Daniel Voigt Godoy

Deep Learning with PyTorch Step-by-Step

A Beginner's Guide

Deep Learning with PyTorch Step-by-Step

A Beginner's Guide

Daniel Voigt Godoy

Version 1.2

Deep Learning with PyTorch Step-by-Step: A Beginner’s Guide

by Daniel Voigt Godoy

May 2021: First Edition

Revision History for the First Edition:

- 2021-05-18: v1.0
- 2021-12-15: v1.1
- 2022-02-12: v1.1.1
- 2024-07-29: v1.2

This book is for sale at http://leanpub.com/pytorch. For more information, please send an email to contact@dvgodoy.com

"*What I cannot create, I do not understand.*"

Richard P. Feynman

Table of Contents

Preface

If you're reading this, I probably don't need to tell you that deep learning is amazing and PyTorch is cool, right?

But I will tell you, briefly, how this book came to be. In 2016, I started teaching a class on machine learning with Apache Spark and, a couple of years later, another class on the fundamentals of machine learning.

At some point, I tried to find a blog post that would visually explain, in a clear and concise manner, the concepts behind binary cross-entropy so that I could show it to my students. Since I could not find any that fit my purpose, I decided to write one myself. Although I thought of it as a fairly basic topic, it turned out to be my most popular blog post[1]! My readers have welcomed the simple, straightforward, and conversational way I explained the topic.

Then, in 2019, I used the same approach for writing another blog post: "Understanding PyTorch with an example: a step-by-step tutorial."[2] Once again, I was amazed by the reaction from the readers!

It was their positive feedback that motivated me to write this book to help beginners start their journey into deep learning and PyTorch. I hope you enjoy reading this book as much as I enjoyed writing it.

[1] https://bit.ly/2UW5iTg

[2] https://bit.ly/2TpzwxR

Acknowledgements

First and foremost, I'd like to thank YOU, my reader, for making this book possible. If it weren't for the amazing feedback I got from the thousands of readers of my blog post about PyTorch, I would have never mustered the strength to start *and finish* such a major undertaking as writing a 1,000-page book!

I'd like to thank my good friends Jesús Martínez-Blanco (who managed to read absolutely *everything* that I wrote), Jakub Cieslik, Hannah Berscheid, Mihail Vieru, Ramona Theresa Steck, Mehdi Belayet Lincon, and António Góis for helping me out and dedicating a good chunk of their time to reading, proofing, and suggesting improvements to my drafts. I'm forever grateful for your support! I'd also like to thank my friend José Luis Lopez Pino for the initial push I needed to actually *start* writing this book.

Many thanks to my friends José Quesada and David Anderson for taking me as a student at the Data Science Retreat in 2015 and, later on, for inviting me to be a teacher there. That was the starting point of my career both as a data scientist and as teacher.

I'd also like to thank the PyTorch developers for developing such an amazing framework, and the teams from Leanpub and Towards Data Science for making it incredibly easy for content creators like me to share their work with the community.

Finally, I'd like to thank my wife, Jerusa, for always being supportive throughout the entire writing of this book, and for taking the time to read *every* single page in it :-)

About the Author

Daniel is a data scientist, developer, writer, and teacher. He has been teaching machine learning and distributed computing technologies at Data Science Retreat, the longest-running Berlin-based bootcamp, since 2016, helping more than 150 students advance their careers.

Daniel is also the main contributor of two Python packages: HandySpark and DeepReplay.

His professional background includes 20 years of experience working for companies in several industries: banking, government, fintech, retail, and mobility.

Frequently Asked Questions (FAQ)

Why PyTorch?

First, coding in PyTorch is **fun** :-) Really, there is something to it that makes it very enjoyable to write code in. Some say it is because it is very **pythonic**, or maybe there is something else, who knows? I hope that, by the end of this book, you feel like that too!

Second, maybe there are even some *unexpected benefits* to your health—check Andrej Karpathy's tweet[3] about it!

Jokes aside, PyTorch is the **fastest-growing**[4] framework for developing deep learning models and it has a **huge ecosystem**.[5] That is, there are many *tools* and *libraries* developed on top of PyTorch. It is the **preferred framework**[6] in academia already and is making its way in the industry.

Several companies are already powered by PyTorch;[7] to name a few:

- **Facebook**: The company is the original developer of PyTorch, released in October 2016.
- **Tesla**: Watch Andrej Karpathy (AI director at Tesla) speak about "*how Tesla is using PyTorch to develop full self-driving capabilities for its vehicles*" in this video.[8]
- **OpenAI**: In January 2020, OpenAI decided to standardize its deep learning framework on PyTorch (source[9]).
- **fastai**: fastai is a library[10] built on top of PyTorch to simplify model training and is used in its "*Practical Deep Learning for Coders*"[11] course. The fastai library is deeply connected to PyTorch and "*you can't become really proficient at using fastai if you don't know PyTorch well, too.*"[12]
- **Uber**: The company is a significant contributor to PyTorch's ecosystem, having developed libraries like Pyro[13] (probabilistic programming) and Horovod[14] (a distributed training framework).
- **Airbnb**: PyTorch sits at the core of the company's dialog assistant for customer service.(source[15])

This book **aims to get you started with PyTorch** while giving you a **solid understanding of how it works**.

Why This Book?

If you’re looking for a book where you can **learn about deep learning and PyTorch** without having to spend hours deciphering cryptic text and code, and one that’s **easy and enjoyable to read**, this is it :-)

The book covers from the **basics of gradient descent** all the way up to **fine-tuning large NLP models** (BERT and GPT-2) using HuggingFace. It is divided into four parts:

- **Part I**: Fundamentals (gradient descent, training linear and logistic regressions in PyTorch)
- **Part II**: Computer Vision (deeper models and activation functions, convolutions, transfer learning, initialization schemes)
- **Part III**: Sequences (RNN, GRU, LSTM, seq2seq models, attention, self-attention, Transformers)
- **Part IV**: Natural Language Processing (tokenization, embeddings, contextual word embeddings, ELMo, BERT, GPT-2)

This is **not** a typical book: most tutorials *start* with some nice and pretty *image classification problem* to illustrate how to use PyTorch. It may seem cool, but I believe it **distracts** you from the **main goal**: learning **how PyTorch works**. In this book, I present a **structured**, **incremental**, and **from-first-principles** approach to learn PyTorch.

Moreover, this is **not** a **formal book** in any way: I am writing this book **as if I were having a conversation with you**, the reader. I will ask you **questions** (and give you answers shortly afterward), and I will also make (silly) **jokes**.

My job here is to make you **understand** the topic, so I will **avoid fancy mathematical notation** as much as possible and **spell it out in plain English**.

In this book, I will **guide** you through the **development** of many models in PyTorch, showing you why PyTorch makes it much **easier** and more **intuitive** to build models in Python: *autograd*, *dynamic computation graph*, *model classes*, and much, much more.

We will build, **step-by-step**, not only the models themselves but also your **understanding** as I show you both the **reasoning** behind the code and **how to avoid** some **common pitfalls** and **errors** along the way.

There is yet another advantage of **focusing on the basics**: this book is likely to have a **longer shelf life**. It is fairly common for technical books, especially those focusing on cutting-edge technology, to become outdated quickly. Hopefully, this is not going to be the case here, since the **underlying mechanics are not changing, and neither are the concepts**. It is expected that some syntax changes over time, but I do not see backward compatibility breaking changes coming anytime soon.

> **One more thing**: If you hadn't noticed already, I **really** like to make use of **visual cues**, that is, **bold** and *italic* highlights. I firmly believe this helps the reader to **grasp** the **key ideas** I am trying to convey in a sentence more easily. You can find more on that in the section "**How to Read This Book.**"

Who Should Read This Book?

I wrote this book for **beginners in general**—not only PyTorch beginners. Every now and then, I will spend some time explaining some **fundamental concepts** that I believe are **essential** to have a proper **understanding of what's going on in the code**.

The best example is **gradient descent**, which most people are familiar with at some level. Maybe you know its general idea, perhaps you've seen it in Andrew Ng's Machine Learning course, or maybe you've even **computed some partial derivatives yourself**!

In real life, the **mechanics** of gradient descent will be **handled automatically by PyTorch** (uh, spoiler alert!). But, I will walk you through it anyway (unless you choose to skip Chapter 0 altogether, of course), because lots of **elements in the code**, as well as **choices of hyper-parameters** (learning rate, mini-batch size, etc.), can be much more easily understood if you know **where they come from**.

Maybe you already know some of these concepts well: If this is the case, you can simply *skip* them, since I've made these explanations as *independent* as possible from the rest of the content.

But **I want to make sure everyone is on the same page**, so, if you have just heard about a given concept or if you are unsure if you have entirely understood it, these explanations are for you.

What Do I Need to Know?

This is a book for beginners, so I am assuming as **little prior knowledge** as possible; as mentioned in the previous section, I will take the time to explain fundamental concepts whenever needed.

That being said, this is what I expect from you, the reader:

- to be able to code in **Python** (if you are familiar with object-oriented programming [OOP], even better)
- to be able to work with PyData stack (**numpy**, **matplotlib**, and **pandas**) and **Jupyter notebooks**
- to be familiar with some basic concepts used in **machine learning**, like:
    - supervised learning: regression and classification
    - loss functions for regression and classification (mean squared error, cross-entropy, etc.)
    - training-validation-test split
    - underfitting and overfitting (bias-variance trade-off)

Even so, I am still briefly touching on **some** of these topics, but I need to draw a line somewhere; otherwise, this book would be gigantic!

How to Read This Book

Since this book is a **beginner's guide**, it is meant to be read **sequentially**, as ideas and concepts are progressively built. The same holds true for the **code** inside the book—you should be able to *reproduce* all outputs, provided you execute the chunks of code in the same order as they are introduced.

This book is **visually** different than other books: As I've mentioned already in the "**Why This Book?**" section, I **really** like to make use of **visual cues**. Although this is not, *strictly speaking*, a **convention**, this is how you can interpret those cues:

- I use **bold** to highlight what I believe to be the **most relevant words** in a sentence or paragraph, while *italicized* words are considered *important* too (not important enough to be bold, though)
- *Variables*, *coefficients*, and *parameters* in general, are *italicized*

- `Classes` and `methods` are written in a `monospaced` font, and they link to PyTorch [16] documentation the first time they are introduced, so you can easily follow it (unlike other links in this book, links to documentation are *numerous* and thus *not* included in the footnotes)
- Every **code cell** is followed by *another* cell showing the corresponding **outputs** (if any)
- **All code** presented in the book is available at its **official repository** on GitHub:

https://github.com/dvgodoy/PyTorchStepByStep

Code cells with **titles** are an important piece of the workflow:

Title Goes Here

```python
# Whatever is being done here is going to impact OTHER code
# cells. Besides, most cells have COMMENTS explaining what
# is happening
x = [1., 2., 3.]
print(x)
```

If there is any output to the code cell, titled or not, there *will* be another code cell depicting the corresponding **output** so you can *check* if you successfully reproduced it or not.

Output

```
[1.0, 2.0, 3.0]
```

Some code cells **do not** have titles—running them does not affect the workflow:

```python
# Those cells illustrate HOW TO CODE something, but they are
# NOT part of the main workflow
dummy = ['a', 'b', 'c']
print(dummy[::-1])
```

But even these cells have their outputs shown!

Output

```
['c', 'b', 'a']
```

I use asides to communicate a variety of things, according to the corresponding icon:

WARNING

Potential **problems** or things to **look out** for.

TIP

Key aspects I really want you to **remember**.

INFORMATION

Important information to **pay attention** to.

IMPORTANT

Really important information to **pay attention** to.

TECHNICAL

Technical aspects of **parameterization** or **inner workings of algorithms**.

QUESTION AND ANSWER

Asking myself **questions** (pretending to be you, the reader) and answering them, either in the same block or shortly after.

DISCUSSION

Really brief discussion on a concept or topic.

LATER

Important topics that will be covered in more detail later.

SILLY

Jokes, puns, memes, quotes from movies.

What’s Next?

It’s time to **set up** an environment for your learning journey using the **Setup Guide**.

[3] https://bit.ly/2MQoYRo

[4] https://bit.ly/37uZgLB

[5] https://pytorch.org/ecosystem/

[6] https://bit.ly/2MTN0Lh

[7] https://bit.ly/2UFHFve

[8] https://bit.ly/2XXJkyo

[9] https://openai.com/blog/openai-pytorch/

[10] https://docs.fast.ai/

[11] https://course.fast.ai/

[12] https://course.fast.ai/

[13] http://pyro.ai/

[14] https://github.com/horovod/horovod

[15] https://bit.ly/30CPhm5

[16] https://bit.ly/3cT1aH2

Setup Guide

Official Repository

This book's official repository is available on GitHub:

https://github.com/dvgodoy/PyTorchStepByStep

It contains **one Jupyter notebook** for every **chapter** in this book. Each notebook contains **all the code shown** in its corresponding chapter, and you should be able to **run its cells in sequence** to get the **same outputs**, as shown in the book. I strongly believe that being able to **reproduce the results** brings **confidence** to the reader.

Even though I did my best to ensure the **reproducibility** of the results, you may **still** find some minor differences in your outputs (especially during model training). Unfortunately, completely reproducible results are not guaranteed across PyTorch releases, and results may not be reproducible between CPU and GPU executions, even when using identical seeds.[17]

Environment

There are **three options** for you to run the Jupyter notebooks:

- Google Colab (*https://colab.research.google.com*)
- Binder (*https://mybinder.org*)
- Local Installation

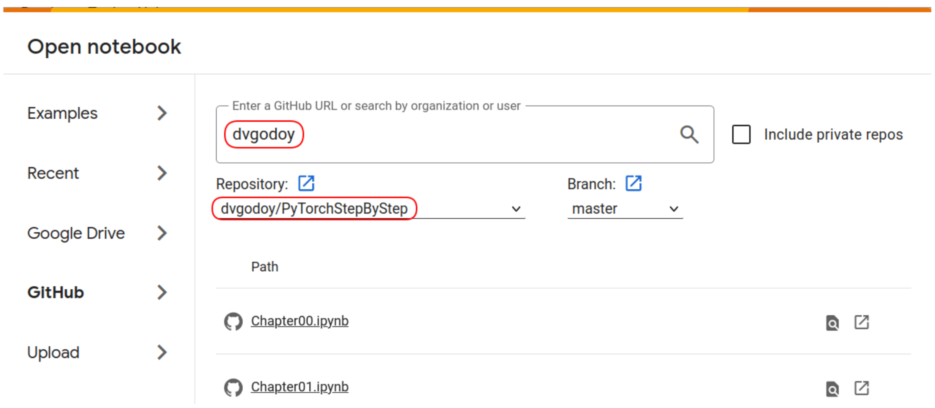
Let's briefly explore the **pros** and **cons** of each of these options.

Google Colab

Google Colab "*allows you to write and execute Python in your browser, with zero configuration required, free access to GPUs and easy sharing.*"[18].

You can easily **load notebooks directly from GitHub** using Colab's special URL (*https://colab.research.google.com/github/*). Just type in the GitHub's user or organization (like mine, `dvgodoy`), and it will show you a list of all its public repositories (like this book's, `PyTorchStepByStep`).

After choosing a repository, it will list the available notebooks and corresponding links to open them in a new browser tab.



dvgodoy

Repository: dvgodoy/PyTorchStepByStep

Branch: master

Chapter00.ipynb

Chapter01.ipynb

Figure S.1 - Google Colab's special URL

You also get access to a **GPU**, which is very useful to train deep learning models **faster**. More important, if you **make changes** to the notebook, Google Colab will **keep them**. The whole setup is very convenient; the only **cons** I can think of are:

- You need to be **logged in** to a Google account.
- You need to (re)install Python packages that are not part of Google Colab's default configuration.

Binder

Binder "*allows you to create custom computing environments that can be shared and used by many remote users.*"[19]

You can also **load notebooks directly from GitHub**, but the process is slightly different. Binder will create something like a *virtual machine* (technically, it is a container, but let's leave it at that), clone the repository, and start Jupyter. This allows you to have access to **Jupyter's home page** in your browser, just like you would if you were running it locally, but everything is running in a JupyterHub server on their end.

Just go to Binder's site (*https://mybinder.org/*) and type in the URL to the GitHub repository you want to explore (for instance, `https://github.com/dvgodoy/PyTorchStepByStep`) and click on **Launch**. It will take a couple of minutes to build the image and open Jupyter's home page.

You can also **launch Binder** for this book's repository directly using the following link: *https://mybinder.org/v2/gh/dvgodoy/PyTorchStepByStep/master*.

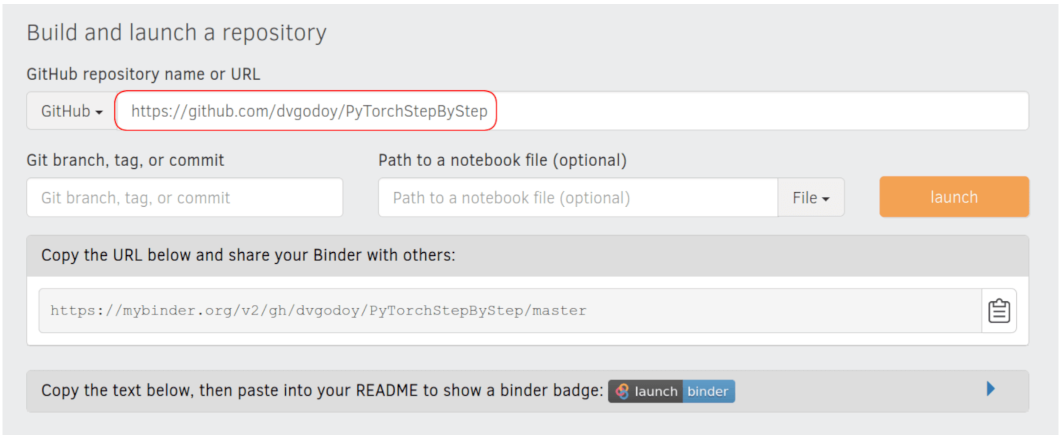

Build and launch a repository
GitHub repository name or URL
GitHub
https://github.com/dvgodoy/PyTorchStepByStep
Git branch, tag, or commit
Path to a notebook file (optional)
File
launch
Copy the URL below and share your Binder with others:
https://mybinder.org/v2/gh/dvgodoy/PyTorchStepByStep/master
Copy the text below, then paste into your README to show a binder badge:
launch binder

Figure S.2 - Binder's page

Binder is very convenient since it **does not require a prior setup** of any kind. Any Python packages needed to successfully run the environment are likely installed during launch (if provided by the author of the repository).

On the other hand, it may **take time** to start, and it **does not keep your changes** after your session expires (so, make sure you **download** any notebooks you modify).

Local Installation

This option will give you more **flexibility**, but it will require **more effort to set up**. I encourage you to try setting up your own environment. It may seem daunting at first, but you can surely accomplish it by following **seven easy steps**:

Checklist

- ☐ 1. Install **Anaconda**.
- ☐ 2. Create and activate a **virtual environment**.
- ☐ 3. Install **PyTorch** package.
- ☐ 4. Install **TensorBoard** package.
- ☐ 5. Install **GraphViz** software and **TorchViz** package (**optional**).
- ☐ 6. Install **git** and **clone** the repository.
- ☐ 7. Start **Jupyter** notebook.

1. Anaconda

If you don't have **Anaconda's Individual Edition**[20] installed yet, this would be a good time to do it. It is a convenient way to start since it contains most of the Python libraries a data scientist will ever need to develop and train models.

Please follow the **installation instructions** for your OS:

- Windows (*https://docs.anaconda.com/anaconda/install/windows/*)
- macOS (*https://docs.anaconda.com/anaconda/install/mac-os/*)
- Linux (*https://docs.anaconda.com/anaconda/install/linux/*)

> Make sure you choose **Python 3.X** version since Python 2 was discontinued in January 2020.

After installing Anaconda, it is time to create an **environment**.

2. Conda (Virtual) Environments

Virtual environments are a convenient way to isolate Python installations associated with different projects.

> "*What is an environment?*"

It is pretty much a **replication of Python itself and some (or all) of its libraries**, so, effectively, you'll end up with multiple Python installations on your computer.

> "*Why can't I just use one single Python installation for everything?*"

With so many independently developed Python **libraries**, each having many different **versions** and each version having various **dependencies** (on other libraries), **things can get out of hand** real quick.

It is beyond the scope of this guide to debate these issues, but take my word for it (or Google it!)—you'll benefit a great deal if you pick up the habit of **creating a different environment for every project you start working on**.

> "*How do I create an environment?*"

First, you need to choose a **name** for your environment :-) Let's call ours `pytorchbook` (or anything else you find easy to remember). Then, you need to open a **terminal** (in Ubuntu) or **Anaconda Prompt** (in Windows or macOS) and type the following command:

```
$ conda create -n pytorchbook anaconda
```

The command above creates a Conda environment named `pytorchbook` and includes **all Anaconda packages** in it (time to get a coffee, it will take a while…). If you want to learn more about creating and using Conda environments, please check Anaconda's "Managing Environments"[21] user guide.

Did it finish creating the environment? Good! It is time to **activate it**, meaning, making **that Python installation** the one to be used now. In the same terminal (or Anaconda prompt), just type:

```
$ conda activate pytorchbook
```

Your prompt should look like this (if you're using Linux):

```
(pytorchbook)$
```

or like this (if you're using Windows):

```
(pytorchbook)C:\>
```

Done! You are using a **brand new Conda environment** now. You'll need to **activate it** every time you open a new terminal, or, if you're a Windows or macOS user, you can open the corresponding Anaconda prompt (it will show up as **Anaconda Prompt (pytorchbook)**, in our case), which will have it activated from the start.

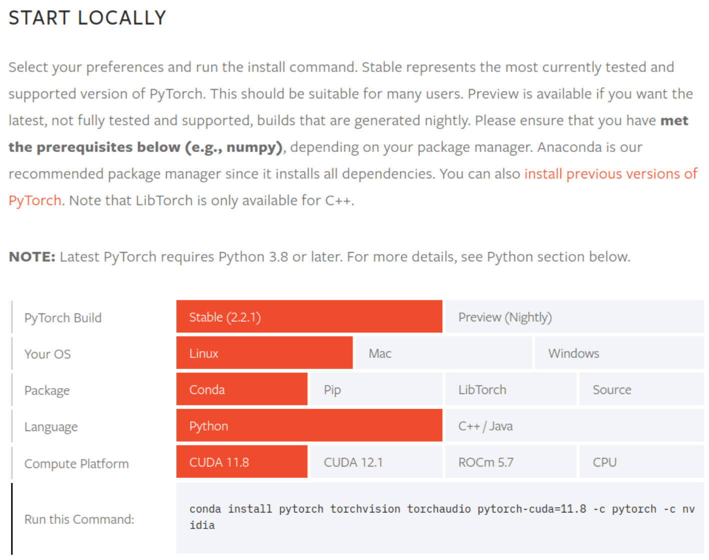
IMPORTANT: From now on, I am assuming you'll activate the `pytorchbook` environment every time you open a terminal or Anaconda prompt. Further installation steps **must** be executed inside the environment.

3. PyTorch

PyTorch is the coolest **deep learning framework**, just in case you skipped the introduction.

It is "*an open source machine learning framework that accelerates the path from research prototyping to production deployment.*"[22] Sounds good, right? Well, I probably don't have to convince you at this point :-)

It is time to install the star of the show :-) We can go straight to the **Start Locally** (*https://pytorch.org/get-started/locally/*) section of PyTorch's website, and it will automatically select the options that best suit your local environment, and it will show you the **command to run**.



START LOCALLY

Select your preferences and run the install command. Stable represents the most currently tested and supported version of PyTorch. This should be suitable for many users. Preview is available if you want the latest, not fully tested and supported, 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++.

NOTE: Latest PyTorch requires Python 3.8 or later. For more details, see Python section below.

PyTorch Build	[✓ Stable (2.2.1)]	Preview (Nightly)		
Your OS	[✓ Linux]	Mac	Windows	
Package	[✓ Conda]	Pip	LibTorch	Source
Language	[✓ Python]	C++ / Java		
Compute Platform	[✓ CUDA 11.8]	CUDA 12.1	ROCm 5.7	CPU

Run this Command:

```
conda install pytorch torchvision torchaudio pytorch-cuda=11.8 -c pytorch -c nv
idia
```

Figure S.3 - PyTorch's Start Locally page

Some of these options are given:

- PyTorch Build: Always select a **Stable** version.
- Package: I am assuming you're using **Conda**.
- Language: Obviously, **Python**.

So, two options remain: **Your OS** and **CUDA**.

"*What is CUDA?*" you ask.

Using GPU / CUDA

CUDA "*is a parallel computing platform and programming model developed by NVIDIA for general computing on graphical processing units (GPUs).*"[23]

If you have a **GPU** in your computer (likely a *GeForce* graphics card), you can leverage its power to train deep learning models **much faster** than using a CPU. In this case, you should choose a PyTorch installation that includes CUDA support.

This is not enough, though: If you haven't done so yet, you need to install up-to-date drivers, the CUDA Toolkit, and the CUDA Deep Neural Network library (cuDNN). Unfortunately, more detailed installation instructions for CUDA are outside the scope of this book.

The **advantage** of using a GPU is that it allows you to **iterate faster** and **experiment with more-complex models and a more extensive range of hyper-parameters**.

In my case, I use **Linux**, and I have a **GPU** with CUDA version 11.8 installed. So I would run the following command in the **terminal** (after activating the environment):

```
(pytorchbook)$ conda install pytorch torchvision\
torchaudio pytorch-cuda=11.8 -c pytorch -c nvidia
```

Using CPU

If you **do not** have a **GPU**, you should choose **None** for CUDA.

"*Would I be able to run the code **without** a GPU?*" you ask.

Sure! The code and the examples in this book were designed to allow **all readers** to follow them promptly. Some examples may demand a bit more computing power, but we are talking about a **couple of minutes** in a CPU, not hours. If you do not have a GPU, **don't worry**! Besides, you can always use Google Colab if you need to use a GPU for a while!

If I had a **Windows** computer, and **no GPU**, I would have to run the following command in the **Anaconda prompt (pytorchbook)**:

```
(pytorchbook) C:\> conda install pytorch torchvision\
torchaudio cpuonly -c pytorch
```

Installing CUDA

CUDA: Installing drivers for a GeForce graphics card, NVIDIA's cuDNN, and CUDA Toolkit can be challenging and is highly dependent on which model you own and which OS you use.

For installing GeForce's drivers, go to GeForce's website (*https://www.geforce.com/drivers*), select your OS and the model of your graphics card, and follow the installation instructions.

For installing NVIDIA's CUDA Deep Neural Network library (cuDNN), you need to register at *https://developer.nvidia.com/cudnn*.

For installing CUDA Toolkit (*https://developer.nvidia.com/cuda-toolkit*), please follow instructions for your OS and choose a local installer or executable file.

macOS: If you're a macOS user, please beware that PyTorch's binaries **DO NOT** support **CUDA**, meaning you'll need to install PyTorch **from source** if you want to use your GPU. This is a somewhat **complicated** process (as described in *https://github.com/pytorch/pytorch#from-source*), so, if you don't feel like doing it, you can choose to proceed **without CUDA**, and you'll still be able to execute the code in this book promptly.

4. TensorBoard

TensorBoard is TensorFlow's **visualization toolkit**, and "*provides the visualization and tooling needed for machine learning experimentation.*"[24]

TensorBoard is a powerful tool, and we can use it even if we are developing models in PyTorch. Luckily, you don't need to install the whole TensorFlow to get it; you can easily **install TensorBoard alone** using **Conda**. You just need to run this command in your **terminal** or **Anaconda prompt** (again, after activating the environment):

```
(pytorchbook)$ conda install -c conda-forge tensorboard
```

5. GraphViz and Torchviz (optional)

This step is optional, mostly because the installation of GraphViz can sometimes be *challenging* (especially on Windows). If for any reason you do not succeed in installing it correctly, or if you decide to skip this installation step, you will still be **able to execute the code in this book** (except for a couple of cells that generate images of a model's structure in the "Dynamic Computation Graph" section of Chapter 1).

GraphViz is an open source graph visualization software. It is "*a way of representing structural information as diagrams of abstract graphs and networks.*"[25]

We need to install GraphViz to use **TorchViz**, a neat package that allows us to visualize a model's structure. Please check the **installation instructions** for your OS at *https://www.graphviz.org/download/*.

If you are using Windows, please use the **GraphViz's Windows Package** installer at *https://graphviz.gitlab.io/_pages/Download/windows/graphviz-2.38.msi*.

You also need to add GraphViz to the PATH (environment variable) in Windows. Most likely, you can find the GraphViz executable file at `C:\ProgramFiles(x86)\Graphviz2.38\bin`. Once you find it, you need to set or change the PATH accordingly, adding GraphViz's location to it.

For more details on how to do that, please refer to "How to Add to Windows PATH Environment Variable."[26]

For additional information, you can also check the "How to Install Graphviz Software"[27] guide.

After installing GraphViz, you can install the **torchviz**[28] package. This package is **not** part of Anaconda Distribution Repository[29] and is only available at **PyPI**[30], the Python Package Index, so we need to `pip install` it.

Once again, open a **terminal** or **Anaconda prompt** and run this command (just once more: after activating the environment):

```
(pytorchbook)$ pip install torchviz
```

To check your GraphViz / TorchViz installation, you can try the Python code below:

```
(pytorchbook)$ python

Python 3.9.0 (default, Nov 15 2020, 14:28:56)
[GCC 7.3.0] :: Anaconda, Inc. on linux
Type "help", "copyright", "credits" or "license" for more
information.
>>> import torch
>>> from torchviz import make_dot
>>> v = torch.tensor(1.0, requires_grad=True)
>>> make_dot(v)
```

If everything is **working correctly**, you should see something like this:

Output

```
<graphviz.dot.Digraph object at 0x7ff540c56f50>
```

If you get an **error** of any kind (the one below is pretty common), it means there is still some kind of **installation issue** with GraphViz.

Output

```
ExecutableNotFound: failed to execute ['dot', '-Tsvg'], make
sure the Graphviz executables are on your systems' PATH
```

6. Git

It is *way* beyond this guide's scope to introduce you to version control and its most popular tool: `git`. If you are familiar with it already, great, you can skip this section altogether!

Otherwise, I'd recommend you to learn more about it; it will **definitely** be useful for you later down the line. In the meantime, I will show you the bare minimum so you can use `git` to **clone the repository** containing all code used in this book and get your **own**, **local copy** of it to modify and experiment with as you please.

First, you need to install it. So, head to its downloads page (*https://git-scm.com/downloads*) and follow instructions for your OS. Once the installation is complete, please open a **new terminal** or **Anaconda prompt** (it's OK to close the previous one). In the new terminal or Anaconda prompt, you should be able to **run `git` commands**.

To clone this book's repository, you only need to run:

```
(pytorchbook)$ git clone https://github.com/dvgodoy/\
PyTorchStepByStep.git
```

The command above will create a `PyTorchStepByStep` folder that contains a local copy of everything available on GitHub's repository.

conda install vs pip install

Although they may seem equivalent at first sight, you should **prefer `conda install`** over `pip install` when working with Anaconda and its virtual environments.

This is because `conda install` is sensitive to the active virtual environment: The package will be installed only for that environment. If you use `pip install`, and `pip` itself is not installed in the active environment, it will fall back to the **global `pip`**, and you definitely **do not** want that.

Why not? Remember the problem with **dependencies** I mentioned in the virtual environment section? That's why! The `conda` installer assumes it handles all packages that are part of its repository and keeps track of the complicated network of dependencies among them (to learn more about this, check this link[31]).

To learn more about the differences between `conda` and `pip`, read "Understanding Conda and Pip."[32]

As a rule, first try to `conda install` a given package and, only if it does not exist there, fall back to `pip install`, as we did with `torchviz`.

7. Jupyter

After cloning the repository, navigate to the `PyTorchStepByStep` folder and, **once inside it**, **start Jupyter** on your terminal or Anaconda prompt:

```
(pytorchbook)$ jupyter notebook
```

This will open your browser, and you will see **Jupyter's home page** containing the repository's notebooks and code.

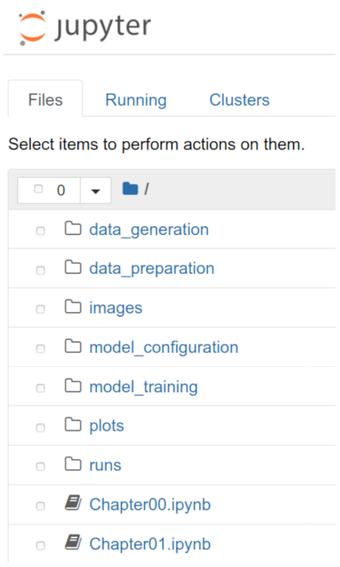


Figure S.4 - Running Jupyter

Moving On

Regardless of which of the three environments you chose, now you are ready to move on and tackle the development of your first PyTorch model, **step-by-step**!

[17] https://pytorch.org/docs/stable/notes/randomness.html

[18] https://colab.research.google.com/notebooks/intro.ipynb

[19] https://mybinder.readthedocs.io/en/latest/

[20] https://www.anaconda.com/products/individual

[21] https://bit.ly/2MVk0CM

[22] https://pytorch.org/

[23] https://developer.nvidia.com/cuda-zone

[24] https://www.tensorflow.org/tensorboard

[25] https://www.graphviz.org/

[26] https://bit.ly/3fIwYA5

[27] https://tinyurl.com/bde9vdn3

[28] https://github.com/szagoruyko/pytorchviz

[29] https://docs.anaconda.com/anaconda/packages/pkg-docs/

[30] https://pypi.org/

[31] https://bit.ly/37onBTt

[32] https://bit.ly/2AAh8J5

Part I

Fundamentals

Chapter 0
Visualizing Gradient Descent

Spoilers

In this chapter, we will:

- define a **simple linear regression model**
- walk through **every step of gradient descent**: initializing parameters, performing a forward pass, computing errors and loss, computing gradients, and updating parameters
- understand **gradients** using **equations**, **code**, and **geometry**
- understand the difference between **batch**, **mini-batch**, and **stochastic** gradient descent
- visualize the **effects on the loss** of using different **learning rates**
- understand the importance of **standardizing / scaling features**
- and much, much more!

There is **no** actual PyTorch code in this chapter… it is *Numpy* all along because our focus here is to understand, inside and out, how gradient descent works. PyTorch will be introduced in the next chapter.

Jupyter Notebook

The Jupyter notebook corresponding to Chapter 0[33] is part of the official ***Deep Learning with PyTorch Step-by-Step*** repository on GitHub. You can also run it directly in **Google Colab**[34].

If you're using a *local installation*, open your terminal or Anaconda prompt and navigate to the `PyTorchStepByStep` folder you cloned from GitHub. Then, *activate* the `pytorchbook` environment and run `jupyter notebook`:

```
$ conda activate pytorchbook

(pytorchbook)$ jupyter notebook
```

If you're using Jupyter's default settings, this link should open Chapter 0's notebook. If not, just click on `Chapter00.ipynb` on your Jupyter's home page.

Imports

For the sake of organization, all libraries needed throughout the code used in any given chapter are imported at its very beginning. For this chapter, we'll need the following imports:

```
import numpy as np
from sklearn.linear_model import LinearRegression
from sklearn.preprocessing import StandardScaler
```

Visualizing Gradient Descent

According to Wikipedia[35]: "***Gradient descent*** *is a first-order iterative optimization algorithm for finding a local minimum of a differentiable function.*"

But I would go with: "**Gradient descent** is an iterative technique commonly used in machine learning and deep learning to find the best possible set of parameters / coefficients for a given model, data points, and loss function, starting from an initial, and usually random, guess."

"*Why **visualizing** gradient descent?*"

I believe *the way gradient descent is usually explained lacks intuition*. Students and beginners are left with a *bunch of equations* and *rules of thumb*—**this is not the way one should learn such a fundamental topic**.

If you **really understand** how gradient descent works, you will also understand how the **characteristics of your data** and your **choice of hyper-parameters** (mini-batch size and learning rate, for instance) have an **impact** on how **well** and how **fast** the model training is going to be.

By *really understanding*, I do not mean working through the equations manually: this does not develop intuition either. I mean **visualizing** the effects of different settings; I mean **telling a story** to illustrate the concept. That's how you **develop intuition**.

That being said, I'll cover the **five basic steps** you'd need to go through to use gradient descent. I'll show you the corresponding *Numpy* code while explaining lots of **fundamental concepts** along the way.

But first, we need some **data** to work with. Instead of using some *external dataset*, let's

- define which **model** we want to train to better understand gradient descent; and
- **generate synthetic data** for that model.

Model

The model must be **simple** and **familiar**, so you can focus on the **inner workings** of gradient descent.

So, I will stick with a model as simple as it can be: a **linear regression with a single feature, *x***!

$$y = b + wx + \epsilon$$

Equation 0.1 - Simple linear regression model

In this model, we use a **feature (*x*)** to try to predict the value of a **label (*y*)**. There are three elements in our model:

- **parameter *b***, the *bias* (or *intercept*), which tells us the expected average value of *y* when *x* is zero
- **parameter *w***, the *weight* (or *slope*), which tells us how much *y* increases, on average, if we increase *x* by one unit
- and that **last term** (why does it *always* have to be a Greek letter?), *epsilon*, which is there to account for the inherent **noise**; that is, the **error** we cannot get rid of

We can also conceive the very same model structure in a less abstract way:

salary = minimum wage + increase per year * years of experience + noise

And to make it even more concrete, let's say that the **minimum wage** is **$1,000** (whatever the currency or time frame, this is not important). So, if you have **no experience**, your salary is going to be the **minimum wage** (parameter ***b***).

Also, let’s say that, **on average**, you get a **$2,000 increase** (parameter ***w***) for every year of experience you have. So, if you have **two years of experience**, you are expected to earn a salary of **$5,000**. But your actual salary is **$5,600** (lucky you!). Since the model cannot account for those **extra $600**, your extra money is, technically speaking, **noise**.

Data Generation

We know our model already. In order to generate **synthetic data** for it, we need to pick values for its **parameters**. I chose ***b* = 1** and ***w* = 2** (as in thousands of dollars) from the example above.

First, let’s generate our **feature (*x*)**: We use *Numpy*'s `rand()` method to randomly generate 100 (*N*) points between 0 and 1.

Then, we plug our **feature (*x*)** and our **parameters *b* and *w*** into our **equation** to compute our **labels (*y*)**. But we need to add some **Gaussian noise**[36] **(*epsilon*)** as well; otherwise, our synthetic dataset would be a perfectly straight line. We can generate noise using *Numpy*'s `randn()` method, which draws samples from a normal distribution (of mean 0 and variance 1), and then multiply it by a **factor** to adjust for the **level of noise**. Since I don’t want to add too much noise, I picked 0.1 as my factor.

Synthetic Data Generation

Data Generation

```
1 true_b = 1
2 true_w = 2
3 N = 100
4
5 # Data Generation
6 np.random.seed(42)
7 x = np.random.rand(N, 1)
8 epsilon = (.1 * np.random.randn(N, 1))
9 y = true_b + true_w * x + epsilon
```

Did you notice the `np.random.seed(42)` at line 6? This line of code is actually more important than it looks. It guarantees that, every time we run this code, the **same random numbers will be generated**.

"*Wait; what?! Aren't the numbers supposed to be **random**? How could they possibly be the **same** numbers?*" you ask, perhaps even a bit annoyed by this.

(Not So) Random Numbers

Well, you know, random numbers are not **quite** random… They are really **pseudo-random**, which means *Numpy*'s number generator spits out a **sequence of numbers** that **looks like it's random**. But it is not, really.

The **good** thing about this behavior is that we can tell the generator to **start a particular sequence of pseudo-random numbers**. To some extent, it works as if we tell the generator: "*please generate sequence #42*," and it will spill out a sequence of numbers. That number, 42, which works like the *index* of the sequence, is called a **seed**. Every time we give it the **same seed**, it generates the **same numbers**.

This means we have the **best of both worlds**: On the one hand, we do **generate** a sequence of numbers that, for all intents and purposes, is **considered to be random**; on the other hand, we have the **power to reproduce any given sequence**. I cannot stress enough how convenient that is for **debugging** purposes!

Moreover, you can guarantee that **other people will be able to reproduce your results**. Imagine how annoying it would be to run the code in this book and get different outputs every time, having to wonder if there is anything wrong with it. But since I've set a seed, you and I can achieve the very same outputs, even if it involved generating random data!

Next, let's **split** our synthetic data into **train** and **validation** sets, shuffling the array of indices and using the first 80 shuffled points for training.

"*Why do you need to **shuffle** randomly generated data points? Aren't they random enough?*"

Yes, they **are** random enough, and shuffling them is indeed redundant in this example. But it is best practice to **always shuffle** your data points before training a model to improve the performance of gradient descent.

There is one **exception** to the "always shuffle" rule, though: **time series** problems, where shuffling can lead to data leakage.

Train-Validation-Test Split

It is beyond the scope of this book to explain the reasoning behind the **train-validation-test split**, but there are two points I’d like to make:

1. The split should **always** be the **first thing** you do—no preprocessing, no transformations; **nothing happens before the split**. That’s why we do this **immediately after the synthetic data generation**.
2. In this chapter we will use **only the training set**, so I did not bother to create a **test set**, but I performed a split nonetheless to **highlight point #1** :-)

Train-Validation Split

```
# Shuffles the indices
idx = np.arange(N)
np.random.shuffle(idx)

# Uses first 80 random indices for train
train_idx = idx[:int(N*.8)]
# Uses the remaining indices for validation
val_idx = idx[int(N*.8):]

# Generates train and validation sets
x_train, y_train = x[train_idx], y[train_idx]
x_val, y_val = x[val_idx], y[val_idx]
```

"*Why didn’t you use* `train_test_split()` *from Scikit-Learn?*" you may be asking.

That’s a fair point. Later on, we will refer to the **indices of the data points** belonging to either train or validation sets, instead of the points themselves. So, I thought of using them from the very start.

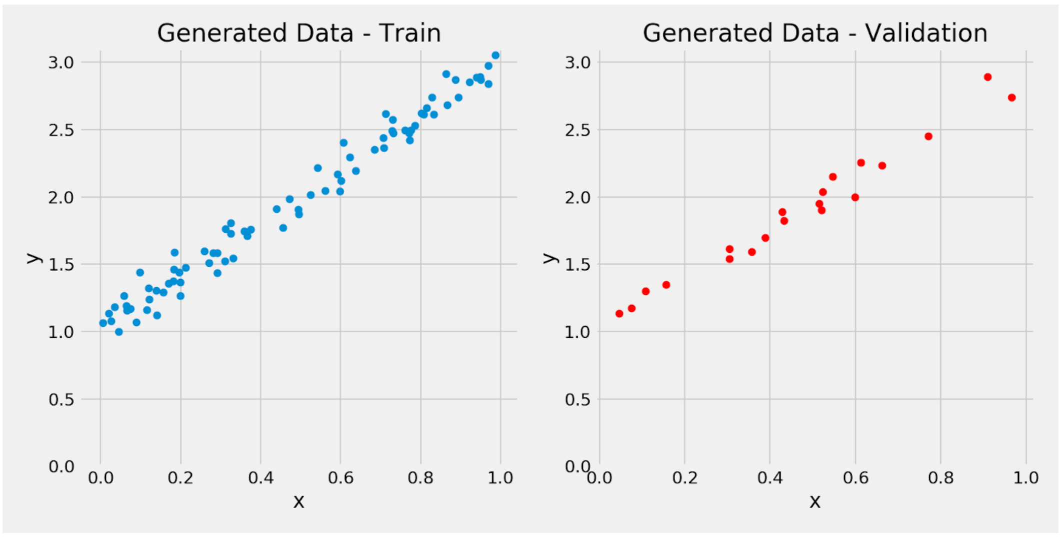


Figure 0.1 - Synthetic data: train and validation sets

We **know** that ***b* = 1**, ***w* = 2**, but now let's see **how close** we can get to the true values by using **gradient descent** and the 80 points in the **training set** (for training, ***N* = 80**).

Step 0 - Random Initialization

In our example, we already **know** the **true values** of the **parameters**, but this will obviously never happen in real life: If we *knew* the true values, why even bother to train a model to find them?!

OK, given that **we'll never know** the **true values** of the parameters, we need to set **initial values** for them. How do we choose them? It turns out a **random guess** is as good as any other.

Even though the initialization is **random**, there are some clever **initialization schemes** that should be used when training more-complex models. We'll get back to them (much) later.

For training a model, you need to **randomly initialize the parameters / weights** (we have only two, ***b*** and ***w***).

Random Initialization

```
# Step 0 - Initializes parameters "b" and "w" randomly
np.random.seed(42)
b = np.random.randn(1)
w = np.random.randn(1)

print(b, w)
```

Output

```
[0.49671415] [-0.1382643]
```

Step 1 - Compute Model's Predictions

This is the **forward pass**; it simply *computes the model's predictions using the current values of the parameters / weights*. At the very beginning, we will be producing **really bad predictions**, as we started with **random values in Step 0**.

Step 1

```
# Step 1 - Computes our model's predicted output - forward pass
yhat = b + w * x_train
```

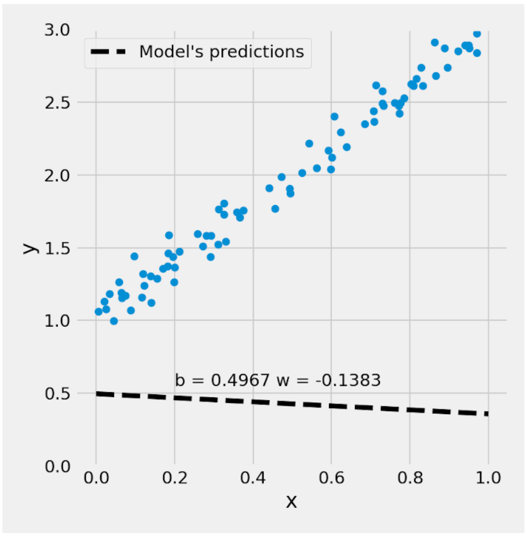


Figure 0.2 - Model's predictions (with random parameters)

Step 2 - Compute the Loss

There is a subtle but fundamental difference between **error** and **loss**.

The **error** is the difference between the **actual value (label)** and the **predicted value** computed for a single data point. So, for a given *i*-th point (from our dataset of *N* points), its error is:

$$\text{error}_i = \hat{y_i} - y_i$$

Equation 0.2 - Error

The error of the **first point** in our dataset (*i* = 0) can be represented like this:

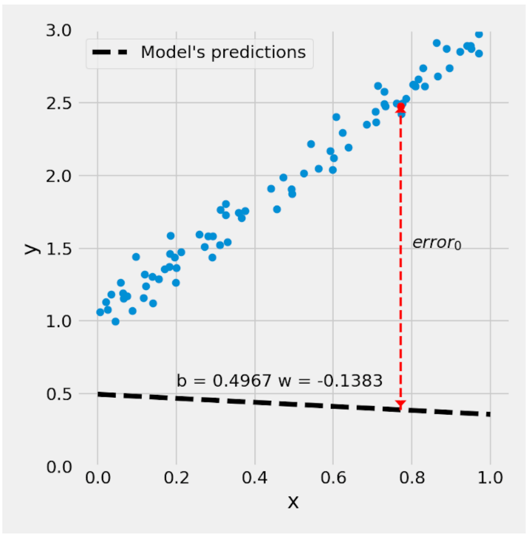


Figure 0.3 - Prediction error (for one data point)

The **loss**, on the other hand, is some sort of **aggregation of errors for a set of data points**.

It seems rather obvious to compute the loss for **all** (*N*) data points, right? Well, yes and no. Although it will surely yield a **more stable path** from the initial **random parameters** to the parameters that **minimize the loss**, it will also surely be **slow**.

This means one *needs to sacrifice (a bit of) stability for the sake of speed*. This is easily accomplished by randomly choosing (*without replacement*) a subset of ***n*** out of ***N*** data points each time we compute the loss.

Batch, **Mini-batch**, *and* **Stochastic Gradient Descent**

- If we use **all points** in the training set (*n* = **N**) to compute the loss, we are performing a **batch** gradient descent;
- If we were to use a **single point** (*n* = **1**) each time, it would be a **stochastic** gradient descent;
- Anything else (*n*) **in between 1 and** ***N*** characterizes a **mini-batch** gradient descent;

For a regression problem, the **loss** is given by the **mean squared error (MSE)**; that is, the average of all squared errors; that is, the average of all squared differences between **labels** (*y*) and **predictions** (*b* + *wx*).

$$
\begin{aligned}
\text{MSE} &= \frac{1}{n}\sum_{i=1}^{n} \text{error}_i{}^2 \\
&= \frac{1}{n}\sum_{i=1}^{n} (\hat{y}_i - y_i)^2 \\
&= \frac{1}{n}\sum_{i=1}^{n} (b + wx_i - y_i)^2
\end{aligned}
$$

Equation 0.3 - Loss: mean squared error (MSE)

In the code below, we are using **all data points** of the training set to compute the **loss**, so ***n* = *N* = 80**, meaning we are indeed performing **batch gradient descent**.

Step 2

```
# Step 2 - Computing the loss
# We are using ALL data points, so this is BATCH gradient
# descent. How wrong is our model? That's the error!
error = (yhat - y_train)

# It is a regression, so it computes mean squared error (MSE)
loss = (error ** 2).mean()

print(loss)
```

Output

```
2.7421577700550976
```

Loss Surface

We have just computed the **loss** (2.74) corresponding to our **randomly initialized parameters** (*b* = 0.49 and *w* = -0.13). What if we did the same for **ALL** possible values of ***b*** and ***w***? Well, not *all* possible values, but *all combinations of evenly spaced values in a given range*, like:

```
# Reminder:
# true_b = 1
# true_w = 2

# we have to split the ranges in 100 evenly spaced intervals each
b_range = np.linspace(true_b - 3, true_b + 3, 101)
w_range = np.linspace(true_w - 3, true_w + 3, 101)
# meshgrid is a handy function that generates a grid of b and w
# values for all combinations
bs, ws = np.meshgrid(b_range, w_range)
bs.shape, ws.shape
```

Output

```
((101, 101), (101, 101))
```

The result of the `meshgrid()` operation was two (101, 101) matrices representing the values of each parameter inside a grid. What does one of these matrices look like?

```
bs
```

Output

```
array([[-2.  , -1.94, -1.88, ...,  3.88,  3.94,  4.  ],
       [-2.  , -1.94, -1.88, ...,  3.88,  3.94,  4.  ],
       [-2.  , -1.94, -1.88, ...,  3.88,  3.94,  4.  ],
       ...,
       [-2.  , -1.94, -1.88, ...,  3.88,  3.94,  4.  ],
       [-2.  , -1.94, -1.88, ...,  3.88,  3.94,  4.  ],
       [-2.  , -1.94, -1.88, ...,  3.88,  3.94,  4.  ]])
```

Sure, we're somewhat *cheating* here, since we *know* the **true** values of *b* and *w*, so we can choose the **perfect ranges** for the parameters. But it is for educational purposes only :-)

Next, we could use those values to compute the corresponding **predictions**, **errors**, and **losses**. Let's start by taking a **single data point** from the training set and computing the predictions for every combination in our grid:

```
dummy_x = x_train[0]
dummy_yhat = bs + ws * dummy_x
dummy_yhat.shape
```

Output

```
(101, 101)
```

Thanks to its broadcasting capabilities, *Numpy* is able to understand we want to multiply the **same *x* value** by **every entry** in the ***ws* matrix**. This operation resulted in a **grid of predictions** for that **single data point**. Now we need to do this for **every one of our 80 data points** in the training set.

We can use *Numpy*'s `apply_along_axis()` to accomplish this:

Look ma, no loops!

```
all_predictions = np.apply_along_axis(
    func1d=lambda x: bs + ws * x,
    axis=1,
    arr=x_train,
)
all_predictions.shape
```

Output

```
(80, 101, 101)
```

Cool! We got **80 matrices** of shape (101, 101), **one matrix for each data point**, each matrix containing a **grid of predictions**.

The **errors** are the difference between the predictions and the labels, but we cannot perform this operation right away—we need to work a bit on our **labels (*y*)**, so they have the proper **shape** for it (broadcasting is good, but not *that* good):

```
all_labels = y_train.reshape(-1, 1, 1)
all_labels.shape
```

Output

```
(80, 1, 1)
```

Our **labels** turned out to be **80 matrices of shape (1, 1)**—the most boring kind of matrix—but that is enough for broadcasting to work its magic. We can compute the **errors** now:

```
all_errors = (all_predictions - all_labels)
all_errors.shape
```

Output

```
(80, 101, 101)
```

Each prediction has its own error, so we get **80 matrices** of shape (101, 101), again, one matrix for each data point, each matrix containing a **grid of errors**.

The only step missing is to compute the **mean squared error**. First, we take the square of all errors. Then, we **average the squares over all data points**. Since our data points are in the **first dimension**, we use `axis=0` to compute this average:

```
all_losses = (all_errors ** 2).mean(axis=0)
all_losses.shape
```

Output

```
(101, 101)
```

The result is a **grid of losses**, a matrix of shape (101, 101), **each loss** corresponding to a **different combination of the parameters *b* and *w***.

These losses are our **loss surface**, which can be visualized in a 3D plot, where the vertical axis (*z*) represents the loss values. If we **connect** the combinations of *b* and *w* that yield the **same loss value**, we'll get an **ellipse**. Then, we can draw this ellipse in the original *b* x *w* plane (in blue, for a loss value of 3). This is, in a nutshell, what a **contour plot** does. From now on, we'll always use the contour plot, instead of the corresponding 3D version.

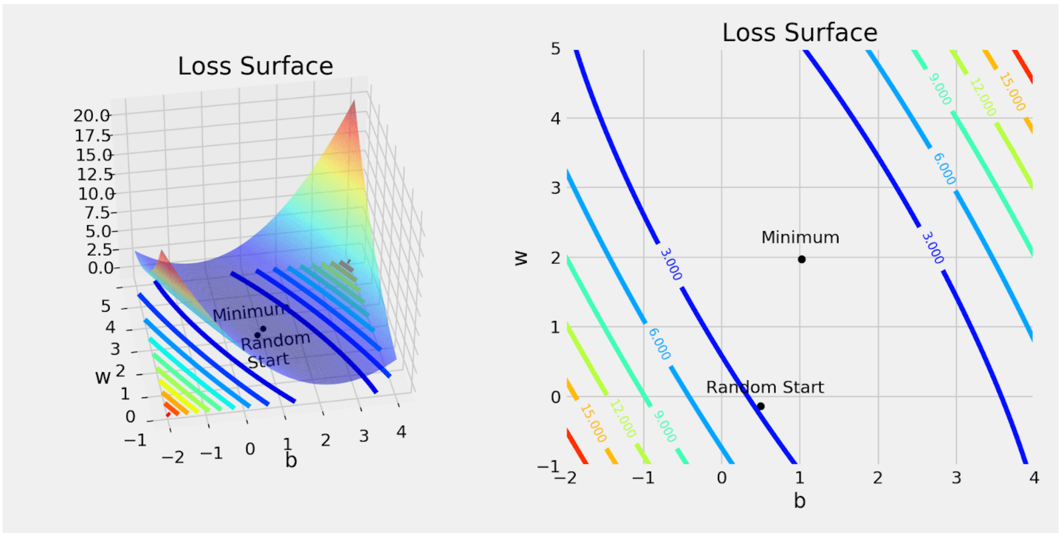


Figure 0.4 - Loss surface

In the center of the plot, where parameters (*b*, *w*) have values close to (1, 2), the loss is at its **minimum** value. This is the point we're trying to reach using gradient descent.

In the bottom, slightly to the left, there is the **random start** point, corresponding to our randomly initialized parameters.

This is one of the nice things about tackling a simple problem like a linear regression with a single feature: We have only **two parameters**, and thus **we can compute and visualize the loss surface**.

Unfortunately, for the absolute majority of problems, **computing the loss surface is not going to be feasible**: we have to rely on gradient descent's ability to reach a point of minimum, even if it starts at some random point.

Cross-Sections

Another nice thing is that we can cut a **cross-section** in the loss surface to check what the **loss** would look like if **the other parameter were held constant**.

Let's start by making ***b* = 0.52** (the value from `b_range` that is closest to our initial random value for *b*, 0.4967). We cut a cross-section *vertically* (the red dashed line) on our loss surface (left plot), and we get the resulting plot on the right:

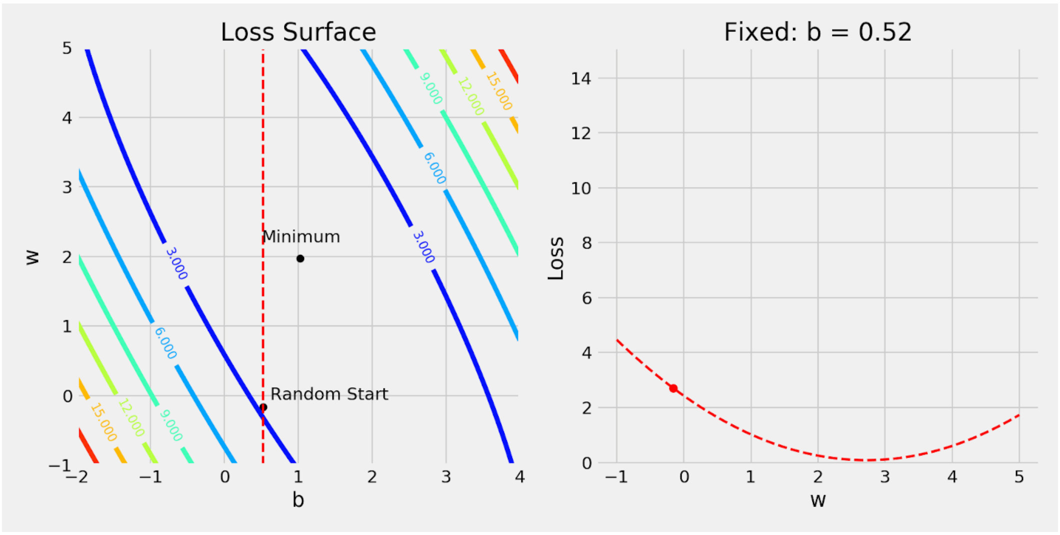


*Figure 0.5 - Vertical cross-section; parameter **b** is fixed*

Read More

Buy complete version: https://leanpub.com/pytorch