–Downgraded-Eduedition–

–Version-0.90–

Contents

Figures

Chapter 1 - Introduction

Why Choose Pytorch As Neural Network And AI Programming Framework?

A lot of authors do begin a programming book on the assumption, that the reader has already decided to learn that language/ framework. I do not. Instead, I think, the proper motivation why to learn specifically the Pytorch framework is the most important part of a book! You, the reader, should know exactly, why you want to learn the Pytorch framework or not. If you do not know well, you will not be fully motivated to spend a significant amount of time in learning. So, you want to know in advance, why to learn Pytorch (and not tensor flow or paddle paddle, instead).

Unfortunately, at the very beginning of a book the benefits of Pytorch can either just fall off the sky or I have to open a huge theory pot, that I cannot give the right factual depth and the proper didactic structure at this early point of time. So, I am presenting the show of benefits in a brief bullet point list, that you have to trust a little.

Pytorch Benefits

- Python has become the industry standard for data science. *Of course, Pytorch has a very polished Python API*, but there are also APIs for C++, Java, Rust, Go, Julia, MATLAB and JavaScript. Additionally, there are native interfaces from C++ and Lua to the original base framework Torch.
- The Pytorch framework is actively supported by most big cloud computing providers through pre-build container environments or container build templates, such as Google Cloud, Amazon AWS, Microsoft Azure, Lightning Studios, Scaleway (France), Open Telekom Cloud (Germany), NTT Cloud Europe (UK), …
- However, currently still “tensor flow” is the most popular AI framework, Pytorch is the second most popular one. But a lot of the “tensor flow” popularity is derived from programming languages other than Python, the data science industry standard. Further, the relative popularity of Pytorch is - in contrast to the one “tensor flow” - growing every year. So, maybe already in 2 or 3 years Pytorch might be as popular or even more popular than “tensor flow”.
- Pytorch is much more intuitive than “tensor flow”. In “tensor flow” neural networks are represented by *connection graphs*; this makes a tensor flow model much more technical than a directly interpretable Pytorch model. The first stable version of “tensor flow” was published (by Google in 2015) round about 2 years early than the first stable of Pytorch (by Facebook in 2017). Thus, the Pytorch developers could intensively learn from the experiences with the “tensor flow” and the “paddle paddle” framework; they have been able to include a lot of usability improvements into Pytorch.
- The Pytorch developers have realized a great deal between easy handling of input variables, target variables, train, test and validation data sets but at the same time allowing for fast execution of big data in portioned batches.
- Additionally Pytorch can be executed either on CPU or on NPU/ GPU, once the proper

driver framework of the hardware is installed, such as cuda and AMD ROCm.

- Pytorch is absolutely “pythonic”. Pytorch tensors data arrays, the data structures in Pytorch, in coding get accessed by about 80% like Numpy data arrays. The indexing (the subsetting) and often even the names of the functions/ methods are identical. Last but not least, Numpy data arrays can be seamless referenced as a ptorch tensor array without copying them.
- In summary Pytorch is much more than just an AI framework. It also is a universal big computing framework. E.g to run a very large matrix multiplications of your individual need on a high performance cluster, that can be prototyped on a simple Linux notebook.

Further Study Advices

Research the author’s feature drops on the internet and validate them. Find out the strengths of the other two AI programming frameworks and try to figure out **what features exactly** do **you** need more than Pytorch offers. So that you do find the best neural network and AI framework solution for your individual needs.

Didactic Concept

First of all, I want to emphasize, that *the design of this compact lecture does follow a refined didactic concept*, while most - currently popular - quantitative programming *cookbooks* just present some of the projects an experienced quantitative programmer has in his/ her pocket, mostly randomly filled up with some syntax theory, just where the code element appears in the code. On the one hand cookbooks can provide larger code syntax examples to *experienced programmers*, who significantly know about the underlying theoretical concepts, already. quantitative programming on the other hand, consists out of two parts: The code syntax and the quantitative mathematical operations taking place in background. These quantitative mathematical operations mostly do not get learned by code syntax reproduction, only. Instead the quantitative background concept at least needs to be explicitly mentioned. Further, mathematical operations do massively build theoretically up. E.g one cannot understand derivations without knowing about mathematical powers.

So, when test-reading a quantitative programming *cookbook* in a bookstore or on Amazon for 5 minutes, typically a giant spoiler effect takes place: Hey, look here, the quantitative programming of xyz is so easy! Buy this book! But when effectively reading the cookbook later on at home, most beginners do already get stuck in chapter 2, because he/ she typically does not have all the quantitative background *and the cookbook does not provide a didactic learn structure to achieve one.*

Major Didactic Elements Of The compact lecture

- The programming theory of Pytorch is presented building straight bottom up. Basic things are presented at the beginning. Complex things are presented in the order they do theoretically build up in later chapters. In-advance mentioning of later subjects is avoided.
- “Eating” the elephant in small slices. In-depth topics are intentionally postponed to later

chapters, e.g. neural network hyperparameter choices.

- The compact lecture always concentrates on the important and popular programming constructs. It knits a red thread through the subject Pytorch, that is closely followed without many excursions.
- Simple running code example, that has been specifically designed for the compact lecture. Instead to spoiler his own programming skills, the author tries to show the central Pytorch techniques as easy as possible. The running code example accompanies all the chapters, that are central for understanding the Pytorch learning technology. (“Pytorch learn process”, “Neural Networks”, “Neural Network Hyperparameters” and “Learn Problems And Heuristic Optimization Choices”). It starts from scratch and iteratively gets more and more refined (and automated) throughout the reading.
- The running code example also works on CPUs, even on small Notebook CPUs. So, you can use your Notebook as learn environment. You can decide over paid NPU/ GPU cloud computing options later, when you already know some Pytorch.
- While other programming books have a prosa to code relation of about 10 : 1, the prosa to code relation of this compact lecture is just about 1.5 : 1. The explaining prosa is on purpose kept short onto the point, in order to make consumption easy.
- Further, the explaining prosa is often provided in bullet point list form, where appropriate, also to make consumption easy.
- The compact lecture covers the Pytorch syntax and the Pytorch handling up to intermediate level. After taking the lecture of the compact lecture you should stand right in the Pytorch starting block, from which you should able to study and understand Pytorch projects from Github on your own without requiring a Pytorch cookbook any more.
- Additionally, it also addresses a lot of practical neural network theory questions on the fly, that an AI programmer faces in his daily work. When you are later on going to study Pytorch projects on github, you will discover, that a significant portion of them is already technically depreciated and another significant portion of them does not look quantitatively well designed, already. With the little practitioner’s neural network theory toolbox from this compact lecture, you are getting some ideas, how to make them more effective.

You cannot even imagine, how many rearrangements of the code syntax presentation, the quantitative theory presentation and the developing running code example it took, to bring it all into line on one red thread through the subject Pytorch! But I have written the compact lecture to teach Pytorch to interested people in a qualitative but compact form, so that the lecture - hopefully - does fit into modern strict time management regimes. Not for money in the first or in the second place. I have mainly done it, because I consider Pytorch to be the best neural network and AI framework, currently.

Unfortunately, the manufacturing process of the book does not allow for footnotes and references. Footnotes are not possible in e-books. References cannot be handled consistently by markdown.

The Author's Qualification

The main author has in total 15+ years experience in quantitative coding and quantitative modeling. Currently, he has 8+ years coding experience in quantitative Python (Numpy, Pandas, SciPy, Sklearn, SymPy) and 3+ years experience in Pytorch. By the way, the main author is also the inventor of FGN-distribution/ S-distribution, that has been published (in the same github repository) as an installable Python package. The new distribution was mainly found by programming a lot of simulations in quantitative Python.

The lecture does also include numerous improvements and enhancements from further authors and contributors. A big thank you for all these helpful valuable contributions to everyone, who has spend a significant portion of time onto this compact lecture. Especially, I am also very thankful for all those contributions, which could not make it onto the final compact lecture. Unfortunately, a compact lecture is - and must be - compact.

The Proceeding Of The Compact Lecture

- Chapter 2 "Pytorch Tensor" contains the syntax part. Especially the syntax regarding to the Pytorch tensor, the central data structure of Pytorch.
- Chapter 3 "Pytorch Learn Process" addresses how AI learning is basically accomplished in ptorch. The foundation components - prediction model, loss function, optimization algorithm and training loop - are presented. Therefor, a central study case is introduced and a corresponding running code example is created from scratch to visualize the foundations of the learn process. Finally, the running code example gets updated with the first Pytorch automatics, an autogradient and an optimizer.
- In chapter 4 "Neural Networks" the model of the running code example is switched to a neural network. The chapter mainly covers the creation of a neural networks in Pytorch, the overfitting problem and the activation functions.
- Chapter 5 "Neural Network Hyperparameters" begins with "levels of measurement" and "loss function" theory. Following a "fitness test and learn stop" procedure gets implemented into the running code example. In the second half of the chapter the subject "neural network architectures" is discussed.
- Chapter 6 "Learn Problems And Optimization Choices" is self speaking. First, the most frequent learn problems and their solves are presented. Then, the author introduces an universal heuristic optimization choice approach. Finally, the didactic lecture of the whole big Pytorch learn process and the journey of the accompanying running code example both come to an end; the author provides the final solution of the central study case.
- Chapter 7 - "Prepare Data" discusses loading and proper preparation of data. The second part of the chapter is a about custom Pytorch datasets and lazy loading. The chapter includes a new running code example to show the full development path from data preparation over custom data sets up to a working, switchable prediction model.

Coordinates

Sourcing addresses for the compact lecture

- github.com/quantsareus/Pytorchcompactlecture/eduedition [Downgraded Eduedition]
- [An official print book version is currently work in progress]
- [An official e-book version is currently work in progress]

Download address for the code examples

- github.com/quantsareus/Pytorchcompactlecture/code

Issue tracker

- github.com/quantsareus/Pytorchcompactlecture/issues

Study Tips

- I guess, you already have a Pytorch project in mind. If not, imagine one.
- Take down bullet point notes of the central Pytorch programming elements by hand, you think you will need for your first Pytorch project. Write them down - pretty separately - on pieces of paper, so that you can assemble them in a folder and you can exchange each one separately by a corrected or extended version. Writing things down by hand is a natural memo technique.
- Install Pytorch before starting studying and try to execute those parts of the Pytorch syntax yourself, you do not understand.
- Study and run the code examples and try to write modified versions of some of them.
- The “further study advices” are meant for your individual evaluation, if you do require this information now, or you can skip it for a while. However, you should mark every skipped further study advice as undone and additionally should notice it on a secondary bucket list.
- Refocus what you really want. Do you really want only the consumption life, forever? Only intense social media, intense TV, party, styling, shopping, holiday, …? Of course everyone likes and needs the consumption life to some degree. But at some point, the MAKER voyage gets more interesting, *what you are capable* to analyze and *to actively engineer on your own*. Pytorch programming is not really a rocket science. (The mathematical background needed to invent a new AI method is another thing.) But, if you want to get successful in Pytorch, you have to be willing, that this project for some time is more important than social media and TV. As reward, you will be granted a lot of new wonderful possibilities to perform things, that have not been possible before at all, respectively that only could be accomplished manually, before. And you will earn some money with this skill, of course.
- Turn off the smart phone and all other learn interruptions. If necessary, wear ear plugs or a noise canceling head set while studying.
- Use a positive learn environment, where you can study undisturbed. You will not learn much in a place, where you feel uncomfortable. If you do not have one, maybe you can set up a temporarily one by recycling old, garden or camping furniture. At the end it just counts, if you have been able to do the lecture, or not.

- Ginkgo extract is a natural phytotherapy, that (unlike similar agents) specifically increases the blood circulation in the fine granular blood vessels. Of course it is no learn wonder drug, but it is known to increase the concentration capability (and also the maximum muscle exercise duration), significantly. It is widely used by elderly people with concentration problems. Maybe, it can give the final kick to get into learning. Request medical/ pharmaceutical advice before starting to take it. *Do not combine with other blood thinning agents.*

Pytorch Setup

Linux Setup

On a Linux system there is already a “basic Python install”. The basic Python install provides parts of the OS base functionality. Installing further python packages into the basic Python install is possible, but comes at the risks, that the configuration changes made by the new packages may harm the basic, Python related OS functionality. Thus, it is more safe to leave the administration of the basic Python install fully in control of the OS package manager, e.g. apt or dnf, depending on the Linux distribution.

A Pytorch programmer not only requires Pytorch but also numerous parts of the quantitative Python stack out of Numpy, Pandas, Scipy, Sklearn and a plotting library (e.g. matplotlib).

Finally, there are two pertinent options to do install Pytorch.

- Installing Anaconda as quantitative Python stack. This is the recommended option for Pytorch beginners.
- Creating a virtual Python environment and installing the quantitative Python stack into the virtual environment.

For the following, the Anaconda distribution is assumed as quantitative Python stack.

CPU Computing Setup

1. Install the Anaconda distribution.
2. Install Pytorch by the Anaconda package manager.

```
conda install pytorch torchvision -c pytorch
```

The CPU computing install is fully sufficient to accomplish all the examples of the lecture! You can decide over paid NPU/ GPU cloud computing options later, when you already know some Pytorch.

NPU/ GPU Computing Setup

The Pytorch framework supports accelerated computing on an NPU/ GPU. Then, the computing library, typically provided by the NPU/ GPU manufacturer, has to be installed first. E.g. the Nividia cuda library. Since 2019 Pytorch also supports the AMD ROCm GPU computing library

(https://rocm.github.io). *For the following, the cuda library is assumed as computing library.*

1. Install cuda.

   For Debian family Linuxes
   `sudo apt install cuda`

   For Fedora family Linuxes
   `sudo dnf install cuda`

   For other Linuxes you might need to install from a private package repository. Look it up on the internet.

2. Install the Anaconda distribution.

3. Install Pytorch by the Anaconda package manager.
   `conda install pytorch torchvision -c pytorch`

Windows Setup

The author is aware of two popular strategies to run Pytorch on MS Windows.

1. Install the windows subsystem for linux (WSL) and install Anaconda and Pytorch on the WSL-Linux environment.
2. Install the Docker container environment and e.g. use the miniconda container from continuumio to run Anaconda. The miniconda container requires an additional docker install of Pytorch.

Both strategies do require to learn some Linux. WSL provides - humble speaking - a Linux emulation under Windows. During the last years the windows subsystem for linux has become a pretty stable and professional Linux run environment.

But Linux is also the base operating system inside most Docker containers. There are also some Windows based Docker containers, but these are absolutely exotic (market share < 0.5%). A quantitative Python stack container or even a Pytorch container is only available with Linux OS inside. But a Windows Docker host does not provide a competitive stability and the clean garbage collection as a Linux Docker host. Anyway, on a (Linux) Docker host the container is run as a simple host process; this setup does not reach the stability of a Kubernetes environment, where each container image is run in a separated virtual machine.

Cloud Computing Setup

- Google Colab (colab.reseach.google.com) offers a cloud Pytorch out of the box, however limited to Python notebooks only. After an initial registration Google Colab grants a free access to a computing session up to maximum of 12 hours. The computing power of the session is limited to cloud computing entry level; usually the session has allocated an old NPU/ GPU from the second predecessor generation. There are also upgrade options to paid plans with powerful Google TPUs in place, but the limitation to python notebooks remains. Nevertheless, for Windows and Mac OS users Google Colab keeps one of the

easierst access possibilities to Pytorch.

- Paid, professional, usually platform specific cloud computing solutions, that support Pytorch by pre-build container environments or container build templates, are - among others - supplied by Google Cloud, Amazon AWS, Microsoft Azure, Lightning Studios, Scaleway (France), Open Telekom Cloud (Germany), NTT Cloud Europe (UK).

A detailed coverage of cloud computing is far out of the scope of a compact lecture.

Chapter 2 - Pytorch Tensors

- Tensors are the central data object structure in Pytorch.
- The major content of a tensor is a static typed data array.
- The data of a Pytorch tensor array is all of the same data type, either integer or float, and also all of the same memory size. (Character arrays are not possible.)
- The central data array of a tensor can be easily converted to a Numpy array. Also a Numpy array can be easily converted to a tensor.
- But a tensor object provides more than the central data array.
  - A tensor object also includes methods for indexing and accessing partial data portions.
  - A tensor object also provides methods for typical operations, e.g sum() and mean().
  - A typical tensor also includes an automatic gradient, so that derivative calculations can be computed fast.
- When referring to a tensor either the major data array can be addressed or the whole tensor object, depending on the context.
- Last but not least tensors also do allow for execution on an NPU/ GPU instead of execution on the CPU, when the corresponding framework is installed, e.g CUDA from Nvidia.

One-Dimensional Tensors

Creation By Tensor Constructor

```
import torch
a= torch.tensor([1.0, 2.0, 3.0])
print( a)
```

Creation By Construction Method

```
b= torch.zeros(3)
c= torch.ones(3)
print( b)
print( c)
```

Indexing (Subsetting)

```
print( a[0])
```

Value Modification

```
b[0]= 3.0
b[1]= 2.0
b[2]= 1.0

print( b)
```

List Of Construction Methods

The tensor construction methods have all to be called following the form `torch.method(...)`

The common Pytorch tensor construction methods are

- tensor(…)
- empty(…); creates an empty tensor containing NULL values
- zeros(…); creates a tensor out of zeros
- ones(…); creates a tensor out of ones
- zeros_like(tensor); zeros in the dimensions of tensor
- ones_like(tensor); ones in the dimensions of tensor
- eye(…); Identity matrix
- arange(start, end, step; A range from start to end with increment of step, default_step = 1
- linspace(start, end, steps= N); A linear range from start to end in N steps
- logpace(start, end, steps= N); A logarithmic range from start to end in N steps
- rand(…); uniform distributed random numbers
- randn(…); normal distributed random numbers

Multi-Dimensional Tensors

Creation By Tensor Constructor

```
import torch
a= torch.tensor([[1.0, 2.0], [3.0, 4.0], [5.0, 6.0]])
print( a)
```

Creation By Construction Method

```
b= torch.zeros(3, 2)
c= torch.ones(3, 2)

print( b)
print( c)
```

Indexing (Subsetting)

```
print( a[0, 1])
```

Value Modification

```
b[0, 0]= 3.0
```

```
b[0, 1]= 2.0
b[0, 2]= 1.0

print( b)
```

Query The Dimensions

```
print( a.shape)
print( b.shape)
```

Slice Indexing

Slice Indexing Of A Python List (Review)

```
alist= list(range(6))
print( alist)
```

```
- alist[:] # All elements
- alist[1:] # From element 1 inclusive up to the end of the list
- alist[:4] # From the begin of the list up to element 4 exclusive
- alist[1:4:2] # From element 1 inclusive to element 4 exclusive in steps of 2
- alist[:-1] # From the begin of list up to the second last element
```

Slice Indexing Of A Tensor

The slice indexing of a Pytorch tensor is just like the slice indexing of a Nummpy array.

```
a= torch.tensor([ [1.0, 2.0, 3.0], [4.0, 5.0, 6.0], [7.0, 8.0, 9.0] ])

- a[1:] # All rows beginning with row 1 and implicitly all columns
- a[1:, :] # All rows beginning with row 1 and explicitly all columns
- a[1:, 0] # All rows beginning with row 1; first column only
- a[None] # Empty tensor container object with the dimensions of tensor a
```

Broadcasting

“Broadcasting” is a technical Python term. When two vector like operands *of different size* are processed by an operator (e.g +, -, *, /), the values of the smaller operand get repeated, so that the operation can take place, although the two operands have different sizes.

The method `rand()` creates uniform distributed values within the interval [0; 1]. If you require uniform random values within the interval [0; 2], you can simply multiply the uniform tensor by 2.

```
a= torch.rand(4, 4)
b= a *2
print( b)
```

How is this code processed in technical detail? First, the scalar value 2 is blown up 4 x 4 times (broadcasting step). Then, every element of a at index position i,j gets multiplied with the index corresponding element of the broadcasted version of 2.

Broadcasting does also work on smaller non-scalar tensors, if the dimension sizes of the smaller tensor all are a non-decimal dividers of the dimension sizes of the larger tensor. (Otherwise, an error of non-matching dimension sizes will be thrown.)

```
c= torch.tensor([1.0, 2.0])
d= c * a
print( d)
```

Automatic broadcasting may feel a little bit strange at first, but it is an established Python technique, without which quantitative Python code would be much longer.

Tip
Do use scalar broadcasting, only. Generally, the broadcasting of smaller arrays is error prone and is not well understood by Non-Python developers!

Data Types

A lot of standard Python objects are dynamically typed. Meaning the data type (and the dimensions) of the data corresponding to a variable gets determined on run time. Where needed, automatic type casts get applied. So, a Python programmer does not have to think much about data types.

Unfortunately, dynamic data types do compute slowly. Thus, Pytorch tensors have static data types, so that they can get computed fast. (Just like the static Numpy arrays.) So, all data elements of a Pytorch tensor have to be of the same data type.

The torch attribute to specify the data type of a tensor is `dtype`.

The Data Type Attribute

```
a= torch.tensor([ [1, 2], [3, 4] ], dtype= torch.float16)
b= torch.ones( (10, 2), dtype= torch.float16)
```

There are also torch data type synonyms, that can be used equivalently. E.g `dtype= torch.half` instead of `dtype= torch.float16`.

List Of Data Types

Data Type	**Synonym**
float64	double
float32	float
float16	half
uint8	
int8	
int16	short
int32	int
int64	long

Data Type	Synonym
bool	

The default data type, that will be created when nothing is specified, is “float32”.

Single character, string and binary blob data types are not possible in Pytorch tensors.

Query The Data Type

```
print( a.dtype)
```

Remember

- Calling a tensor attribute, such as shape or dtype, follows the form `tensor_name.attribute`.
- Calling a tensor class method follows the form `tensor_name.method()`.

Cast The Data Type Of A Tensor

```
c= a.to( torch.double)
d= b.short()
```

The returns of a function can get casted, directly.

```
e= torch.zeros(10, 2).double()
f= torch.zeros(10, 2).to(dtype= torch.short)
```

`.to()` only performs a data type conversion, if necessary.

When mixed (numeric) data types get assigned to one tensor, the tensor will be automatically converted to the largest data type of the mixed set.

Proper Use Of Data Types

When computing on the CPU usually “float32” is used, the default data type. “float64” does not significantly improve the quality of a neural network, but it requires much more computing time. “float16” is not actively supported by modern CPUs any more.

When computing on a NPU/ GPU, however, “float16” is supported. And usually it does achieve significantly faster computing speed. (At some loss in accuracy.)

When a tensor is used as an index for another tensor, Pytorch expects the index tensor to be of type “int64”.

Conclusions

- In a CPU work load “float32” and “int64” will be the dominant data types.
- In an NPU/ GPU work load “float16” and “int64” will be the dominant data types, except there are special precision constrains.

The definition of CPU and NPU/ GPU work loads will be explained later in section “Device Attribute”.

Tensor Operations (Tensor API)

The vast majority of operations on and between tensors, the tensor operations, are available in the torch module.

They can either be called from torch module, directly.

```
a_t= torch.transpose(a, 0, 1)
```

They can either be called as a method of the tensor instance.

```
a_t= a.transpose(0, 1)
```

Most of the tensor operations of the torch module are organized exhaustively into the following groups:

- Creation Operations. E.g zeros(), ones().
- Indexing, Slicing, Joining and Modification Operations. E.g transpose().
- Math Operations
  - Pointwise Operations - Mathemaical functions that are applied separately to each element of the tensor. E.g abs() and sin().
  - Reduction Operations - Functions to aggregate the values by iterating through the tensor. E.g sum(), mean(), std(), var().
  - Comparison Operations, e.g equal() and max(). For each comparison operator sign exists a named function.
  - Spectral Operations - Functions for operating and transforming in the frequency domain. E.g stft(), hamming_window()
  - Other Operations - Special function operators on vectors and matrices. E.g cross(), trace().
  - BLAS and LAPACK Operations - Basic linear algebra functions for scalar, vector-vector, matrix-vector and matrix-matrix operations.
  - Random Sampling - Functions for generating random values following a certain distribution. E.g rand(), randn().
- Serialization - Functions for loading and saving a tensor. E.g load(), save()
- Parallelizm - Functions to control parallel execution. E.g set_num_threads().

In the following, the author will present an overview of the tensor operation methods by topics. A detailed introduction of every method will be too boring and out of the scope of a lean compact lecture.

But, it should be pretty helpful to get to know the names of the operation methods there are in Pytorch. This way, a hot candidate for a desired functionality can be searched in the manual or on the internet. Knowing the name of the function candidate already will save time. A lot of operation method names are self explaining, anyway. To others, the author may have added a short “# comment” about their functionality.

Concatenation And Modification Methods

- cat()
- stack()

- gather()
- view()
- squeeze()
- unsqueeze()
- transpose()
- reshape()
- sort()
- permute()

Select And Split Methods

- chunk()
- split()
- select()
- index_select()
- take()

Reduction Methods (Aggregation Methods)

- sum()
- prod()
- cumsum()
- cumprod()
- mean()
- median() # 50% quantile
- mode() # maximum frequency
- std() # standard deviation
- var() # variance
- dist() # returns the p-norm of a distribution

Pointwise Operations (Elementwise Operations)

- Mathematical
  - abs() # absolute values
  - sign() # signature of a value
  - mul()
  - divide()
  - remainder()
  - reciprocal()
  - exp()
  - log()
  - pow()
  - sqrt()
- Trigonometric
  - cos(), cosh()
  - sin(), sinh()

    - tan(), tanh()
    - acos(), asin(), atan()
- Rounding
    - round() # standard round
    - floor() # round down
    - ceil() # round up
    - trunc() # cut of the decimals
- Special
    - clamp() # shortcut for min() and max()
    - sigmoid # sigmoid function
    - erf() # Gaussian error function
    - erfinv() # Inverse Gaussian error function

Matrix Multiplication

- @ # Generic matrix multiplication. This is a relatively new Pytorch feature.
- dot() # Vector multiplication
- mv() # Matrix vector multiplication
- addmv() # Matrix vector multiplication plus adding the result to the input matrix
- mm() # Matrix multiplication
- addmm() # Matrix multiplication plus adding the result to the input matrix

Comparison Operations - Elementwise

- eq() # equal
- gt() # greater
- ge() # greater or equal
- lt() # lower
- le() # lower or equal
- ne() # not equal

Comparison Operations - Non-Elementwise

- equal() # Tensor comparison
- max() # Maximum of
- min() # Minimum of
- topk() # Top k values along a dimension
- kthvalue() # Tuple of k smallest values along a dimension

Reference Names, Output Objects And Side Effects

As typical in Python for any complex data object *a second assignment of an already named tensor* to another variable name by default just creates a second name reference for the data values (“a second pointer”). This is a standard Python practice to save memory allocation time. However, this can cause side effects, when the data values, that do exist only once, get altered under the second reference name.

```
a= torch.ones(2, 2)
b= a
b[0, 1]= 100
print( a)
```

In order to avoid side effects, you have to create a copy of the tensor using `clone()`, so that *the data values are created in memory a second time*.

```
a= torch.ones(2, 2)
b= a.clone()
b[0, 1]= 100
print( a)
```

Fortunately, most operations functions called on a tensor do return a new output object. Then, like in copy by `clone()` above, assigning a variable to the output is assigning the first reference name to a new independent data instance.

Unfortunately, there are exceptions to that general rule. The tensor modification methods

- *transpose()*
- *reshape()*
- *sort()*
- *permute()*

do *not* create a new output object, *but return the original data values*. **Take care for side effects, when performing value modifications on the outputs from these methods.**

In-Place Operations

Most operation methods called on a tensor do return a new output object. But there are situations, where we on purpose want modify the original data. E.g in the beginning of a loop we want to reset the tensor to zeros. This is performed by an in-place operation method.

- In-place operation methods can be used to modify the original data of the input tensor, directly.
- The names of in-place operation methods all contain a trailing underscore. E.g zero_() instead of zero().
- In-place methods can only be called as instance methods.

```
a= torch.ones(3, 2)
a.zeros_()
print( a)
```

Dispatching

As we have seen before, tensor data can be an array of 1 to n dimensions. And each dimension can be of different size. Obviously, the tensor operations from the tensor API have to be looped somehow through the individual shape of the data array. [In Numpy wording the looping process through the data is called vectorization.] Fortunately, this is usually all performed on automatic. The only thing we have to remember here so far is, that looping through the data is called “dispatching”. Just in case once upon a time in future a “dispatching error” may occur on executing our program.

However, there also do exist specialized tensors, e.g the quantitized tensors, where the dispatching needs to be specified by the programmer.

Save And Load A Tensor

The easierst way to save a tensor on the hard disk is the following.

```
torch.save( points, './data/points.t')
```

However, when we want to perform a more sophisticated save, e.g for writing parts of a tensor only, we have to open a file descriptor.

```
with open( './data/points.t', 'wb') as f:
    torch.save( points, f)
```

Loading a tensor

```
points= torch.load( './data/points.t')
```

Saving and loading files is also called “serialization” for technical reasons. Thus, you do find `load()` and `save()` in the torch module serialization.

Device Attribute

Tensor Device Attribute

So far, we have assumed, that a tensor is executed on the CPU. This is the default device, when nothing is specified. However, Pytorch tensors can also get executed on an NPU/ GPU. This is one of Pytorch’s major benefits.

If your python/ conda setup does include the package cuda, a tensor can also get executed on an NPU/ GPU using the device attribute. We can check for an installed cuda package by the following request.

```
if torch.cuda.is_available:
    ...
```

Creating a tensor on a cuda NPU/ GPU.

```
points_gpu= torch.tensor([ [1.0, 2.0], [3.0, 4.0] ], device='cuda')
```

It is also possible to convert a CPU tensor to an NPU/ GPU tensor using the `.to()` cast method.

```
points_gpu= points.to(device= 'cuda')
```

If your machine has more than one NPU/ GPU, you can also decide on which processing unit the tensor should be created.

```
points_gpu= points.to(device= 'cuda:0')
```

By doing so, the following will happen.

1. Sending the data of points to the NPU/ GPU.
2. Create a new tensor in the RAM of the NPU/ GPU.
3. A handle to the new NPU/ GPU tensor is returned.

All ongoing operations on an NPU/ GPU tensor are performed on the NPU/ GPU. The CPU does

not know anything about an NPU/ GPU tensor, until an output to another CPU tensor is created.

```
points_cpu= points_gpu.to(device= 'cpu')
```

The device attribute and the data type attribute of a tensor can be changed in one go.

General Device Attribute

There is also a general device attribute, that alters the default device. Once it has been set, every new tensor, every new function and every new model created from now on will be instanced on the new default device, unless explicitly specified otherwise. But already existing tensors and models, that have been created the modification of the default device, have to swapped to the new device by an explicit `to(...)` cast.

```
if torch.cuda.is_available():
    device= torch.device("cuda")
else
    device= torch.device("cpu")

# Swapping a former CPU model onto the cuda device
model.to(device)
```

Interoperability With Numpy

A Pytorch tensor array can be converted seamless to a Numpy array using `numpy()`.

```
points= torch.ones(3, 2)
points_np= points.numpy()
```

But there are internal technical differences dependent on the device attribute of the tensor. When executing the tensor on the CPU, the Numpy array is created as a new reference name to the old data instance, only. This makes the interoperability with Numpy very fast, but also error prone for side effects.

In case of processing the tensor on the NPU/ GPU, however, the Numpy array is created as new data instance in the NPU/ GPU RAM.

Vice versa a Numpy array can also be converted seamless to a Pytorch tensor array using `from_numpy()`.

```
import numpy as np

points_np2= np.ones(3, 2)
points2= torch.from_numpy(points_np2)
```

The new tensor “points2” will also use the same memory-sharing as above by assigning a second reference name to the old original data, only, when run on the CPU.

Further, “points2” will be different to “points” this way, that the data type will be float64 (not float32). In Numpy the default data type is np.float64, which does remain.

Learn Progress Feedback

By now, you have already accomplished approximately 1/4 of the Pytorch fundamentals. When been familiar with Numpy already, some parts of the chapter may have been a piece a cake, have they? But effectively it does not count anything, how much it took you to get here. The point is, you got here!

This is maybe a good moment to take a break. But carry on, right the next time. *The more soon you do finish the lecture, the more probable you will truly become a Pytorch programmer!*

Further, also you are very welcome to report any error or improvement suggestion regarding this chapter or the one before. The address is: github.com/quantsareus/Pytorchcompactlecture/issues. Where have you got stuck and why? In case, how can this section get explained better? All the reports will be collected in the central issue database, be thoroughly reviewed and - if possible - the issue will be fully solved or mitigated in the next edition of the book.

Especially I want to know, if you would like to have an additional chapter “Special Tensor Topics” as next chapter, that would contain the following. - Memory organization and contiguity. This is either about a deeper understanding of memory storage in Pytorch. And it is also very helpful when performing transposes. - Named tensors. Tensors indexing can also be performed by named tensor dimensions (“named columns”). This is more intuitive, but the functionality is currently still considered experimental by the Pytorch developers. Further, the handling of named tensors is currently (still?) slightly different from named Numpy arrays, which have been in many aspects the predecessor for Pytorch tensor arrays.

Chapter 3 - The Pytorch Learn Process

In this chapter you will learn about the Pytorch learn process. Armed with the tensor theory from chapter 2 we gonna start to program our first Pytorch learn process, now. For this, we need some data. Unfortunately, data loading and data preparation are more advanced topics, that - depending on the prediction model - also do require some quantitative theory background. But we want to get the Pytorch airplane fly smooth and elegant without unnecessary bumps. So, a very simple running study case is introduced, that will accompany the lecture of the Pytorch learn process throughout the next chapters.

Introduction Of The Celsius To Fahrenheit Case

Professor Roentgen from Sweden has forgotten the formula for the conversion from European degrees Celsius to Anglo-Amercian degrees Fahrenheit. The whole internet in all of Scandinavia is down; all the internet data cables through the sea have been capped by the Russian ghost tanker fleet pulling the ship anchors over the sea ground. Fortunately, Professor Roentgen has two thermometers in his lab, one Celsius and one Fahrenheit thermometer. Using the two thermometers he has taken a measurement series. Now, it is up to you, Professor Roentgen’s young promising assistant (and prospective Pytorch engineer, of course), to find the best approximation formula for the conversion from Celsisus to Fahrenheit based on the measurement

data. So that the research can keep going by using the approximation formula, until the internet cables will be repaired in about 12 months.

Measurement	Celsius	Fahrenheit
1	-4.0	21.8
2	0.5	35.7
3	3.0	33.9
4	6.0	48.4
5	8.0	48.9
6	11.0	56.3
7	13.0	60.4
8	14.0	55.9
9	15.0	58.2
10	21.0	68.4
11	28.0	81.9

Fig. 1: Tab. Celsius Fahrenheit Measurements

A Pytorch learn process is based on three main components.

1. The prediction model, typically just referred as model.
2. The loss function, that evaluates the deviations (typically the differences) between the predicted values and the true values. In some domains the loss function is also called cost function or objective function to minimize.
3. The training process, that iteratively minimizes the loss function by parameter updates. Typically, the training process consists out of
    1. An optimization algorithm or optimization operator (especially the gradient). The gradient is also the main ingredient of sophisticated neural network optimization algorithms.
    2. A training loop, that includes
        - the start values of the parameters.
        - the calculations to execute in each iteration step.
        - the parameter update procedure.
        - the loss tensor update (the recalculation of the loss with the updated parameters).
4. Run the training loop.

Now, let us have a look at our first Pytorch learn process code.

As the book is designed as a (hopefully) well structured, compact lecture (and not as a somehow together muddled cookbook) we do - for didactic reasons - on purpose start from scratch. The example case will deliver us a unique, hands-on insight into the Pytorch learn process. While working through the chapter the running example will of course get much more refined and also much more automated, as well.

The prediction model of our first Pytorch code example is a simple univariat model. Instead of the one-step OLS calculation of the parameters b0 und b1 in a regression, we do optimize the two parameters iteratively by the Pytorch gradient.

Code

Fig. 2: Cod. linmodel01.py

Pytorch Autogradient

As the gradient is very common used for the training/ the optimization of neural networks (and also in optimization algorithms in general), the Pytorch framework includes an autograd method, that tracks and computes the derivatives of the tensor with respect its predecessor tensor chain. This is really nice for us, as manually calculating the gradient sometimes may be pretty challenging and error prone.

Therefor, all Pytorch tensors have an attribute grad. The autogradient functionality is activated by the parameter `requires_grad= True`. E.g.

```
params= torch.tensor( [0.0, 1.0], requires_grad= True)
```

Typically, the loss function tensor is the starting point of the gradient chain. The starting point of the so called “backpropagation of errors” is set by

```
loss.backward()
```

When the params gradient attribute and loss.backward() are set, Pytorch will start to calculate the gradients of the whole tensor chain between the params tensor and the loss tensor (as far their functions are differentiable). In background an “automatic differentiation” process takes place for the whole tensor chain. The following code example contains a simple autogradient chain.

```
Code
```

Fig. 3: Cod. linmodel02.py

Control The Autogradient

In any tensor without an autogradient the attribute tensorname.grad is “None”.

The autogradient attribute can be queried as follows.

```
if params.grad is None:
    ...
```

When modifying/ updating the tensor data array values, the autogradient calculation should be disabled using

```
with torch.no_grad():
    ...
```

When modifying/ updating the tensor data array, we usually also require to (re)initialize the autogradient with zeros. This is performed by the in-place method

```
params.grad.zero_()
```

So, we can use the autograd feature in our simple univariate linear model, now.

```
Code
```

Fig. 4: Cod. linmodel03.py

Further Study Advices

- Depending on the individually desired quantitative programming project it might be necessary to learn about “automatic differentiation” or not.

The Pytorch Optim Module

The Pytorch module optim offers us mainly two things.

- A more automatic handling of the training process.
- Full rank optimization algorithms, much more sophisticated than the simple gradient.

The import is

```
import torch.optim as optim
```

The following statement lists up the available packages and optimization algorithms.

```
dir(optim)
```

Optimizers (Optimization Algorithms)

An optimization algorithm from the optim module can be setup using the form

```
optimizer= optim.algorithm( [parameters], lr= learn_rate)
```

Important optimization algorithms of the optim module

- Adam()
- Adagrad()
- SGD() # stochastic gradient descent
- RMSprop()
- Adadelta()

Remember

- Every optimization algorithm has some “individual base learn rate”, that either can be increased to more aggressive and fast learning or can be decreased to more safe and slow learning. Unfortunately, each base learn rate depends on the data and on the loss function niveau, thus is usually not known, initially. When the learn rate gets to high, the optimization algorithm will run totally out of the loss valley and will not find it again. If the learn rate gets too low, the learn process runs into the right direction, but does not reach the minimum of the loss function within the maximum iteration limit.
- Thus, each algorithm has an individual learn rate window for each new learn task, out of which the algorithm does reach the minimum of the loss function (or close beyond). But outside this learn rate window, the minimum loss point (or close beyond) will *not* be found.
- To make the task even more challenging, the learn rate window of each algorithm has also different width. The good ones, have a more broad learn rate window.

Once the optimizer has been set up, a (re)initialization with zeros can be accomplished more comfortable.

```
optimizer.zero_grad()
```

Also, the update of the searched parameters in each iteration step can be performed automatically.

```
optimizer.step()
```

Now, we are implementing the training automatics provided by the optim module into our linear model code example. Except for alternative optimization algorithms. These ones can be knit in much more easy in the further following code example.

```
Code
```

Fig. 5: Cod. linmodel04.py

Chapter 4 - Neural Networks

Till now, we have been dealing - for didactic purposes - with our own handmade linear model, only. This way, we have learned the Pytorch learn process elephant in small slices. But now, it is time for our first neural network. We are going to create a (single) linear perceptron network.

Linear perceptron models have two major benefits.

- They do provide very good insight, how a larger neural network works. Usually, any larger neural network contains at least one layer of linear perceptrons, especially in the input layer.
- The bias parameter of a linear perceptron corresponds to the intercept of a linear regression; the weight parameter of linear perceptron corresponds to the regression coefficient of a linear regression. Thus, a trained perceptron network and a regression solution can be compared *directly* and the comparison does allow for a review of the quality of the optimization algorithm, that has been used to train the neural network parameters. (We will perform such optimization algorithm test later on.)

The torch module for neural networks is nn. A single perceptron model is created as follows.

```
import torch.nn as nn
linpercepmodel= nn.linear(1, 1)
```

That is it. In Pytorch creating neural network architectures is much more easy than in “tensor flow”.

The model expects the input tensor and the target tensor as so called “batch tensors” with another batch dimension, that - however - can be of size one. So we have to add a dummy dimension of size to x and y using `unsqueeze`.

```
x= x.unsqueeze(1)
y= y.unsqueeze(1)
```

Some neural network activation functions can only process target values out of the the interval [0; 1]. Although our linear perceptron model does not contain an activation function, we do standardize y already yet a little bit, in order to keep consistency with future neural networks.

```
y= 0.01 *y
```

But when we gonna show the results in the matplot figure, we want to see original Fahrenheit values, So, y is transformed back before plotting. Putting it all together, we get our first neural

network, a single perceptron model.

Last, we now also do knit in the alternative optimization algorithms from the last section.

Code

Fig. 6: Cod. neuralnetwork01.py

Not very surprisingly, the values of linearpercep_model.bias and linearpercep_model.weight do count almost the same, as the parameters of the former handmade linear model divided by 100.

Auto Loss Function

The torch.nn class provides the typical loss functions out of the box. So we can delete our handmade lossfn() and put

```
lossfn= nn.MSELoss()
```

into the call of the training loop, instead.

Code

Fig. 7: Cod. neuralnetwork02.py

Sequential Network

Now, we want to create a usual sequential network. The network should consist out of an input layer, a hidden layer and an output layer.

First, the simple way to define this network.

```
no_neurons= 20
seq_model= nn.Sequential(
    nn.linear(1, no_neurons),
    nn.Tanh()
    nn.linear(no_neurons, 1)
    )
```

The first layer of a neural network is always the input layer. It contains one neuron. The second layer is the hidden layer. It contains 20 neurons. The third layer is the output layer. It contains 1 neuron.

A neural network can have multiple hidden layers. E.g two.

```
no_neurons= 20
seq_model= nn.Sequential(
    nn.linear(1, no_neurons),
    nn.Tanh()
    nn.linear(no_neurons, no_neurons),
    nn.Tanh()
    nn.linear(no_neurons, 1)
    )
```

But in the running code example, we stay with one hidden layer.

When we afterwards want to view the weights and the biases of the linear layers, however, we have to define the layers by an ordered dict. (The activation layer does not have weights nor biases; it is just a transfer function for the - already modified - input values passing through.)
Code

Fig. 8: Cod. neuralnetwork03.py

The Overfitting Problem

When comparing the mean squared error (MSE) of the sequential 3-layer network to the one of the linear single perceptron model, the MSE is much lower, now.

However, when the true Celsius to Fahrenheit conversion formula counts
`F= 32 + 1.8C`
This formula will obviously generates a true straight line graph. When comparing the curved graph of the sequential network to a thought straight line, we get aware, that the sequential network has achieved the lower MSE by also remembering the measurement errors of each measurement in the data series. But we are looking the universal Celsius to Fahrenheit formula to generalize on any data set (with other measurement errors). The undesired surplus remembering of a neural network is called “overfitting”.

Warning
Because of the “overfitting problem” you can never ever train a neural network designated for production like shown above. Instead, you always have to split the total available data into a train, a test and a validation subset and you have to train the neural network on the train data subset, only! The running code example has been designed this way for didactic purpose, only!

The test subset is required to determine the point of time, before a significant, non-generalizing overfitting does start and he learning is stopped. The need to interrupt the learn process of neural networks is the reason, why neural networks are classified a supervised machine learning method. When instead using an unsupervised machine learning method (or an applied statistics prediction model such as regression or GLM), there will not be a “when to properly stop the learn process problem”.

The validation data subset is required to measure the generalizing prediction quality of the final trained neural network.

Save And Load A Trained Model

As we already are on the go to transform our running code example into a professional project, we are also adding a save options for a trained neural network.

Basically, there are two options to save a neural network.

Option 1 - Model without structure changes

```
torch.save(model, "/tmp/model")
model= torch.load("/tmp/model")
```

Option 2 - Model with structure changes

Load a model containing changes in its structure from a dict

```
modelstate_dict= torch.load("/tmp/model0.20139074")
model.load_state_dict(modelstate_dict)
```

Save a model containing changes in its structure to a dict

```
torch.save(model.state_dict(), path)
Code
```

Fig. 9: Cod. neuralnetwork04.py

Load A Trained Model From Torch Hub

The Torch hub allows to load and to share trained Pytorch models. In the following, however, only the model load from Torch hub gets addressed.

The Torch hub consists out of two things.

- Since 2019 there is the Torch.hub API, that provides an interface to load and share trained Torch models. A Torch hub model is effectively published as a standard github repository, that additionally contains a hubconf.py file with some Torch hub definitions.
- Since 2024 there is also a beta version of a central directory/ search engine online at “pytorch.org/hub”, which by the time of writing has indexed about 50 Torch hub models.

However, some more Torch hub models can be found, when searching github directly for ‘hubconf.py’. Once you are enrolling from a hotspot network, for which the github search functionality has been disabled for performance reasons, you can also query indirectly by google/ startpage for ‘hubconf.py site:gihub.com’. Also most other search engines do understand the ‘site:gihub.com’ tag.

Once a repository with a matching model has been found, the model can be loaded by the Torch hub API. Let us assume we want to get the resnet18 model from the master branch of the repository `github.com/pytorch/vision`. Then, the import of the trained model is performed as follows.

```
import torch
from torch import hub

resnet18_model = hub.load('pytorch/vision:master', 'resnet18',
pretrained=True)

...

model= resnet18_model
```

Activation Functions

The torch.nn class offers a lot of activation functions.

List Of Important Pytorch Activation Functions

- Activation functions with limited maximum output value
  - Sigmoid() -> [0; 1]
  - Tanh() -> [-1; 1]
  - Hardsigmoid() -> [0; 1]
  - Hardtanh() -> [-1; 1]
  - Softmax() -> [0; 1]
- Activation functions with unlimited maximum output value
  - ReLU() -> [0; +Inf]
  - LeakyReLU() -> [-Inf; +Inf]
  - Softplus() -> [0; +Inf]

The limited activation functions have insensitivities, when the values of the input variable get very low or very large. This can lead to dying gradients. Dying gradients can be avoided by pre-transforming an input variable this way, that most of the input values will fall down on a higher slope part of the activation function in the first layer.

Sigmoid like and tanh like nactivation functions are considered same effective, in general. With the major difference however, that the tanh (hardtanh) activation can only get loaded approximately symmetrical, initially, if the input values are negative and positive. However, an individual pre-transform of every input variable makes a lot of work. Thus in practice usually either the sigmoid (hardsigmoid) or tanh (hardtanh) activation is chosen, that meets the majority of the input variables' value distributions best.

In our running example the input values of the Celsius measurements are mostly all positive. Thus, we do change to `Sigmoid()` activation function.

The unlimited activation functions have been invented to provide always sensitive activation functions. Coming at the cost to easily create exploding gradients. From the authors personal experience exploding gradients usually can be avoided by combining with a limited activation function in an additional last activation layer (, if there is not such one already).

`Code`

Fig. 10: Cod. neuralnetwork05.py

Further Study Advices

This is a very good point of time you can make your own experiences. But keep in mind, till here you have just been shown the components of a Pytorch neural network and its corresponding training process, but not how to assemble them properly! Sometimes, a lecture simply cannot be maximum compact/ maximum easy to consume and as well maximum well guiding at the same time.

- Is the MSE value of a neural network model computed out of training data a proper prognosis quality measure of a neural network? Why yes or why not?(Hint: A neural network is a learning technique, that has to be supervised.)
- Is the MSE value of a GLM model computed out of training data a proper prognosis

quality measure of a neural network? Why yes or why not? (Hint: A GLM model is an unsupervised learning technique.)

- What is happening to the MSE when transforming the target variable?
- Play a little around with alternative neural network sizes and activation functions, just to get coding expertise. [Some combinations may not properly learn. Do not worry, that is normal.]
- Look up the graphs of the activation function with limited maximum output values on the internet. For each notice the corresponding input values, where the corresponding output can be considered almost insensitive.

Learn Progress Feedback

By now, you have already accomplished approximately 1/2 of the Pytorch fundamentals. By now, you already do know about

- the important tensor syntax elements
- the important syntax elements for creating a neural network

You can to take a break. But carry on, right the next time. View it this way. *The more soon you do finish the lecture, the more probable you will truly become a Pytorch programmer!*

Further, also you are invited to report any error or improvement suggestion regarding this chapter or the one before. The address is: github.com/quantsareus/Pytorchcompactlecture/issues. Where have you got stuck and why? In case, how can this section get explained better? All the reports will be collected in the central issue database, be thoroughly reviewed and - if possible - the issue will be fully solved or mitigated in the next edition of the book.

Chapter 5 - Neural Network Hyperparameters

Initially, an organizational remark. This chapter will get more more quantitative (and brainy), now. We have already learned a lot of Pytorch syntax elements. When deciding how to use them properly, we need to consider the quantitative background. To keep this journey easy and well understandable for a broader audience, we will perform it in prosa only, not mathematical formulas. However, - depending on your personal statistical/ mathematical background - it might be necessary to read the full theory behind the presented quantitative stuff up on the internet to achieve a deep understanding of the matter. (This is - and should remain - a compact lecture.)

From the previous chapter we have learned, how to create a sequential network. This chapter - among other smaller pieces - will mainly address alternative loss functions and the architecture of a neural network prediction model. So, this chapter is about the first two components of the Pytorch learn process.

Pytorch learn process (Recapture)

1. prediction model
2. loss function
3. training process
    1. Optimization algorithm or optimization operator

2. Training loop
4. Run the training loop.

Levels Of Measurement (Review)

Before we can choose the proper loss functions, we need to know about the levels of measurement. The “levels of measurement” (aka “measurement scales of a variable”) is a generally accepted statistics theory, that determines, what kind of mathematical transforms are allowed on which type of variable. It has been developed the American psychologist and applied statistician Stanley S. Stevens in 1946. However, not all transform rules are strictly followed by all data scientist. Meanwhile, also the modern “types of data” philosophy has become very popular, that accounts interval scale and ratio scale variables both as numerical data type and effectively treats them the same.

Levels Of Measurement	**Types of Data**
Binomial/ Multinomial	Categorical/ Qualitative
Ordinal	Rank
Interval	Numerical/ Quantitative (continuous or discrete)
Ratio	Numerical/ Quantitative (continuous or discrete)

Fig. 11: Tab. Levels Of Measurement

Nonetheless, Stevens differentiation of numerical variables is still interesting today. In Stevens’ systematic all numerical data typically have at least interval scale, thus the distance between 1 and 2 counts the same as the one between 2 and 3, and so on for all values. Stevens postulated, that strictly positive variables with an (unreachable) theoretical zero point are primarily interpreted as multiples of one (instead of a sum), so that there is a natural understanding of the ratio of two values, e.g. value2/ value1. He called this variables “ratio scale variables”. Temperatures in Kelvin, equity stock prices and stock supply quantities are typical examples for ratio scale variables. [The absolute minimum temperature of 0 Kelvin cannot be realized anywhere in nature. When a stock price gets zero, the company’s stocks will be delisted from the trade exchange. When your supply of flour packages gets zero, you can also account “None” instead of “0”, because there is no physical supply of flour, any more. You can owe someone a package of flour, but the physical supply cannot become negative.]

Steven’s - even theoretically - strictly positive *ratio scale variables - even in modern times - might be hot candidates for modeling in relative error* instead of standard modeling in absolute error. This is the neoclassical enhancement of Steven’s original theory.

Loss Functions

A “loss function” is a mathematical description, how to evaluate the deviations between the predicted value and the true value. Depending on the context, the field of science and the business sector, a loss function is also called “cost function” or (negative) “objective function.”

Following, the list of the important loss functions is presented. If available in Pytorch, also the Pytorch nn.lossfunction() is provided.

Target Type	**Loss Function**	**Pytorch Class**	**Purpose**
Categorical	Binomial Cross-Entropy	nn.CrossEntropyLoss()	Pytorch standard loss function for binomial target
Categorical	Multinomial Cross-Entropy	nn.CrossEntropyLoss()	Pytorch standard loss function for multinomial target
Categorical	0/ 1 Dummies		Depreciated categorical loss function
Numerical	Squared Error		Standard loss function for numerical target
Numerical	Mean Squared Error	nn.MSELoss()	Squared error divided by #observations
Numerical	L1 Loss (Absolute Value Error)	nn.L1Loss()	For special cases; undifferentiable minimum and bad convergence property
Numerical	Log-Cosh Loss		Proxy for L1 loss with differentiable minimum and improved convergence property around the minimum
Image	Cross-Entropy	nn.CrossEntropyLoss()	Pytorch standard loss function for classification; already includes softmaxx activation
Image	NLL Loss	nn.NLLLoss()	Special image classification; allows to handle imbalanced datasets
Image	NLL Loss 2d	nn.NLLLoss2d()	Pixel-wise classification; mostly for semantic image segmentation

Fig. 12: Tab. List of Loss Functions

Model Categorical Scale Variables

A categorical variable can be represented in data in three ways.

- One category column contains the observed features (outcomes) of a categorical variable. Often, the category information is coded to a number. E.g. the column color contains the following values: “1” (=white), “2” (=yellow), “3” (=red), … .
- Multiple 0/1 dummy columns (aka “hot ones columns”). They contain the value “0” or “1”, but only one columns counts 1 per line. E.g. line 1: color_1= 1, color_2= 0, color_3= 0.
- Multiple logit score columns. They contain some high two logit values, e.g. “10” and “-10”, but only one columns counts 10 per line. E.g. line 1: color_1= 10, color_2= -10, color_3= -10.

Usually, a categorical variable is delivered as a category column for the reason of efficient storage. But most data science models do process a categorical target variable either in dummy coding or in logit scores, only. So, a categorical target variable has to be transformed.

Generally, there is a difference, wether the model should predict 0/1 values or normed probabilities, that sum up to 1 per each categorical variable. When binomial probabilities are desired, usually the sigmoid function is chosen as activation in the last hidden layer; when multinomial probabilities are desired, usually the softmaxx function is chosen as activation in the last hidden layer.

In Pytorch however, a categorical target variable is typically modeled as multiple logit scores and the loss function is chosen as `nn.CrossEntropyLoss()`. `nn.CrossEntropyLoss()` chooses binomial and multinomial cross entropy, automatically. This way, **neither sigmoid activtion function nor softmaxx activation function are required any more**, because they are already included in the cross entropy loss function!

Modeling a categorical target by logit scores and cross entropy loss function - for mathematical reasons - always produces a higher goodness of fit than modeling a categorical target by 0/1 dummies and MSE loss.

Model Rank Scale Variables

Ordinal measurement scale variables or simply rank variables are self speaking. They contain ranks of some kind.

A rank variable contains more information than a categorical variable. Additional to the category class they also do provide the rank order among the classes. A rank variable contains less information than an interval scale, as it misses the equal distance property between the values.

Unfortunately, there very little mathematical methods in place to process a rank scale variable truly in rank scale, that results in a standard approach and a workaround approach.

- In the standard approach, a rank variable is downgraded to categorical scale for proper processing (, because of the information hierarchy in the levels of measurement).

- In the workaround approach, a rank variable can be upgraded to categorical scale. For approximately proper processing however, the loss function must to be adopted from L2 loss to L1 or log-cosh, because there is no equal distance property between the values. This is basically the same as in quantile regression, where OLS regression gets replaced by L1 regression.

Model Poll And Survey Scores

Poll and survey scores are tricky statistics data. Usually they are created from “0= Do not have at all/ do not like at all” up to “10= Do have 100%/ do like most”. When the data is loaded into a Pytorch tensor the former integer value get automatically converted to decimals like [0.0, 1.0, …, 9.0, 10.0]. Thus, on the first view they do look like numerical data, but they are not.

Poll and survey scores are rank scale variables. Every participant in a poll has a different individual understanding of “do not have at all/ do not like at all” and “do have 100%/ do like most”. Thus, there is no equal distance property between the values.

Model Numerical Scale Variables

True numerical scale variables are typically taken as is. They are not transformed to another scale.

Loss function choice on a numerical target.

- L2, L1 or log-cosh loss function
  - 90% of all data scientist simply choose L2 loss function (squared error/ MSE). Simply for its ease of optimization.
  - Following economic decision theory, on contrary, the appropriate loss function has to reflect the model user’s decision criterion between the options. In a single choice out of a bundle of predicted options the user’s risk aversion is best reflected by squared error/ MSE. However, there is no such pertinent loss function recommendation, when the model user wants to select a whole portfolio of decision options based on the model predictions. E.g a financial investor trying to select the 20 most underpriced stocks or a farmer trying to choose the most yield promising crops for his 10 fields.
- Loss functions as observation weights
  - When dealing with outlying value observations in practice, the choice of the loss function has huge impacts on the model parameters. (In regression this issue is addressed as influential hat values.)
  - In this regard a loss function can be viewed as a weighting of the observations. L= sum [some_weighting *(Y_t -Y_p)]
  - A squared error function will weight an outlying observation magnitudes higher than an L1 loss function!
  - Thus, it is important to know the loss function formula to understand the implicitly performed weighting.

In Pytorch we prefer meaned loss functions (e.g MSE) over summed loss functions (e.g SE). While it does not make any difference for an analytical optimization of a loss function, the

choice of meaned or a summed loss function does make a remarkable difference in Pytorch. Whenever the loss function significantly changes in niveau, the learn rate of an optimization algorithm requires adoptions. Thus, niveau stable meaned loss functions are much more applicable.

Relative Error

Let us discuss the proper error modeling of the sold units quantity in a sales forecast model. In a DIY superstore there are many articles, that sell in high frequency at low prices, e.g. screws and nails. On the other hand a large DIY superstore also offers high value articles with low selling frequencies. E.g. professional electric generators for about $ 5000 each. Usually there is in average just sold 1 generator unit per month. If the sales quantities were modeled as absolute error in a forecast model, the model will mostly be trained on the “high frequency low price articles.” But the sell of one generator creates much more profit than the sell of 100 screws. To get a fair neutral sales forecast model for all articles, low price and high price ones, the forecast errors must be modeled as relative error. Quantities like this at the same time are often “ratio scale variables” following Stevens’ levels of measurement systematic.

The easierst way to accomplish relative error is an initial logarithmic transform of the target variable. This way all the standard absolute error loss function models can get applied without any adoption, because ln(a) -ln(b)= ln(a/b) . The logarithm transform trick does only work straight forward, if the target variable does not contain zero values or negative values.

In our Celsius to Fahrenheit case the Fahrenheit target is strictly positive in the measurement sample. But theoretically, e.g. in strong winters, Fahrenheit temperatures could also get negative. Thus, we model the errors of the Fahrenheit target as standard absolute error, not as relative error.

Further Study Advices

- Research the mathematical formulas of the loss functions on the internet and take the formulas down to your learn notes.

Goodness Of Fit

We do not want the mean squared error (MSE) as prediction quality metric, any more. Anytime we do transform the target variable, the MSE jumps to another level. So, we cannot compare the quality of models with different target variable transforms by it. Instead, we want a real standardized goodness of fit metric as prediction quality gauge, so that we do get a proper orientation about, how much of the pattern/ the formula hidden inside the data got extracted by the model. For a numerical target variable (in our example the Fahrenheit value), R-L2 (a.k.a. standard R-square) is the most important goodness of fit metric. It is worth mentioning here, that maximizing R-L2 and minimizing MSE both lead to the same (neural network) parameters, because R-L2 and MSE are both based on the same variance core loss function to evaluate the deviations between the predicted value and the true measurement value.

At loss functions above we have seen, that there are different types of loss function, that just differ in the power of the error accounting (the p-norm of the error). When the p-norm is 2, it is an L2 loss function, when the p-norm is 1 it is a L1 loss function.

This can also be accomplished on the famous R-L2 (standard R-square) goodness of fit. So, there is a standard R-L2 goodness of fit and alternative R-L1 goodness of fit, with an error evaluation by the power of 1.

A model, that has been trained with L2 loss, will usually reach a higher R-L2, than an equal powerful model trained with L1 loss. And a model, that has been trained with L1 loss, will usually reach a higher R-L1, than an equal powerful model trained with L2 loss. Thus, when ever we turn the loss function to L1 loss (or log-losh loss), we usually also turn to L1 goodness of fit.

Code

Fig. 13: Cod. neuralnetwork06.py

Fitness Test And Learn Stop

Now, we have got a goodness of fit metric. So, the next thing we need to do is to get the goodness of fit metric into operation. Specifically we want to *measure the current fitness of the model on the unknown test data* and we want to stop the learn process before a significant overfitting takes place, namely the goodness of fit computed on the test data decreases.

To accomplish the learn stop, we require the following things.

- A train and test split of the data set. (Not an ideal random one, yet. But at least there is one. In a lecture we want to get reproducible results, right?)
- An automatic test procedure, that stops the learning, when the gain in goodness of fit computed on the test data gets very small or even negative.
- The test procedure must tolerate an initially erratic learn progress. We choose a hurdle level of R-L2= 0.2, up to which to allow for erratic learning.

Some further little gimmicks also get implemented on the go.

- Add loss_hist and gof_hist to view into the historic losses gofs, if needed
- Add the measured gof to the save model name

Now, when a relevant overfitting will take place for the first time, namely the goodness of fit computed on the test data decreases, the learn process will be stopped right there, then.

Code

Fig. 14: Cod. neuralnetwork07.py

Further Study Advices

- In the code example above the goodness of fit is tested only every 100 epochs. Why is that? (Hint: Stochastic gradient descent algorithm.)

Neural Network Architectures

Popular Neural Network Types

As of 2025 the most popular types of neural networks are the following ones.

Type	Standard/ Special	Major Application
Feedforward backward propagation	Standard	E.g. normal data (with categorical/ rank/ numerical target)
Convolutional neural networks (CNNs)	Special	Image classification
Generative Adversial Networks (GANs)	Special	Image generation
Transformer Networks (BERT, FastBERT, GPT, …)	Special	Language processing

Fig. 15: Tab. Popular Neural Network Types

For any application case, for which there has not a special network type become pertinent yet, the standard “feedforward backward propagation network” remains the relevant type, that will be applied on default.

This chapter is going to discuss the architecture of plain vanilla “feedforward backward propagation networks”, only. The lecture is - and should remain - a compact lecture. But although it only covers standard networks, explicitly, some of the insights can also be generalized onto the specialized complex network types.

Number Of Neurons Per Layer

Input Layer

The number of neurons in the input layer equals the number of input variables.

Output Layer

- On a numerical prediction target, the number of neurons in the output layer equals the number of predicted output variables. Typically, just one output variable is predicted.
- In a classification prediction on a categorical target, the number of neurons in the output layer equals the number of classes (categories) of the categorical target variable.

Hidden Layers

- Basically, a hidden layer can contain any number of neurons.
- The more neurons a hidden layer has, the more flexible and the more powerful a neural network gets. A neural network with a larger number of hidden neurons can capture more

complex patterns/ formulas, that are hidden in the data.

- At the same time a more complex neural network tends to overfit more easily.
- The total number of neurons is always chosen in regards to the available computing power. A large neural networks requires more iterations (epoches) till the final maximum fit, than a small one. The number of required training iterations (epoches) increases exponentially by the total number of neurons, in a fully connected neural network.

Architecture Optimization

Neuron Distribution On Multiple Hidden Layers (Thumb Rule)

- Usually, the first hidden layer is the largest hidden layer, in a well fitting neural network.
- Usually, any following hidden layer is smaller than the previous one, in a well fitting neural network.
- Usually, the optimum number of neurons in the following hidden layer counts around 50% of the previous one, in a well fitting neural network.

Thus, the hidden layer architecture of a well fitting multi-layer neural network usually looks like a fir tree.

Simple Architecture Optimization Heuristic

A heuristic is a basic, usually recipe like procedure to find an acceptable solution for a problem, for which either a mathematical optimum solution is not known or too complicated to apply.

Follow the goodness of fit heuristic

0. Start with a small single layered neural network, of which number of hidden neurons can be formulated as an exponent of 2 (2**x). E.g. 8, 16 or 32.
1. Try to introduce a new hidden layer without increasing the total number of hidden neurons. Redistribute the number of neurons on all layers this way, that the number of neurons in each following layer counts around 50% of the previous one. If the new architecture generates a higher fit, keep the new hidden layer.
2. Try to increase the total number of hidden neurons by factor the 1.5. If the new architecture generates a *significantly higher* fit, keep the new number of hidden neurons.
3. If the network architecture has been changed in the current adoption cycle and the maximum number of training epoches is not reached yet, go to to step 1 again.

During last 10 years cloud computing has become very fast, cheap and easy. Thus, the neural network architecture question has almost fully lost its relevance in normal data prediction models (categorical and numerical target variables).

In image recognition problems the architecture question is still relevant. However, the average money spending of an image recognition customer for additional goodness of fit is lower. Often they simply do not like to invest another $5’000 for 1% better hit rate.

In very large neural network models however, the optimum network architecture question is still highly relevant, e.g. in large language models.

Unfortunately, the running code example is too small to show the simple architecture optimization heuristic.

Architecture Optimizer Tools

Till about the last 10 years, there is a low, but steady output of neural network architecture optimizer tools. Some of them are commercial, non-oss solutions. As far the author has read, an architecture optimizer tool can generate a high performance plus versus the simple architecture optimization heuristic, if it

- is based on a mathematical optimization strategy. (Primary criterion)
- additionally can provide a specific optimization strategy for the type of neural network, e.g plain vanilla, CNN, GAN or transformer network. (Secondary criterion)
- additionally can provide a specific optimization strategy for the type of prediction, e.g classification, object detection or semantic segmentation. (Tertiary criterion)

Chapter 6 - Learn Problems And Optimization Choices

Basically, there are two different strategies to handle learn problems in neural networks.

- Knowing about the relevant math behind the neural network, you are dealing with. The mathematician’s approach. Best option, but out of the scope of a compact lecture.
- Knowing the character of each learn problem and the solution, how to fix it. The practitioner’s approach. Second best option and inside the scope of a compact lecture.

Recapture

- Each optimization algorithm has an individual learn rate window for each new learning task, out of which the algorithm does reach the minimum of the loss function (or close beyond). But outside this learn rate window, the minimum loss point (or close beyond) will *not* be found.
- To make the task even more challenging, the learn rate window of each algorithm has also different width. The good ones, have a more broad learn rate window.

Learn Problems

- The algorithm misses the loss minimum and always ends at a very low goodness of fit niveau, maybe even negative.
  - Solution: Decrease the learn rate.
- The algorithm makes only very slow goodness of fit progress.
  - Solution: Increase the learn rate.
- The algorithm gets stranded in a local minimum or saddle point (rare case).
  - Solution 1: Perform multiple trainings and store the reached goodness of fit and the parameter values. Finally choose the stored neural network model with the best goodness of fit.
  - Solution 2: Increase the size of the hidden layers. Initially, the network parameters are in a randomly initialized. Thus, the more parameters there are neural network, the less are the chances, that all parameter derivations do strand at the same time in a (small)

local minimum.

Heuristic Optimization Choices

A heuristic is a basic, usually recipe like procedure to find an acceptable solution for a problem, for which either a mathematical optimum solution is not known or too complicated to apply.

Remember the sections “Neural Network Architecture” and “Loss Functions”. There are already a lot of choices in the neural network hyperparameters, already.

And there are further choices to do for the optimization algorithm (the optimizer) and for the activation function. To make it worse, usually also the learn rate needs to be re-adopted, when switching from a limited activation function to an unlimited one, or vice versa.

All these choices do easily soon multiply up over 100 choices in total, one cannot/ does not want to test all through. Thus, a heuristic for choosing the the optimizer and the activation function is very welcome, in order to keep the total number of choices low.

Optimizer And Activation Interdependence Problem

Often, there is an interdependence between the optimization algorithm and the activation function regarding the final goodness of fit. E.g.

1. rank: optimization algorithm 1 - activation function 1
2. rank: optimization algorithm 2 - activation function 3

Heuristic Optimizer And Activation Choice

1. Create an activation function free network and test through Adam, Adagrad, SGD and Rmsprop algorithm.
2. If a learn problem appears, adopt the learn rate as recommended in section “learn problems.”
3. Choose the best performing optimization algorithm and keep it fixed.
4. Fill the neural network with activation functions again and test through all the important activation functions mentioned in section activation functions.

The author has observed in his programming practice, that neural networks with activation-less hidden layers (at maximum containing the predetermined activation functions) very often create similar ranks in the optimization algorithms as the average optimization algorithm ranks from a full algorithm X activation trial series.

The presented heuristic is a massive time saver. When altering the activation function(s), especially when changing from an unlimited activation function to limited one or vice versa, usually also the learn rate of the algorithm needs to be re-adopted.

In about 85% or 90% of the data cases the Adam algorithm will turn out the best algorithm. So, many AI programmers simply fix plug-in the Adam algorithm. However, there are data cases, where the Adam learn rate at the loss minimum is pretty high. Then Adam looses more goodness

of fit during the fitness test cycle and mostly Adagrad gets the best performing algorithm, then.

Final Solution Of The Celsius To Fahrenheit Case

The ADAM optimizer has won the activation-empty competition with significant distance to the next best optimizer. Thus, the ADAM optimizer has been chosen.

Second, all “ADAM optimizer - activation function” combinations have been tested through.

Out of this, the combination

- ADAM optimizer - Softmax activation function

turned out best.

Code

Fig. 17: Cod. neuralnetwork08.py

The ADAM - Softmax combination performs on my machine with an R-L2 goodness of fit in the interval ~ [0.898; 0.942].

It is also the combination generating the highest average R-L2.

In a neural network training designated for production we of course have to compute the final R-L2 goodness of fit from the randomly chosen validation subset.

Unfortunately, random data sets do not create reproducible results, thus are not very suitable for teaching. Away from that, the simple running code example with the Celsius to Fahrenheit case has - hopefully - done a very good job to get to know the Pytorch learning elephant in small, well digestible slices.

Further Study Advices

- Try through the “optimizer - activation function” combinations with Adagrad and Rmsprop algorithm, that have been left out by the heuristic. Do you find a combination performing better than the “ADAM optimizer - Softmax activation function” combination?

Learn Progress Feedback

By now, you have already accomplished approximately 90% of the Pytorch fundamentals. By now, you already do know about

- the important tensor syntax elements
- the important syntax elements for creating a neural network
- the important network hyperparameter choices
- the most frequent learn problems and heuristic optimization choices

There is just one further chapter left. After taking the last chapter of the lecture you will - hopefully - stand right in the Pytorch starting block, ready for your personal Pytorch journey.

Of course, you are still invited to report any error or improvement suggestion regarding this chapter or the one before. The address is: github.com/quantsareus/Pytorchcompactlecture/issues. Where have you got stuck and why? In case, how can this section get explained better? All the reports will be collected in the central issue database, be thoroughly reviewed and - if possible - the issue will be fully solved or mitigated in the next edition of the book.

Chapter 7 - Load And Prepare Data

Load hdf5 Files

The Hadoop hdf5 format is the standard format in Python to save and exchange big data files.

The setup is

```
conda install h5py
```

respectively

```
pip3 install h5py
```

```
#Saving
import h5py

f= h5py.File('./datafile.hdf5', 'w')
f.create_dataset('coords', data= points.numpy() )
f.close()
#Loading
import h5py

f= h5py.File('./datafile.hdf5', 'r')
dset= f('coords')
points= torch.from_numpy(dset)
f.close()
```

Load CSV Files

Small and medium data sets, however, are mostly saved and exchanged in the comma separated values format (CSV).

Introduction Of The Wine Quality Case

The wine quality dataset (archive.ics.uci.edu/ml/datasets/wine+quality) is a very popular dataset in the Python based data science community. There is a lot of code available for this one on the internet, thus a lot of free additional support. The data set consists out of 11 numerical input variables from a chemical analysis and a sensory quality score from a degustation (a wine tasting) in column 12.

Basically, there two major options in quantitative Python to load data sets.

- numpy.loadtxt()

- pandas.read_csv()

A CSV file is in detail a usual text file, that contains the values of the data table line by line. Within a data line each data value is delimited from another one by comma, semicolon, tab or space. So, every csv file has to be checked upfront for the used delimiter.

The values between the delimiters can contain literals of any data type. Thus, it is much more safe to load a csv file initially into a pandas dataframe. A pandas dataframe can carry columns out of different data types. A numpy array on contrary, can only consist out of one single data type. Thus, e.g just one single character value “A” would cause the whole numpy array to be casted automatically to character; that is undesired.

The pandas function for loading a data set is pandas.read_csv(). It has a header argument to ignore some initial non-data lines for import. The other arguments are self speaking.

```
pandas.read_csv("pathto/file.csv", delimiter= ";", header= 0)
```

Feature Engineering Of A Rank Scale Variable

The wine quality target, column 12 in the wine quality dataset, has been measured as a sensory quality score from a degustation (a wine tasting). Quality marks, theoretically ranging from 1 to 10, are most comparable to a poll and survey score. Thus, the wine quality is best treated as a rank scale target variable. (Compare section “Model Poll And Survey Scores”.)

For the processing of the rank scale target variable, the standard approach is chosen, downgrading to categorical variable. (Compare section “Model Rank Scale Variables”.)

Category Column -> Dummy Columns

There are two pertinent transform operations in place to perform this task.

1. torch.scatter_()
2. pandas.get_dummies()

The two transform operations do behave differently.

1. In `torch.scatter_()` the total number of categories has to be specified upfront. Further, for each upfront defined category a corresponding dummy column is created, regardless if the specified category feature effectively exists in the data.
2. In `pandas.get_dummies()` the total number of categories is computed dynamically. Only category features, effectively existing in the data, get created as a dummy column.

Usually, the transform option 2 fits a data scientist’s needs better.

`Code`

Fig. 18: Cod. prepare-dset01.py

Dummy Columns -> Category Column

The - much more less needed - reverse operation is among others provided by

- sklearn.preprocessing.LabelEncoder()

Data Split Into Train, Test and Validation

Quantitative Python has a lot of built-in possibilities to split a data set into a train, test and validation subsets. Among others `torch.utils.data.Subset()`. However, most experienced programmers do not use them.

First of all, the naming of the data sets in Python is wrong. The data subset used for testing the current fit during the training process has of course to be called test set. And the subset required for the validation of the whole final model of course must be called validation set. Second, the built-in procedures do effectively split into 3 data sets; but handling the data partitions within one data set is much more practical.

So, the author shows the handy practice to create a new data column, only, that keeps the train, test and validation partition information. This way the programmer has to wrangle around with less data sets. The easierst way to accomplish it, is to generate a vector of random numbers, first. In the next step, the random number vector is used to assign the data partition information.

- “1” = “train”
- “2” = “test”
- “3” = “validate”

The subsets train, test and validation should get sized in the ratios 4 : 2 : 1. So, the effective sizes are 4/7, 2/7 and 1/7.

`Code`

Fig. 19: Cod. prepare-dset02.py

Further Study Advices

- Search the internet for further Python methods to split a data set into a train, test and validation and choose the one, that fits your personal needs best.

Pytorch Dataset Class

Loading and converting data, that is ready for training can often end up in a tedious work, that consumes a lot of time. Especially, different types of models and even some loss functions do require a specific data preparation. In daily data science practice, however, a data developer - when having tried a second model - wants to switch easily back to the first one, *including the specific data preparations of the first model.* And straight forward also switching to a third model, *including the specific data preparations*, and back.

Therefor, Pytorch has developed the class `torch.utils.data.Dataset`. Especially, it allows to build a parametric converter class for a dataset. So that, the columns of the source data set come out of the custom dataset class, just like they are needed for the model. The custom dataset class is a versatile, absolutely fantastic and very widely used feature of Pytorch.

Converted Custom Dataset

Typically, a custom dataset class derived from `torch.utils.data.Dataset` has at minimum to contain

- a `__len__()` method, that returns the size of our dataset (len) and
- a `__getitem__()` method, that reads and returns a data line.

The `__getitem__()` method also must include a Python iterator object, in order to grant access to a single element of the dataset object. Either an index or a map() object. Indexes are most common.

```
class CustomDataset(torch.utils.data.Dataset):

    def __len__(self):
        pass

    def __getitem__(self, index):
        pass
```

So, we are writing a CustomWine class with a parameter for train, test and validation data partition. In the first development state, the CustomWine class should return the input values as is for x and the already transformed dummy values of the target variable “quality” for y (not the original values).

`Code`

Fig. 20: Cod. prepare-dset03.py

A parametric CustomWine class with parameters for train, test and validation data partition, that returns synchronous input and target values for x and y on automatic, already saves a lot of tedious work.

But when we want to use the crossentropy() loss function, we have to feed it with logit scores instead of dummy values. However, when performing a pseudo-metric prediction, we do require the original quality categories. So we implement a further parameter y_out, in order to switch the y output.

Additionally, we also want to switch between “normed x values” and “as is x values”, in order to examine the prediction quality gain from norming. This is implemented by the parameter x_norm.

Last but not least we need the whole training stuff again, just like before in the Celsius Fahrenheit case.

Final Solution Of The Wine Quality Case

`Code`

Fig. 21: Cod. prepare-dset04.py

Maybe you find the whole data set customization by the CustomWine class a little bit confusing at first. But now we can switch between

- an MSE error model on target dummies,
- a cross entropy model on logit score, and
- a pseudometric MSE error model on the original category values

within 30 seconds. And this is, what a quantitative developer needs in practice. If you truly become a Pytorch developer, you will use this a lot! It keeps the whole code of the modeling project much more compact and maintainable.

Lazy Loading Of Big Data

Pytorch is very often used as big data/ AI computing framework. Big data can come as one very large data set (e.g. one 100GB dataset). When the whole data set does not fit into the memory, a special loader procedure is required to load the data iteratively in small portions (batches) onto the memory (and the CPU) for training. This is called “lazy loading”.

But big data can also be delivered as an ensemble out of thousands/ even millions of single data items. E.g a wildlife cam takes an image every 5 seconds and stores them in a year/month/day/hour folder structure. Then, the data must be collected and loaded lazy in small portions onto the memory (and the CPU) in one go.

Lazy loading is performed by three main components.

- A lazy Dataset() class derived from torch.utils.data.Dataset [as foundation object]
- the iterator of **getitem**() method in the CustomDataset() class [as low-level access to an element in the virtual data set]
- torch.utils.data.DataLoader() [for effectively loading]

The main difference between a lazy Dataset() class and a custom Dataset() class is, that the loading is performed within the **getitem**() method [instead of within the **init**() method].

Sometimes it is also possible to use the default class `torch.utils.data.TensorDataset` as basis for Dataloader(). But usually an individual one is created.

Lazy Loading Of JPG Images

The following code performs a lazy loading of the jpg images version of the Mnist data set. The jpg images are contained in a folder structure, that has to be processes properly. So, the important step is to assemble all the file pathes to an iterable structure.

`Code`

Fig. 22: Cod. readmnist-fromjpg.py

Lazy Loading Of Byte Data

There is also a precompiled ubyte version of the Mnist data set. Ubyte files are binary files, into which the data from ensembles out of multiple items have been gathered for easier processing. Typically, they do contain exactly the same bytes as the multiple items. This is a very good occasion to learn how to read Python byte data objects on the go. Of course, also Python byte

objects can be loaded lazy using a file descriptor.

`Code`

Fig. 23: Cod. readmnist-fromubyte.py

Pytorch Online Data Set Library

Pytorch Org provides three online data set libraries

- pytorch.org/vision/stable/datasets.html
- pytorch.org/audio/stable/datasets.html
- pytorch.org/text/stable/datasets.html

Popular data sets of the vision library are

- MNIST,
- Fashion-MNIST,
- KMNIST,
- EMNIST,
- QMNIST,
- FakeData,
- COCO,
- Captions,
- Detection,
- LSUN,
- ImageFolder,
- DatasetFolder,
- ImageNet,
- CIFAR,
- STL10,
- SVHN,
- PhotoTour,
- SBU,
- Flickr,
- VOC,
- Cityscapes,
- SBD,
- USPS,
- Kinetics-400,
- HMDB51,
- UCF101,
- CelebA

Online Access Library

The data sets of the Pytorch online library can be accessed remote by

- torch.utils.data.Dataset

- Torchvision Datasets

You do find a lot of code examples on the internet how to achieve that.

The important point here is, that they do exactly behave like a virtual custom/ lazy Pytorch dataset. In case you do experience any problems with `DataLoader()` on any remote data set from the library, you have to read up the definition of its `CustomDataset()` class. In some data sets the data set elements are not iterated by an index but by a map.

Learn Progress Feedback

You have made it! You have accomplished 100% of the Pytorch and neural network fundamentals. Now, you do know about

- the important tensor syntax elements
- the important syntax elements for creating a neural network
- the important network hyperparameter choices
- the most frequent learn problems and heuristic optimization choices
- how to load and prepare data; even big data sets, that do not fit into the memory of the machine.

These skills should enable you now to start your own personal Pytorch journeys. You should able to study and investigate most Pytorch projects from Github on your own, now. Of course a large portion of this projects is build on some math in the background. But you do not need to understand the full math, as far your desired application is pretty similar to a prototype project from github. Anyway, the time required to invent/ modify a mathematical engine usually exceeds the maximum project costs. Thus, very often the realization of a modeling project stands and falls with an available prototype for this kind of problem.

From the author’s point of view, almost anyone gets a much better starting block place in this quantitative topics - respectively in the programming part at least - by a didactic well concepted lecture, than by a random lecture cookbook.

This is the last chapter, already. So, it is maybe the last opportunity to report any error or improvement suggestion regarding this chapter or the whole concept of the compact lecture in general. Otherwise, you will most probably turn to other tasks and your brilliant idea will get lost. The address is: github.com/quantsareus/Pytorchcompactlecture/issues. Where have you got stuck and why? In case, how can this section get explained better? All the reports will be collected in the central issue database, be thoroughly reviewed and - if possible - the issue will be fully solved or mitigated in the next edition of the book.

If this learning script was helpful for your neural network/ AI/ big computing programming journey:

DO NOT FORGET TO RECOMMEND IT TO OTHER INTERESTED PEOPLE!

And, once you have become a successful programmer and you are going to reach the age of 50:

DO NOT FORGET TO SHARE YOUR WORTHY KNOWLEDGE AND WRITE YOUR OWN

Chapter A - List Of Critical Terms And Wordings

- A “stochastic variable” is one characteristic of a thing or one characteristic of the result of an experiment.
- The “features of a stochastic variable” are the outcomes of that variable observed in an experiment or measurement series, that further following do appear as “data values” on the data set.
- The “input variables of a data set” or just the “inputs” are the variables used to compute the prediction from.
- The “target variable of a data set” or just the “target” is the variable to predict by the prediction model.