files emailed directly to you.

You can download the code files by following these steps:

1. Log in or register at `http://www.packt.com`.
2. Select the **SUPPORT** tab.
3. Click on **Code Downloads & Errata**.
4. Enter the name of the book in the **Search** box and follow the on-screen instructions.

Once the file is downloaded, please make sure that you unzip or extract the folder using the latest version of:

- WinRAR / 7-Zip for Windows
- Zipeg / iZip / UnRarX for macOS
- 7-Zip / PeaZip for Linux

The code bundle for the book is also hosted on GitHub at `https://github.com/hhsecond/HandsOnDeepLearningWithPytorch`. We also have other code bundles from our rich catalog of books and videos available at `https://github.com/PacktPublishing/`. Check them out!

Download the color images

We also provide a PDF file that has color images of the screenshots/diagrams used in this book. You can download it here: `http://www.packtpub.com/sites/default/files/downloads/9781788834131_ColorImages.pdf`.

Conventions used

There are a number of text conventions used throughout this book.

`CodeInText`: Indicates code words in text, database table names, folder names, filenames, file extensions, pathnames, dummy URLs, user input, and Twitter handles. For example; "Mount the downloaded `WebStorm-10*.dmg` disk image file as another disk in your system."

A block of code is set as follows:

```
def forward(self, batch):
    hidden = self.hidden(batch)
    activated = torch.sigmoid(hidden)
    out = self.out(activated)
    return out
```

When we wish to draw your attention to a particular part of a code block, the relevant lines or items are set in bold:

```
def binary_encoder(input_size):
    def wrapper(num):
        ret = [int(i) for i in '{0:b}'.format(num)]
        return [0] * (input_size - len(ret)) + ret
    return wrapper
```

Any command-line input or output is written as follows:

```
python -m torch.utils.bottleneck /path/to/source/script.py [args]
```

Bold: Indicates a new term, an important word, or words that you see on the screen, for example, in menus or dialog boxes, also appear in the text like this. For example: "Select **System info** from the **Administration** panel."

Warnings or important notes appear like this.

Tips and tricks appear like this.

Get in touch

Feedback from our readers is always welcome.

General feedback: If you have questions about any aspect of this book, mention the book title in the subject of your message and email us at `customercare@packtpub.com`.

Errata: Although we have taken every care to ensure the accuracy of our content, mistakes do happen. If you have found a mistake in this book we would be grateful if you would report this to us. Please visit, `http://www.packt.com/submit-errata`, selecting your book, clicking on the Errata Submission Form link, and entering the details.

Piracy: If you come across any illegal copies of our works in any form on the Internet, we would be grateful if you would provide us with the location address or website name. Please contact us at `copyright@packt.com` with a link to the material.

If you are interested in becoming an author: If there is a topic that you have expertise in and you are interested in either writing or contributing to a book, please visit `http://authors.packtpub.com`.

Reviews

Please leave a review. Once you have read and used this book, why not leave a review on the site that you purchased it from? Potential readers can then see and use your unbiased opinion to make purchase decisions, we at Packt can understand what you think about our products, and our authors can see your feedback on their book. Thank you!

For more information about Packt, please visit `packt.com`.

1

Deep Learning Walkthrough and PyTorch Introduction

At this point in time, there are dozens of deep learning frameworks out there that are capable of solving any sort of deep learning problem on GPU, so why do we need one more? This book is the answer to that million-dollar question. PyTorch came to the deep learning family with the promise of being NumPy on GPU. Ever since its entry, the community has been trying hard to keep that promise. As the official documentation says, PyTorch is an optimized tensor library for deep learning using GPUs and CPUs. While all the prominent frameworks offer the same thing, PyTorch has certain advantages over almost all of them.

The chapters in this book provide a step-by-step guide for developers who want to benefit from the power of PyTorch to process and interpret data. You'll learn how to implement a simple neural network, before exploring the different stages of a deep learning workflow. We'll dive into basic convolutional networks and generative adversarial networks, followed by a hands-on tutorial on how to train a model with OpenAI's Gym library. By the final chapter, you'll be ready to productionize PyTorch models.

In this first chapter, we will go through the theory behind PyTorch and explain why PyTorch gained the upper hand over other frameworks for certain use cases. Before that, we will take a glimpse into the history of PyTorch and learn why PyTorch is a need rather than an option. We'll also cover the NumPy-PyTorch bridge and PyTorch internals in the last section, which will give us a head start for the upcoming code-intensive chapters.

Understanding PyTorch's history

As more and more people started migrating to the fascinating world of machine learning, different universities and organizations began building their own frameworks to support their daily research, and Torch was one of the early members of that family. Ronan Collobert, Koray Kavukcuoglu, and Clement Farabet released Torch in 2002 and, later, it was picked up by Facebook AI Research and many other people from several universities and research groups. Lots of start-ups and researchers accepted Torch, and companies started productizing their Torch models to serve millions of users. Twitter, Facebook, DeepMind, and more are part of that list. As per the official Torch7 paper [1] published by the core team, Torch was designed with three key features in mind:

1. It should ease the development of numerical algorithms.
2. It should be easily extended.
3. It should be fast.

Although Torch gives flexibility to the bone, and the Lua + C combo satisfied all the preceding requirements, the major drawback the community faced was the learning curve to the new language, Lua. Although Lua wasn't difficult to grasp and had been used in the industry for a while for highly efficient product development, it did not have widespread acceptance like several other popular languages.

The widespread acceptance of Python in the deep learning community made some researchers and developers rethink the decision made by core authors to choose Lua over Python. It wasn't just the language: the absence of an imperative-styled framework with easy debugging capability also triggered the ideation of PyTorch.

The frontend developers of deep learning find the idea of the symbolic graph difficult. Unfortunately, almost all the deep learning frameworks were built on this foundation. In fact, a few developer groups tried to change this approach with dynamic graphs. Autograd from the Harvard Intelligent Probabilistic Systems Group was the first popular framework that did so. Then the Torch community on Twitter took the idea and implemented torch-autograd.

Next, a research group from **Carnegie Mellon University** (**CMU**) came up with DyNet, and then Chainer came up with the capability of dynamic graphs and an interpretable development environment.

All these events were a great inspiration for starting the amazing framework PyTorch, and, in fact, PyTorch started as a fork of Chainer. It began as an internship project by Adam Paszke, who was working under Soumith Chintala, a core developer of Torch. PyTorch then got two more core developers on board and around 100 alpha testers from different companies and universities.

The whole team pulled the chain together in six months and released the beta to the public in January 2017. A big chunk of the research community accepted PyTorch, although the product developers did not initially. Several universities started running courses on PyTorch, including **New York University** (**NYU**), Oxford University, and some other European universities.

What is PyTorch?

As mentioned earlier, PyTorch is a tensor computation library that can be powered by GPUs. PyTorch is built with certain goals, which makes it different from all the other deep learning frameworks. During this book, you'll be revisiting these goals through different applications and by the end of the book, you should be able to get started with PyTorch for any sort of use case you have in mind, regardless of whether you are planning to prototype an idea or build a super-scalable model to production.

Being a **Python-first framework**, PyTorch took a big leap over other frameworks that implemented a Python wrapper on a monolithic C++ or C engine. In PyTorch, you can inherit PyTorch classes and customize as you desire. The imperative style of coding, which was built into the core of PyTorch, was possible only because of the Python-first approach. Even though some symbolic graph frameworks, like TensorFlow, MXNet, and CNTK, came up with an imperative approach, PyTorch has managed to stay on top because of community support and its flexibility.

The **tape-based autograd** system enables PyTorch to have **dynamic graph** capability. This is one of the major differences between PyTorch and other popular symbolic graph frameworks. Tape-based autograd powered the backpropagation algorithm of Chainer, autograd, and torch-autograd as well. With dynamic graph capability, your graph gets created as the Python interpreter reaches the corresponding line. This is called *define by run*, unlike TensorFlow's *define and run* approach.

Tape-based autograd uses reverse-mode automatic differentiation, where the graph saves each operation to the tape while you forward pass and then move backward through the tape for backpropagation. Dynamic graphs and a Python-first approach allow **easy debugging**, where you can use the usual Python debuggers like Pdb or editor-based debuggers.

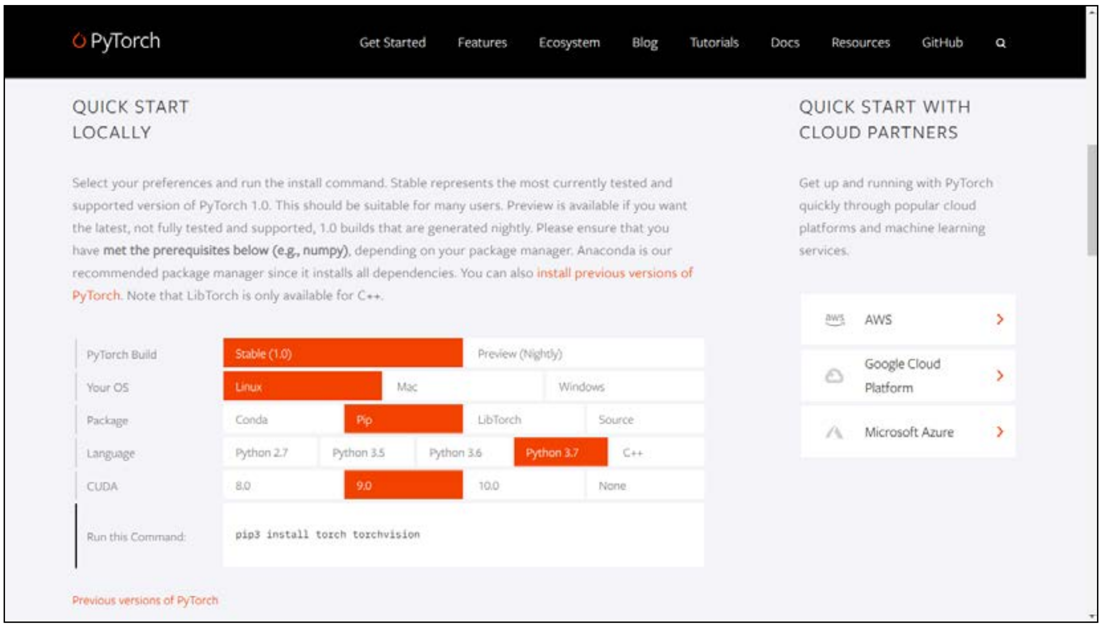
The PyTorch core community did not just make a Python wrapper over Torch's C binary: it optimized the core and made improvements to the core. PyTorch intelligently chooses which algorithm to run for each operation you define, based on the input data.

Installing PyTorch

If you have CUDA and cuDNN installed, PyTorch installation is dead simple (for GPU support, but in case you are trying out PyTorch and don't have GPUs with you, that's fine too). PyTorch's home page [2] shows an interactive screen to select the OS and package manager of your choice. Choose the options and execute the command to install it.

Though initially the support was just for Linux and Mac operating systems, from PyTorch 0.4 Windows is also in the supported operating system list. PyTorch has been packaged and shipped to PyPI and Conda. PyPI is the official Python repository for packages and the package manager, `pip`, can find PyTorch under the name Torch.

However, if you want to be adventurous and get the latest code, you can install PyTorch from the source by following the instructions on the GitHub `README` page. PyTorch has a nightly build that is being pushed to PyPI and Conda as well. A nightly build is useful if you want to get the latest code without going through the pain of installing from the source.



QUICK START LOCALLY

Select your preferences and run the install command. Stable represents the most currently tested and supported version of PyTorch 1.0. This should be suitable for many users. Preview is available if you want the latest, not fully tested and supported, 1.0 builds that are generated nightly. Please ensure that you have **met the prerequisites below (e.g., numpy)**, depending on your package manager. Anaconda is our recommended package manager since it installs all dependencies. You can also install previous versions of PyTorch. Note that LibTorch is only available for C++.

PyTorch Build	[✓ Stable (1.0)]	Preview (Nightly)			
Your OS	[✓ Linux]	Mac	Windows		
Package	Conda	[✓ Pip]	LibTorch	Source	
Language	Python 2.7	Python 3.5	Python 3.6	[✓ Python 3.7]	C++
CUDA	8.0	[✓ 9.0]	10.0	None	
Run this Command:	`pip3 install torch torchvision`				

Previous versions of PyTorch

QUICK START WITH CLOUD PARTNERS

Get up and running with PyTorch quickly through popular cloud platforms and machine learning services.

- AWS
- Google Cloud Platform
- Microsoft Azure

Figure 1.1: The installation process in the interactive UI from the PyTorch website

What makes PyTorch popular?

Among the multitude of reliable deep learning frameworks, static graphs or the symbolic graph-based approach were being used by almost everyone because of the speed and efficiency. The inherent problems with the dynamic network, such as performance issues, prevented developers from spending a lot of time implementing one. However, the restrictions of static graphs prevented researchers from thinking of a multitude of different ways to attack a problem because the thought process had to be confined inside the box of static computational graphs.

As mentioned earlier, Harvard's Autograd package started as a solution for this problem, and then the Torch community adopted this idea from Python and implemented torch-autograd. Chainer and CMU's DyNet are probably the next two dynamic-graph-based frameworks that got huge community support. Although all these frameworks could solve the problems that static graphs had created with the help of the imperative approach, they didn't have the momentum that other popular static graph frameworks had. PyTorch was the absolute answer for this. The PyTorch team took the backend of the well-tested, renowned Torch framework and merged that with the front of Chainer to get the best mix. The team optimized the core, added more Pythonic APIs, and set up the abstraction correctly, such that PyTorch doesn't need an abstract library like Keras for beginners to get started.

PyTorch achieved wide acceptance in the research community because a majority of people were using Torch already and probably were frustrated by the way frameworks like TensorFlow evolved without giving much flexibility. The dynamic nature of PyTorch was a bonus for lots of people and helped them to accept PyTorch in its early stages.

PyTorch lets users define whatever operations Python allows them to in the forward pass. The backward pass automatically finds the way through the graph until the root node, and calculates the gradient while traversing back. Although it was a revolutionary idea, the product development community had not accepted PyTorch, just like they couldn't accept other frameworks that followed similar implementation. However, as the days passed, more and more people started migrating to PyTorch. Kaggle witnessed competitions where all the top rankers used PyTorch, and as mentioned earlier, universities started doing courses in PyTorch. This helped students to avoid learning a new graph language like they had to when using a symbolic graph-based framework.

After the announcement of Caffe2, even product developers started experimenting with PyTorch, since the community announced the migration strategy of PyTorch models to Caffe2. Caffe2 is a static graph framework that can run your model even in mobile phones, so using PyTorch for prototyping is a win-win approach. You get the flexibility of PyTorch while building the network, and you get to transfer it to Caffe2 and use it in any production environment. However, with the 1.0 release note, the PyTorch team made a huge jump from letting people learn two frameworks (one for production and one for research), to learning a single framework that has dynamic graph capability in the prototyping phase and can suddenly convert to a static-like optimized graph when it requires speed and efficiency. The PyTorch team merged the backend of Caffe2 with PyTorch's Aten backend, which let the user decide whether they wanted to run a less-optimized but highly flexible graph, or an optimized but less-flexible graph without rewriting the code base.

ONNX and DLPack were the next two "big things" that the AI community saw. Microsoft and Facebook together announced the **Open Neural Network Exchange** (**ONNX**) protocol, which aims to help developers to migrate any model from any framework to any other. ONNX is compatible with PyTorch, Caffe2, TensorFlow, MXNet, and CNTK and the community is building/improving the support for almost all the popular frameworks.

ONNX is built into the core of PyTorch and hence migrating a model to ONNX form doesn't require users to install any other package or tool. Meanwhile, DLPack is taking interoperability to the next level by defining a standard data structure that different frameworks should follow, so that the migration of a tensor from one framework to another, in the same program, doesn't require the user to serialize data or follow any other workarounds. For instance, if you have a program that can use a well-trained TensorFlow model for computer vision and a highly efficient PyTorch model for recurrent data, you could use a single program that could handle each of the three-dimensional frames from a video with the TensorFlow model and pass the output of the TensorFlow model directly to the PyTorch model to predict actions from the video. If you take a step back and look at the deep learning community, you can see that the whole world converges toward a single point where everything is interoperable with everything else and trying to approach problems with similar methods. That's a world we all want to live in.

Using computational graphs

Through evolution, humans have found that graphing the neural network gives us the power of reducing complexity to the bare minimum. A computational graph describes the data flow in the network through operations.

A graph, which is made by a group of nodes and edges connecting them, is a decades-old data structure that is still heavily used in several different implementations and is a data structure that will be valid probably until humans cease to exist. In computational graphs, nodes represent the tensors and edges represent the relationship between them.

Computational graphs help us to solve the mathematics and make the big networks intuitive. Neural networks, no matter how complex or big they are, are a group of mathematical operations. The obvious approach to solving an equation is to divide the equation into smaller units and pass the output of one to another and so on. The idea behind the graph approach is the same. You consider the operations inside the network as nodes and map them to a graph with relations between nodes representing the transition from one operation to another.

Computational graphs are at the core of all current advances in artificial intelligence. They made the foundation of deep learning frameworks. All the deep learning frameworks existing now do computations using the graph approach. This helps the frameworks to find the independent nodes and do their computation as a separate thread or process. Computational graphs help with doing the backpropagation as easily as moving from the child node to previous nodes, and carrying the gradients along while traversing back. This operation is called automatic differentiation, which is a 40-year-old idea. Automatic differentiation is considered one of the 10 great numerical algorithms in the last century. Specifically, reverse-mode automatic differentiation is the core idea used behind computational graphs for doing backpropagation. PyTorch is built based on reverse-mode auto differentiation, so all the nodes keep the operation information with them until the control reaches the leaf node. Then the backpropagation starts from the leaf node and traverses backward. While moving back, the flow takes the gradient along with it and finds the partial derivatives corresponding to each node. In 1970, Seppo Linnainmaa, a Finnish mathematician and computer scientist, found that automatic differentiation can be used for algorithm verification. A lot of the other parallel efforts were recorded on the same concepts almost at the same time.

In deep learning, neural networks are for solving a mathematical equation. Regardless of how complex the task is, everything comes down to a giant mathematical equation, which you'll solve by optimizing the parameters of the neural network. The obvious way to solve it is "by hand." Consider solving the mathematical equation for ResNet with around 150 layers of a neural network; it is sort of impossible for a human being to iterate over such graphs thousands of times, doing the same operations manually each time to optimize the parameters. Computational graphs solve this problem by mapping all operations to a graph, level by level, and solving each node at a time. *Figure 1.2* shows a simple computational graph with three operators.

The matrix multiplication operator on both sides gives two matrices as output, and they go through an addition operator, which in turn goes through another sigmoid operator. The whole graph is, in fact, trying to solve this equation:

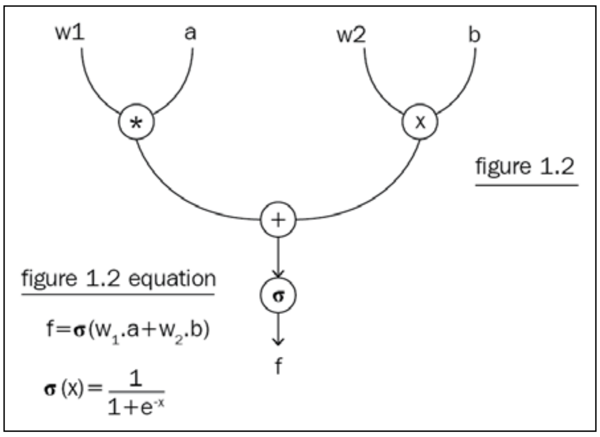


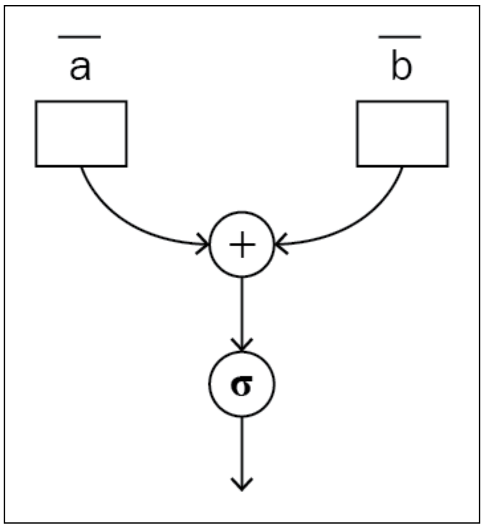
Figure 1.2: Graph representation of the equation

However, the moment you map it to a graph, everything becomes crystal clear. You can visualize and understand what is happening and easily code it up because the flow is right in front of you.

All deep learning frameworks are built on the foundation of automatic differentiation and computational graphs, but there are two inherently different approaches for the implementation–static and dynamic graphs.

Using static graphs

The traditional way of approaching neural network architecture is with static graphs. Before doing anything with the data you give, the program builds the forward and backward pass of the graph. Different development groups have tried different approaches. Some build the forward pass first and then use the same graph instance for the forward and backward pass. Another approach is to build the forward static graph first, and then create and append the backward graph to the end of the forward graph, so that the whole forward-backward pass can be executed as a single graph execution by taking the nodes in chronological order.



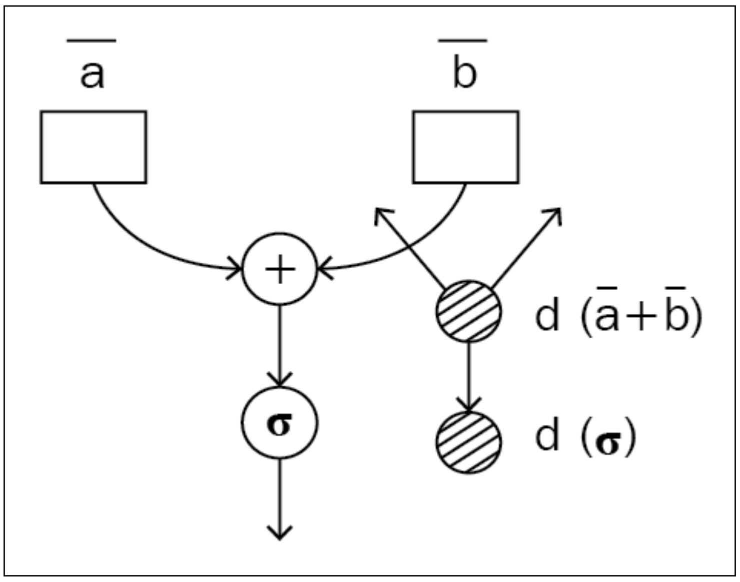


Figure 1.3 and 1.4: The same static graph used for the forward and backward pass

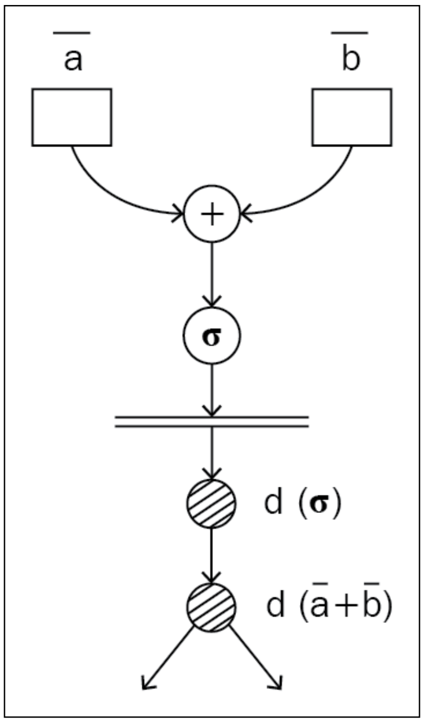


Figure 1.5: Static graph: a different graph for the forward and backward pass

Static graphs come with certain inherent advantages over other approaches. Since you are restricting the program from dynamic changes, your program can make assumptions related to memory optimization and parallel execution while executing the graph. Memory optimization is the key aspect that framework developers worry about through most of their development time, and the reason is the humungous scope of optimizing memory and the subtleties that come along with those optimizations. Apache MXNet developers have written an amazing blog [3] talking about this in detail.

The neural network for predicting the XOR output in TensorFlow's static graph API is given as follows. This is a typical example of how static graphs execute. Initially, we declare all the input placeholders and then build the graph. If you look carefully, nowhere in the graph definition are we passing the data into it. Input variables are actually placeholders expecting data sometime in the future. Though the graph definition looks like we are doing mathematical operations on the data, we are actually defining the process, and that's when TensorFlow builds the optimized graph implementation using the internal engine:

```
x = tf.placeholder(tf.float32, shape=[None, 2], name='x-input')
y = tf.placeholder(tf.float32, shape=[None, 2], name='y-input')
w1 = tf.Variable(tf.random_uniform([2, 5], -1, 1), name="w1")
```

```
w2 = tf.Variable(tf.random_uniform([5, 2], -1, 1), name="w2")
b1 = tf.Variable(tf.zeros([5]), name="b1")
b2 = tf.Variable(tf.zeros([2]), name="b2")
a2 = tf.sigmoid(tf.matmul(x, w1) + b1)
hyp = tf.matmul(a2, w2) + b2
cost = tf.reduce_mean(tf.losses.mean_squared_error(y, hyp))
train_step = tf.train.GradientDescentOptimizer(lr).minimize(cost)
prediction = tf.argmax(tf.nn.softmax(hyp), 1)
```

Once the interpreter finishes reading the graph definition, we start looping it through the data:

```
with tf.Session() as sess:
    sess.run(init)
    for i in range(epoch):
        sess.run(train_step, feed_dict={x_: XOR_X, y_: XOR_Y})
```

We start a TensorFlow session next. That's the only way you can interact with the graph you built beforehand. Inside the session, you loop through your data and pass the data to your graph using the `session.run` method. So, your input should be of the same size as you defined in the graph.

If you have forgotten what XOR is, the following table should give you enough information to recollect it from memory:

INPUT	**OUTPUT**	
A	**B**	**A XOR B**
0	0	0
0	1	1
1	0	1
1	1	0

Using dynamic graphs

The imperative style of programming has always had a larger user base, as the program flow is intuitive to any developer. Dynamic capability is a good side effect of imperative-style graph building. Unlike static graphs, dynamic graph architecture doesn't build the graph before the data pass. The program will wait for the data to come and build the graph as it iterates through the data. As a result, each iteration through the data builds a new graph instance and destroys it once the backward pass is done. Since the graph is being built for each iteration, it doesn't depend on the data size or length or structure. Natural language processing is one of the fields that needs this kind of approach.

For example, if you are trying to do sentiment analysis on thousands of sentences, with a static graph you need to hack and make workarounds. In a vanilla **recurrent neural network** (**RNN**) model, each word goes through one RNN unit, which generates output and the hidden state. This hidden state will be given to the next RNN, which processes the next word in the sentence. Since you made a fixed length slot while building your static graph, you need to augment your short sentences and cut down long sentences.

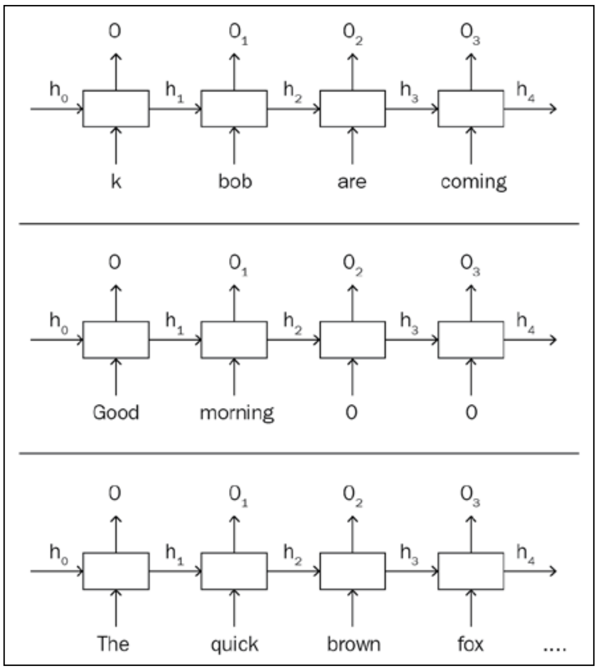


Figure 1.6: Static graph for an RNN unit with short, proper, and long sentences

The static graph given in the example shows how the data needs to be formatted for each iteration such that it won't break the prebuilt graph. However, in the dynamic graph, the network is flexible such that it gets created each time you pass the data, as shown in the preceding diagram.

The dynamic capability comes with a cost. Your graph cannot be preoptimized based on assumptions and you have to pay for the overhead of graph creation at each iteration. However, PyTorch is built to reduce the cost as much as possible. Since preoptimization is not something that a dynamic graph is capable of doing, PyTorch developers managed to bring down the cost of instant graph creation to a negligible amount. With all the optimization going into the core of PyTorch, it has proved to be faster than several other frameworks for specific use cases, even while offering the dynamic capability.

Following is a code snippet written in PyTorch for the same XOR operation we developed earlier in TensorFlow:

```
x = torch.FloatTensor(XOR_X)
y = torch.FloatTensor(XOR_Y)
w1 = torch.randn(2, 5, requires_grad=True)
w2 = torch.randn(5, 2, requires_grad=True)
b1 = torch.zeros(5, requires_grad=True)
b2 = torch.zeros(2, requires_grad=True)

for epoch in range(epochs):
    a1 = x @ w1 + b1
    h1 = a2.sigmoid()
    a2 = h2 @ w2 + b1
    hyp = a3.sigmoid()
    cost = (hyp - y).pow(2).sum()
    cost.backward()
```

In the PyTorch code, the input variable definition is not creating placeholders; instead, it is wrapping the variable object onto your input. The graph definition is not executing once; instead, it is inside your loop and the graph is being built for each iteration. The only information you share between each graph instance is your weight matrix, which is what you want to optimize.

In this approach, if your data size or shape is changing while you're looping through it, it's absolutely fine to run that new-shaped data through your graph because the newly created graph can accept the new shape. The possibilities do not end there. If you want to change the graph's behavior dynamically, you can do that too. The example given in the recursive neural network session in *Chapter 5*, *Sequential Data Processing,* is built on this idea.

Exploring deep learning

Since man invented computers, we have called them intelligent systems, and yet we are always trying to augment their intelligence. In the old days, anything a computer could do that a human couldn't was considered artificial intelligence. Remembering huge amounts of data, doing mathematical operations on millions or billions of numbers, and so on was considered artificial intelligence. We called Deep Blue, the machine that beat chess grandmaster Garry Kasparov at chess, an artificially intelligent machine.

Eventually, things that humans can't do and a computer can do became just computer programs. We realized that some things humans can do easily are impossible for a programmer to code up. This evolution changed everything. The number of possibilities or rules we could write down and make a computer work like us with was insanely large. Machine learning came to the rescue. People found a way to let the computers to learn the rules from examples, instead of having to code it up explicitly; that's called machine learning. An example is given in *Figure 1.9,* which shows how we could make a prediction of whether a customer will buy a product or not from his/her past shopping history.

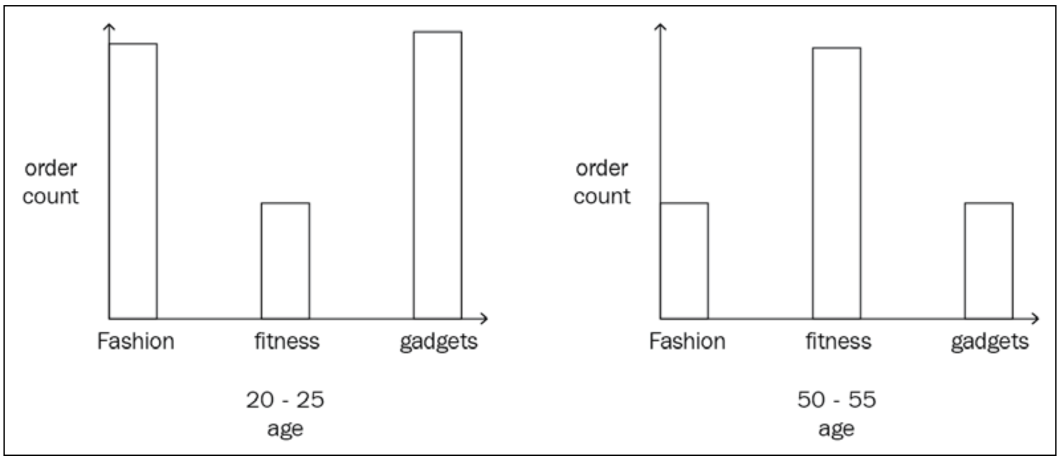


Figure 1.7: Showing the dataset for a customer buying a product

We could predict most of the results, if not all of them. However, what if the number of data points that we could make a prediction from is a lot and we cannot process them with a mortal brain? A computer could look through the data and probably spit out the answer based on previous data. This data-driven approach can help us a lot, since the only thing we have to do is assume the relevant features and give them to the black box, which consists of different algorithms, to learn the rules or pattern from the feature set.

There are problems. Even though we know what to look for, cleaning up the data and extracting the features is not an interesting task. The foremost trouble isn't this, however; we can't predict the features for high-dimensional data and the data of other media types efficiently. For example, in face recognition, we initially found the length of particulars in our face using the rule-based program and gave that to the neural network as input, because we thought that's the feature set that humans use to recognize faces.

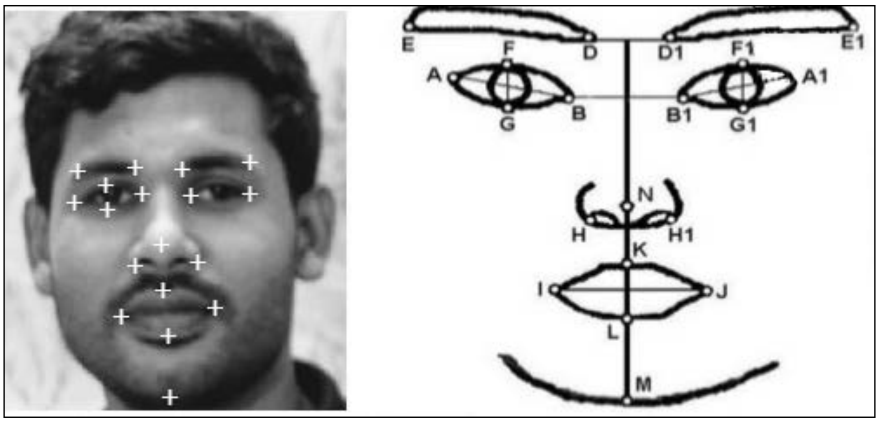


Figure 1.8: Human-selected facial features

It turned out that the features that are so obvious for humans are not so obvious for computers and vice versa. The realization of the feature selection problem led us to the era of deep learning. This is a subset of machine learning where we use the same data-driven approach, but instead of selecting the features explicitly, we let the computer decide what the features should be.

Let's consider our face recognition example again. FaceNet, a 2014 paper from Google, tackled it with the help of deep learning. FaceNet implemented the whole application using two deep networks. The first network was to identify the feature set from faces and the second network was to use this feature set and recognize the face (technically speaking, classifying the face into different buckets). Essentially, the first network was doing what we did before and the second network was a simple and traditional machine learning algorithm.

Deep networks are capable of identifying features from datasets, provided we have large labeled datasets. FaceNet's first network was trained with a huge dataset of faces with corresponding labels. The first network was trained to predict 128 features (generally speaking, there are 128 measurements from our faces, like the distance between the left eye and the right eye) from every face and the second network just used these 128 features to recognize a person.

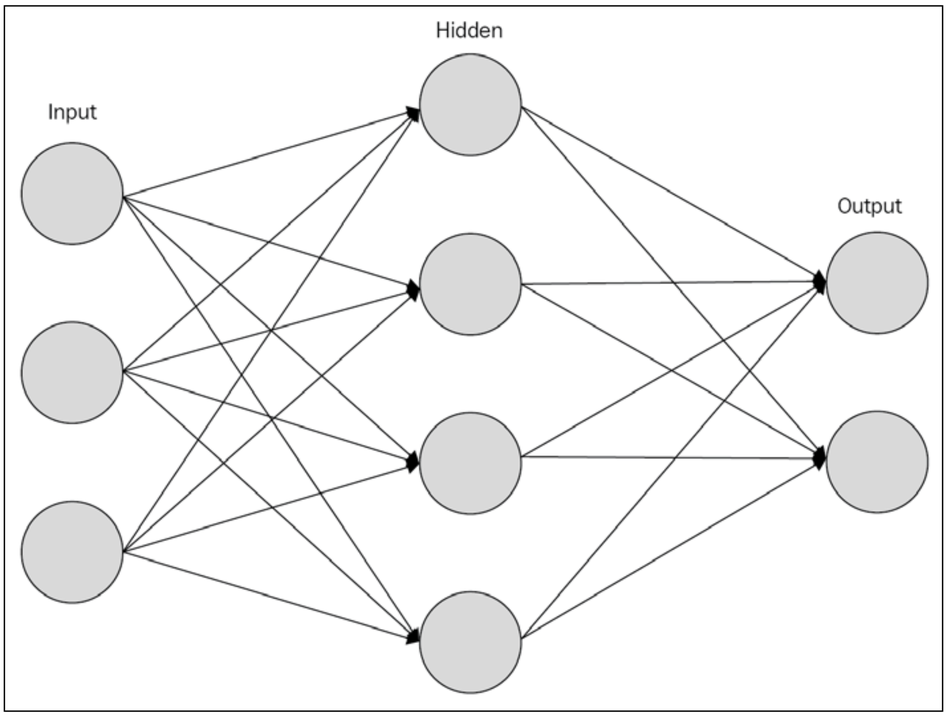


Figure 1.9: A simple neural network

A simple neural network has a single hidden layer, an input layer, and an output layer. Theoretically, a single hidden layer should be able to approximate any complex mathematical equation, and we should be fine with a single layer. However, it turns out that the single hidden layer theory is not so practical. In deep networks, each layer is responsible for finding some features. Initial layers find more detailed features, and final layers abstract these detailed features and find high-level features.

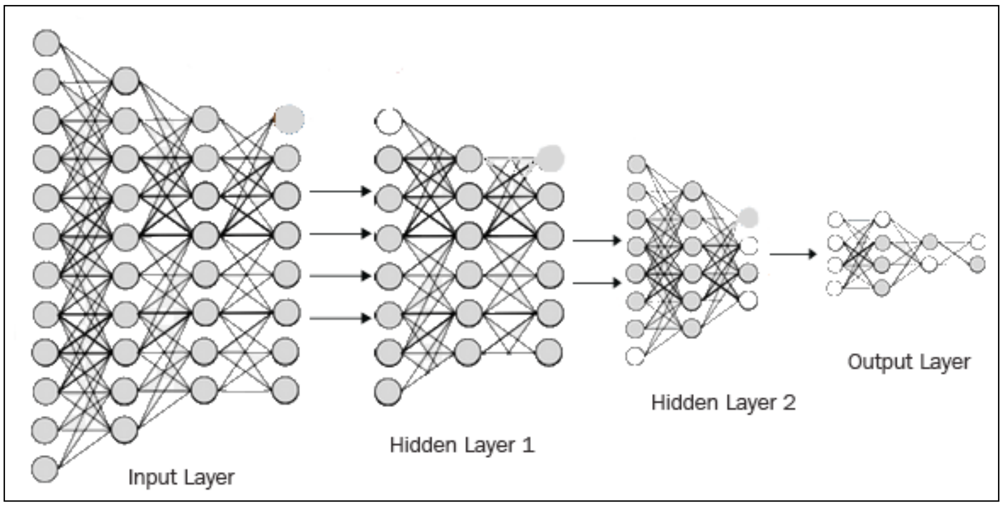


Figure 1.10: A deep neural network

Getting to know different architectures

Deep learning has been around for decades, and different structures and architectures evolved for different use cases. Some of them were based on ideas we had about our brain and some others were based on the actual working of the brain. All the upcoming chapters are based on the state-of-the-art architectures that the industry is using now. We'll cover one or more applications under each architecture, with each chapter covering the concepts, specifications, and technical details behind all of them, obviously with PyTorch code.

Fully connected networks

Fully connected, or dense or linear, networks are the most basic, yet powerful, architecture. This is a direct extension of what is commonly called machine learning, where you use neural networks with a single hidden layer. Fully connected layers act as the endpoint of all the architectures to find the probability distribution of the scores we find using the below deep network. A fully connected network, as the name suggests, has all the neurons connected to each other in the previous and next layers. The network might eventually decide to switch off some neurons by setting the weight, but in an ideal situation, initially, all of them take part in the communication.

Encoders and decoders

Encoders and decoders are probably the next most basic architecture under the deep learning umbrella. All the networks have one or more encoder-decoder layers. You can consider hidden layers in fully connected layers as the encoded form coming from an encoder, and the output layer as a decoder that decodes the hidden layer into output. Commonly, encoders encode the input into an intermediate state, where the input is represented as vectors and then the decoder network decodes this into an output form that we want.

A canonical example of an encoder-decoder network is the **sequence-to-sequence** (**seq2seq**) network, which can be used for machine translation. A sentence, say in English, will be encoded to an intermediate vector representation, where the whole sentence will be chunked in the form of some floating-point numbers and the decoder decodes the output sentence in another language from the intermediate vector.

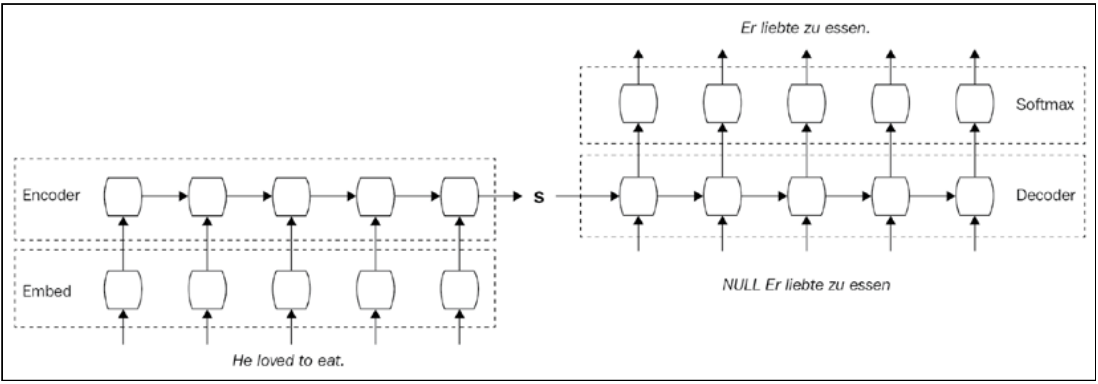


Figure 1.11: Seq2seq network

An autoencoder is a special type of encoder-decoder network and comes under the category of unsupervised learning. Autoencoders try to learn from unlabeled data, setting the target values to be equal to the input values. For example, if your input is an image of size 100 x 100, you'll have an input vector of dimension 10,000. So, the output size will also be 10,000, but the hidden layer size could be 500. In a nutshell, you are trying to convert your input to a hidden state representation of a smaller size, re-generating the same input from the hidden state.

If you were able to train a neural network that could do that, then voilà, you would have found a good compression algorithm where you could transfer high-dimensional input to a lower-dimensional vector with an order of magnitude's gain.

Autoencoders are being used in different situations and industries nowadays. You'll see a similar architecture in *Chapter 4*, *Computer Vision*, when we discuss semantic segmentation.

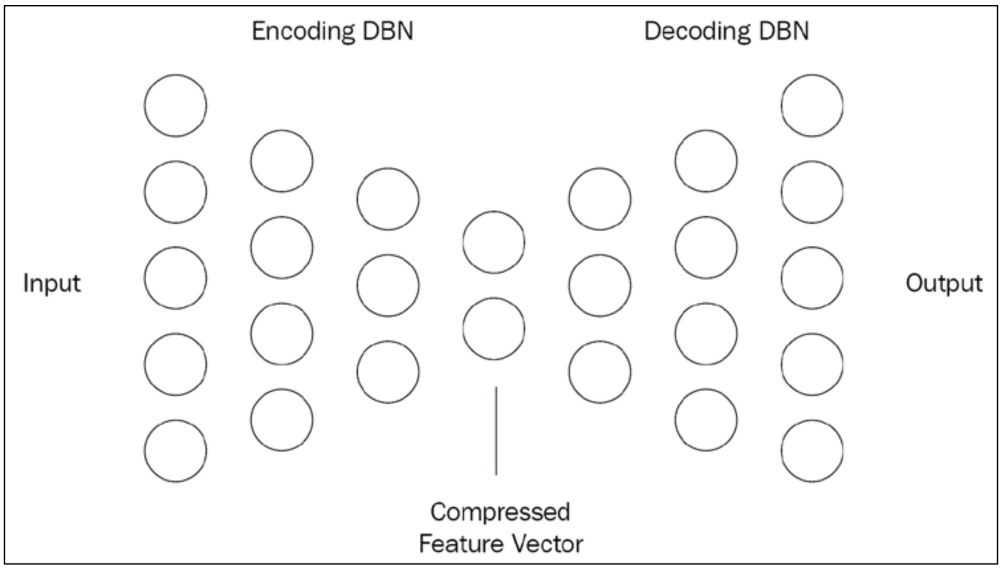


Figure 1.12: Structure of an autoencoder

Recurrent neural networks

RNNs are one of the most common deep learning algorithms, and they took the whole world by storm. Almost all the state-of-the-art performance we have now in natural language processing or understanding is because of a variant of RNNs. In recurrent networks, you try to identify the smallest unit in your data and make your data a group of those units. In the example of natural language, the most common approach is to make one word a unit and consider the sentence as a group of words while processing it. You unfold your RNN for the whole sentence and process your sentence one word at a time. RNNs have variants that work for different datasets and sometimes, efficiency can be taken into account while choosing the variant. Long **short-term memory** (**LSTM**) and **gated recurrent units** (**GRUs**) cells are the most common RNN units.

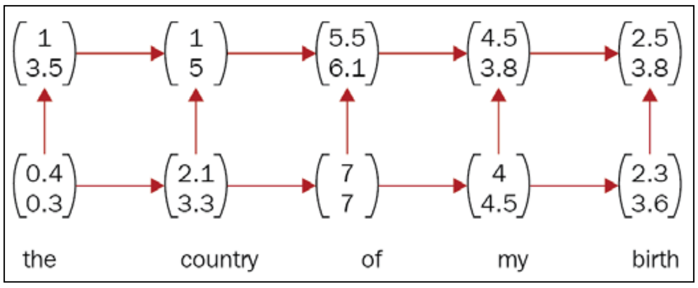


Figure 1.13: A vector representation of words in a recurrent network