Contents

Figures

Draft Version For Peer Review !!

This is just the draft of the learn script. It's purpose is, to get it reviewed for global legislation and compliance check by designated peer-reviewers. If you are not such designated, globally authorized peer-reviewer, you will not have very much fun with the draft, I guess. Some central parts are still missing and the layout of the draft is - humble speaking - still very provisional. For studying you have to buy an official version. After the obligatory legislation and compliance check period of 2 weeks, the draft gets offered to globally authorized publishers. I am very convinced, the learn script will soon be available in most official, well ordered book stores and at online book sellers. This is higher politics out of my reach.

In case found global legislation and compliance issues, and all other found issues, should be reported as standard github project issue under

gihub.com/quantsareus/PyTorch-learn-script–draft-for-designated-peer-reviewers-only

The deadline for global legislation and compliance issues ends on 2025-02-28 23:59:59 GMT.

Delayed filed global issues will be worked on a best effort basis.

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 are considering to spend a lot of time in learning the PyTorch framework. If you do not know well, you will not be fully motivated for learning. So, you want to know this in advance.

Unfortunately, at the very beginning of a PyTorch book the statement of the benefits 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. Feel free to research further on the internet and to validate, so that you do find the best neural network and AI framework option for your individual needs.

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 modern than “tensor flow”. “tensor flow” code is much more technical than PyTorch code. “tensor flow” was published by Google in 2015. The first stable marked version of PyTorch was published in 2017 by Facebook. The PyTorch developers have intensively learned from the experiences with the “tensor flow” and the “paddle paddle” framework, and have included a lot of improvements into PyTorch. Thus, PyTorch is much more handy and much more easy to learn than the other two.
- 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. Also Numpy data arrays can be seamless integrated into a PyTorch program.
- PyTorch can be executed either on CPU or on NPU/ GPU, once the proper driver framework of the hardware is installed, such as cuda. This makes PyTorch much more than an AI framework. It is also a general big computing framework. E.g just to run a very large matrix multiplications on a high performance cluster.

From my point of view, you should figure out now, what **exactly** do **you** need more than that.

Didactic Concept

First of all, I want to emphasize, that **the design of this learn script does follow a refined didactic concept**, while most - currently popular - programming *cookbooks* mostly just present performed projects an experienced programmer has in his/ her pocket, filled up with some theory, that gets more or less randomly dropped, just where the code element appears in the project code. While on the one hand there of course is some demand for showing successful projects to *experienced programmers*, on the other hand a giant spoiler effect takes place, when test-reading a programming *cookbook* in a bookstore or on Amazon for 5 minutes: Hey, look here, xyz programming is so easy! Buy this book! But when effectively reading the cookbook later on at home, most beginners do get stuck in chapter 2, already, because he/ she does not have the theory background and the cookbook does not provide a helpful learning structure to achieve one.

Major Didactic Elements Of The learn script

- 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 learn script 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 learn script. 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 Learning Process”, “Neural Networks”, “Neural Network Hyperparameters” and “Learning 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 learning 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 learn script is just about 1.5 : 1. The explaining prosa is on purpose kept short onto the point, in order to speed up the consumption.
- Further, the explaining prosa is often provided in bullet point list form, where appropriate, also to speed up consumption.
- The learn script covers the PyTorch syntax and the PyTorch handling up to intermediate level. After taking the lecture of the learn script 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 learn script, you are getting some ideas, how to make them more effective.

You may have recognized from above already, that I have written the learn script to teach PyTorch to interested people fast, not for money in the first or in the second place. I have done it, because I consider PyTorch to be the best neural network and AI framework, currently.

The Proceeding Of The Learn Script

- Chapter 2 “PyTorch Tensor” contains the syntax part. Especially the syntax regarding to the PyTorch tensors, the central data structures of PyTorch.
- Chapter 3 “PyTorch Learning Process” addresses how AI learning effectively gets accomplished in ptorch by the central components. An prediction model, a loss function, an optimization algorithm and a training loop. The central study case is introduced and a

corresponding running code example is created from scratch. Finally, the running code example gets updated with the first PyTorch automatics, an autogradient and an optimizer.

- Chapter 4 “Neural Networks” is mainly about the creation of a neural networks in PyTorch, the overfitting problem and the activation functions. The model in the running code example is switched to a neural network. Further, an autolossfunction and an alternative activation functions are added.
- Chapter 5 “Neural Network Hyperparameters” begins with the implementation of a “Fitness Test And Learning Stop” procedure into the running code example. Following, the important activation function and the important loss functions are presented. Finally, the subject “neural network architectures” is discussed.
- Chapter 6 “Learning Problems And Heuristic Optimization Choices” is self speaking. First, the most frequent learning problems and their solves are presented. Then, the author introduces an universal heuristic optimization approach. Finally, the approach is applied and a final solution of the running code example is provided, that gets its long journey through 5 chapters to an end.
- Chapter 7 “Data Import”.

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 *actively engineer on your own*. PyTorch programming is not really a rocket science. (The mathematical background needed to invent a new AI learning 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 learning interruptions. If necessary, wear ear plugs

or a noise canceling head set while studying.

- Use a positive learning 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 learning 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.*

Download addresses of the learn script

Download addresses for the code examples

Setting Up PyTorch

FDSA

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 numerical 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
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 learn script.

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()**

Do remember the bold written modification methods. These have got a very special behavior, that will be investigated in detail later on.

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() # compress values into a certain interval
  - 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.

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.

Model Device Attribute

Later in the book, we will create a PyTorch model containing a neural network. We can also move a model to the NPU/ GPU using the `.to` cast.

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

In a PyTorch deep learning project, the training of the model usually consumes more than 95% of the required total computing performance. Thus, in practice often just the model is computed on the NPU/ GPU, not any tensor.

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.

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.

Learning 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 from learning. 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 give me feedback by email regarding a difficulty during this chapter. Where have you got stuck and why? In case, how can this section get explained better? All the feedbacks will be collected in a central 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.
- Saving and loading hdf5 file format. The haadoop hdf5 file format is the industry standard for big data lakes.

Chapter 3 - The PyTorch Learning Process

In this chapter you will learn about the PyTorch learning process. Armed with the tensor theory from chapter 2 we gonna start to program our first PyTorch learning process, now. This chapter contains a very simple example case, that is running throughout the whole chapter.

Example case

Professor Radiated 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 Radiated 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 Radiated'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 learning 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 learning process code.

As the book is designed as a (hopefully) well structured, learn script (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 learning 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.

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). The following code example contains a simple autogradient chain.

```
Code
```

Fig. 3: Cod.

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.

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= learning_rate)
```

Important optimization algorithms of the optim module

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

Remember

- Every optimization algorithm has some “individual base learning rate”, that either can be increased to more aggressive and fast learning or can be decreased to safe and slow learning. Unfortunately, each base learning rate depends on the data and the loss function used, thus is usually not known, initially. When the learning rate gets to high, the optimization algorithm will run totally out of the loss valley and will not find it again. If the learning gets too low, the learning 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 learning 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 learning rate window, the minimum loss point (or close beyond) will *not* be found.
- To make the task even more challenging, the learning rate window of each algorithm has also different width. The good ones, have a more broad learning 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.

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 learning 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.

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 loss_fn() and put

```
loss_fn= nn.MSELoss()
```

into the call of the training loop, instead.

```
Code
```

Fig. 7: Cod.

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.

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 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 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 code example above - *containing an absolutely bad practice regarding to production designation* - has been mentioned for didactic purposes, 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 learning 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 learning 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.

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.

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.
- Replace the author's pre-transform of the target variable y by a pre-transform using clamp(). Make proper individual clamps for the activation functions with limited maximum output values.

Learning 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 from learning. 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 give me feedback by email regarding a difficulty during this chapter or the chapter before. Where have you got stuck and why? In case, how can this section get explained better? All the feedbacks will be collected in a central 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 brainy and also more quantitative, now. We have already learned a lot of PyTorch syntax elements. When deciding how to use them properly, we need to consider more quantitative background. We will take this journey in small steps, that can be followed well. However, - depending on your personal statistical/ mathematical background - it might be necessary to read further quantitative theory up on the internet. (This is - and should remain - a learn script.)

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 learning process.

PyTorch Learning 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.

Fitness Test And Learning Stop

But before we can do that, we need to stop the learning process of the running code example on time, before a significant overfitting takes place. To accomplish the learning stop, we require a random split of the data set into train, test and validation. Respectively, into train and test a least, because the data set of our running example is just 11 observations long. We really are in delay do to the split!

Further, we do not want the mean squared error (MSE) as goodness of fit metric (prognosis 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 alternative target variable transforms by it. Instead, we want a standardized goodness of fit metric, 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 continuous 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. From the previous chapters you have still in mind, that a neural network effectively consists out of weight and bias parameters, right?

The new updated version of the running code example containing

- train and test split (not an ideal random one, but at least there is one; in a learning environment we want to get similar results, right?)
- R-L2 goodness of fit metric
- an automatic learning stop, when the goodness of fit gain computed on the test data gets very small or even negative (after a tolerated, initially erratic learning progress up to the level R-L2= 0.2)

is the following.

```
Code
```

Fig. 11: Cod.

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.)

Levels Of Statistical Measurement (Review)

Stevens (1946)	**Modern**
Binomial/ Multinomial	Categorical
Ordinal	Rank
Interval	Continuous
Ratio	Continuous

Fig. 12: Tab. Levels Of Statistical Measurement

The level of measurement of a variable is also called “the scale of a variable.”

Loss Functions

Till now I have withholded the alternative loss functions. Of course just for the didactic purpose to enable faster learning. So, we are doing this here, now.

A “loss function” is a mathematical description, how to evaluate all the deviations between each predicted value and each 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, the PyTorch nn.lossfunction() is provided.

Target Type	**Loss Function**	**PyTorch Class**	**Purpose**
Categorical	0/ 1 Hit Rate		Proportion of classified correctly
Categorical	Binomial Cross-	nn.CrossEntropyLoss()	Standard loss function

Target Type	Loss Function	PyTorch Class	Purpose
	Entropy		for binomial target
Categorical	Multinomial Cross-Entropy	nn.CrossEntropyLoss()	Standard loss function for multinomial target
Continuous	Squared Error		Standard loss function for continuous target
Continuous	Mean Squared Error	nn.MSELoss()	Squared error divided by no. observations
Continuous	L1 Loss (Absolute Value Error)	nn.L1Loss()	For special cases; undifferentiable minimum and bad convergence property
Continuous	Log-Cosh Loss		Proxy for L1 loss with differentiable minimum and improved convergence property around the minimum
Image	Cross-Entropy	nn.CrossEntropyLoss()	Standard loss function for image classification
Image	NLL Loss	nn.NLLLoss()	Special image classification; allows to handle imbalanced datasets
Image	NLL Loss 2d	nn.NLLLoss2d()	Pixel-wise classification; mostly for image segmentation

Fig. 14: Tab. List of Loss Functions

Till yet, there is no mathematical/ no economical scientific consent yet, in which prediction situations better to use L1 loss or log-cosh loss instead of L2 loss. Thus, an advice in this regard cannot be provided.

Absolute Error Versus Relative Error

In the case of a continuous target, additionally to the loss functions choice itself, the deviations between the predicted value and the true value can be either accounted as absolute error or as relative error. A relative error evaluation of a strictly positive target variable is usually accomplished by an initial logarithmic transform of the target variable. This way an “absolute error loss function” can get applied succeedingly without any adoption. [ln(a) -ln(b)= ln(a/b)]

The choice between absolute error or relative error has a huge impact on the prediction model. From the author’s personal point of view a ratio scale target variable (following Stevens levels of measurement) is modeled best with relative error. But this is not a pertinent data science practice, yet. The logarithmic transform trick does not work, if the target variable contains zero values or negative values. However, zero and negative values should not occur in a “true ratio scale target variable”. Coming back to our running code example Fahrenheit temperatures theoretically also could be negative, so they are not ratio scale, anyway.

Alternative Goodness Of Fit

Not only the p-norm of the loss function can be altered from L2 to L1. This is also possible for the goodness of fit metric. Then, the R-L2 (standard R-square) goodness of fit is straight forward transposed to R-L1.

The core loss function of the R-Lx goodness of metric can be absolute value error or squared error. So, 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. 15: Cod.

Further Study Advices

- Research the mathematical formulas of the loss functions on the internet and take the formulas down to your learning notes. When dealing with outlying value observations in practice, it is important to know the loss function formula for understanding the implicit weighting of the observations performed by the loss function. In this regard view a loss function as

L ~= sum [some_weighting *(Y_t -Y_p)]

Neural Network Architectures

Right here, we already know the syntax, how to create a sequential neural network. This chapter is going to discuss the architecture of neural networks of type “plain vanilla feedforward-backward-propagation network”, the standard type of a neural network. (The lecture is - and should remain - a learn script.)

Important Modern Neural Network Types

- Plain Vanilla Feedforward-Backward-Propagation Networks; for normal data (categorical, continuous)
- Convolutional Neural Networks (CNNs); for image classification
- Generative Adversial Networks (GANs); for image generation

- Transformer Networks (BERT, FastBERT, GPT, …); for language recognition, generation and summary

Although the lecture only covers plain vanilla feedforward-backward-propagation networks, explicitly, a lot of the insights can be generalized onto the more complex network types.

Number off Neurons Per Layer

Input Layer

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

Output Layer

- On a continuous 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

0. Start with a small neural network, of which number of hidden neurons is a multiple of 3. E.g. 9, 27 or 81.
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 two. If the new architecture generates a **significantly higher** fit, keep the new number of hidden neurons.
3. Goto 1, until the optimization loop does not increase the fit any more or the maximum computing power limit is reached.

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 continuous 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 continuous 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 - Learning Problems And Heuristic Optimization Choices

Basically, there are two different strategies to handle learning 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 learn script.
- Knowing the character of each learning problem and the solution, how to fix it. The practitioner’s approach. Second best option and inside the scope of a learn script.

Recapture

- Each optimization algorithm has an individual learning 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 learning rate window, the minimum loss point (or close beyond) will *not* be found.
- To make the task even more challenging, the learning rate window of each algorithm has also different width. The good ones, have a more broad learning rate window.

Learning Problems

- The algorithm misses the loss minimum and always ends at a very low goodness of fit niveau, maybe even negative.
  - Solution: Decrease the learning rate.
- The algorithm makes only very slow goodness of fit progress.
  - Solution: Increase the learning 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 for Optimizer And Activation Function Choice

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 section “Neural Network Architecture” and the previous one. There are already a lot of choices in the neural network architecture and also in the learning rate, when altering the activation functions, especially from limited to unlimited, or vice versa.

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

Optimizer And Activation Function 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

Free and Predetermined Activation Functions

Categorical Target

When predicting the classes of a categorical target (classification model), the last activation function is already predetermined. The activation function layer before the output has to be chosen as follows.

Target Type	**Desired Prediction**	**Activation Function**	**PyTorch Class**
Categorical	Binomial Probabilities	Sigmoid	nn.Sigmoid()
Categorical	Multinomial Probabilities	Softmaxx	nn.Softmaxx()
Image	Binomial Probabilities	Sigmoid	nn.Sigmoid()
Image	Multinomial Probabilities	Softmaxx	nn.Softmaxx()

Fig. 16: Tab. Activation Function Choice for Probability Predictions

If the neural network has more than one activation layer, all lower activation layers are free activation layers.

Continuous Target

When predicting a continuous target, all activation function layers can be chosen freely.

Choice Heuristic

1. Create an activation function free network (except for an in case predetermined layer) and test through Adam, Adagrad and Rmsprop algorithm.
2. If a learning problem appears, adopt the learning rate as recommended in section “learning 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 learning 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 learning 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 Model

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.

The ADAM - Softmax combination performs on my machine with an R-L2 goodness of fit in the interval ~ [0.898; 0.936].

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

In a real neural network training 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-Fahrenheit prediction 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?

Learning 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 learning problems and heuristic optimization choices

There is just one further chapter left. After taking the last chapter of the 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.

Of course, you are still invited to give me feedback by email regarding a difficulty during this chapter or the chapter before. Where have you got stuck and why? In case, how can this section get explained better? All the feedbacks will be collected in a central database, be thoroughly reviewed and - if possible - the issue will be fully solved or mitigated in the next edition of the book.

Chapter 7 - Data Import

Data Set

Data Set Import

Data Set Preprocessing

Image

Image Import

Image Preprocessing

Chapter A - List Of Abbreviations